Deadlock (1)

Roger Dannenberg
Dave Eckhardt
Bruce Maggs
Geoff Langdale

Synchronization – P2

- You should *really* have
  - Figured out where wrappers belong, why
  - Made some system calls
  - Designed mutexes & condition variables
  - Drawn pictures of thread stacks (even if not perfect)

Synchronization – P2

- Debugging reminder
  - We can't really help with queries like:
    - We did x...
    - ...something strange happened...
    - ...can you tell us why?
  - You need to progress beyond "*something* happened"
    - What was it that happened, exactly?
    - printf() is not always the right tool
      - produces correct output only if run-time environment is right
      - captures only what you told it to, only "C-level" stuff
      - *changes your code* by its mere presence!!!
    - We're serious about examining register dumps!
    - Overall, maybe re-read "Debugging" lecture notes

Synchronization - P2

- **Reminder - P2 Q&A day**
  - **Can be Friday – *if you bring enough hard questions***
  - **Otherwise Monday**

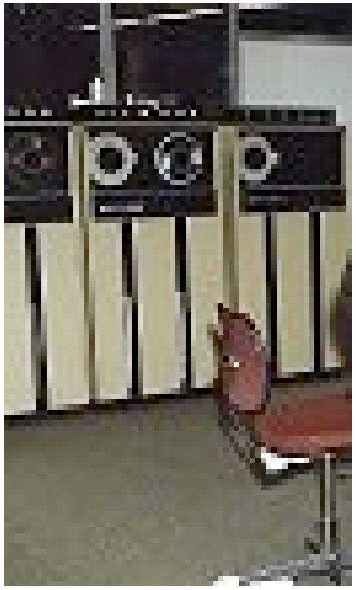
Synchronization – Readings

- Next three lectures
  - Deadlock: 6.5.3, 6.6.3, Chapter 7
- Reading ahead
  - Scheduling: Chapter 5
  - Virtual Memory: Chapter 8, Chapter 9

Outline

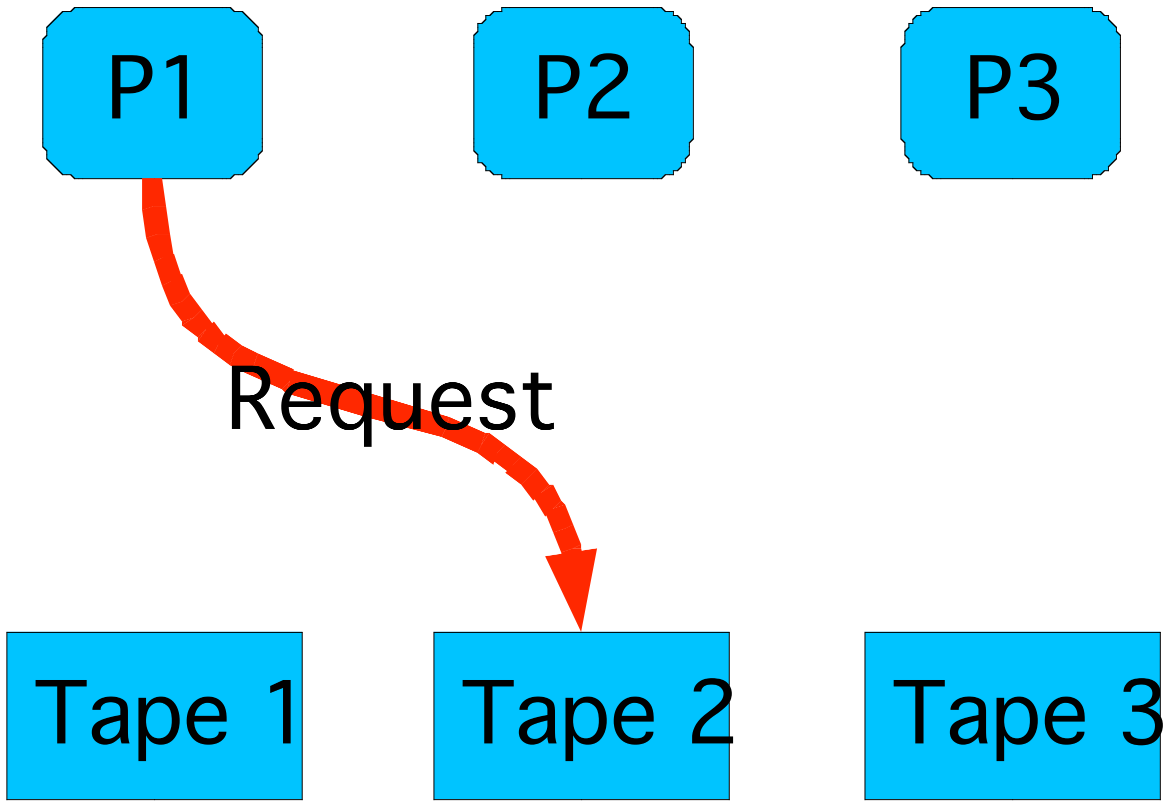
- **Process resource graph**
- **What is deadlock?**
- **Deadlock** *prevention*
- **Next time**
  - **Deadlock** *avoidance*
  - **Deadlock** *recovery*

Tape Drives

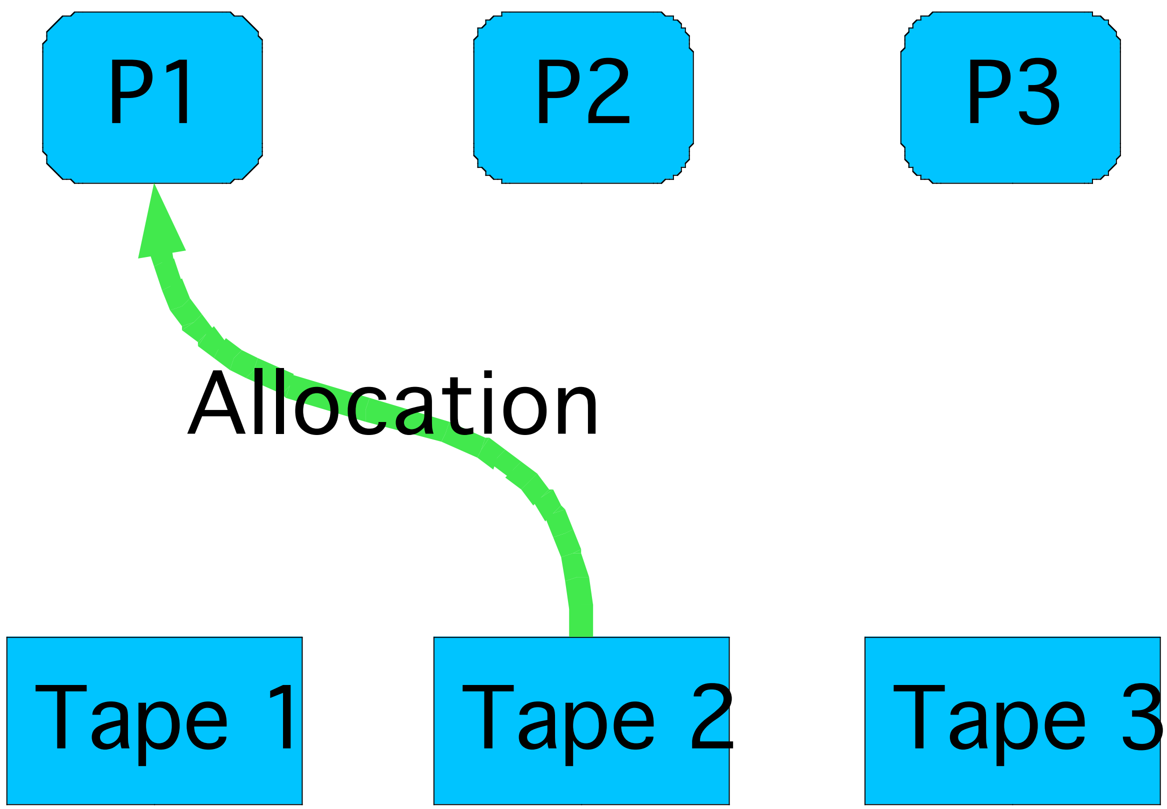


- A word on "tape drives"
  - Ancient computer resources
  - Access is sequential, read/write
  - Any tape can be mounted on any drive
  - One tape at a time is mounted on a drive
    - Doesn't make sense for multiple processes to simultaneously access a drive
    - Reading/writing a tape takes a while
- Think "CD burner"...

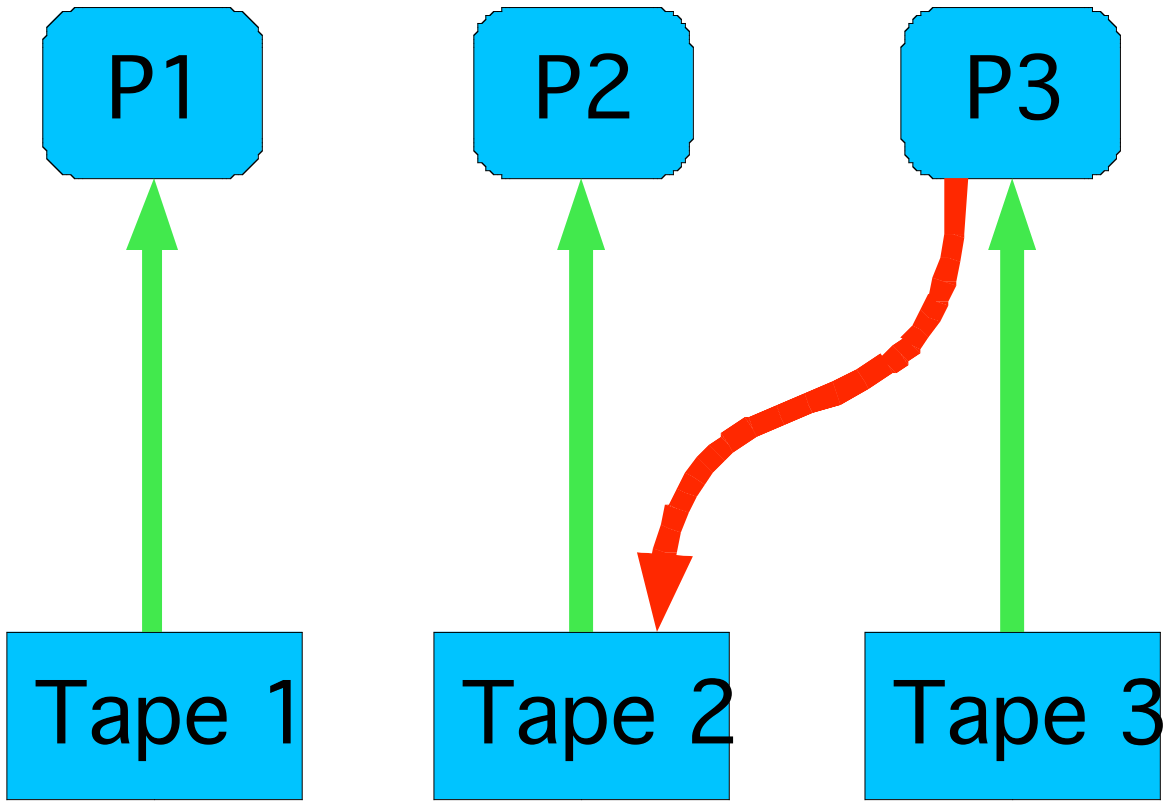
Process/Resource graph



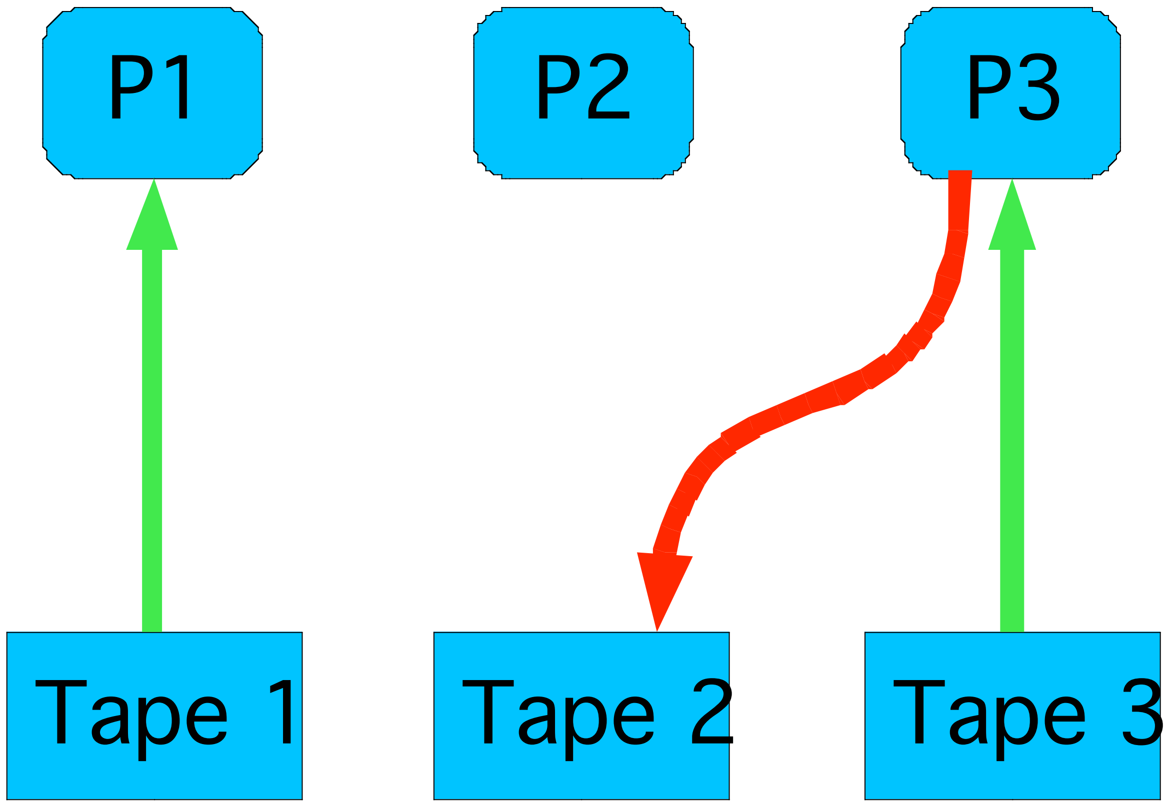
Process/Resource graph



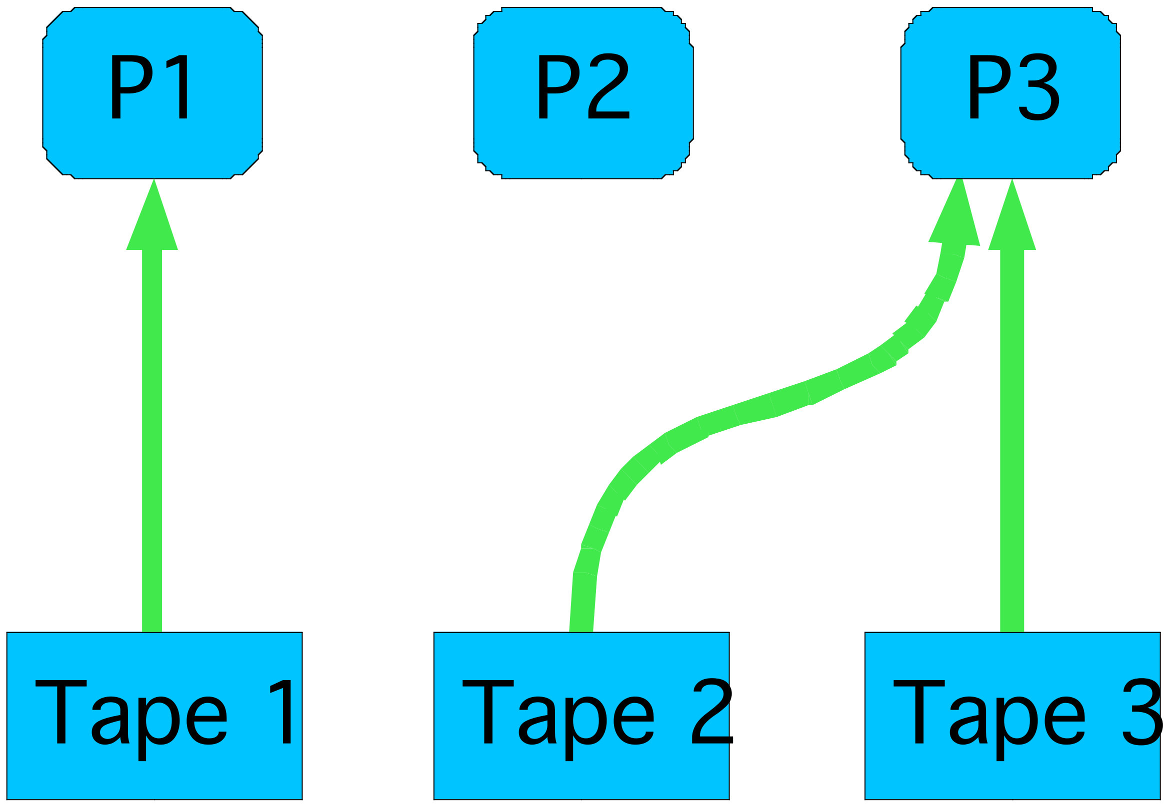
Waiting



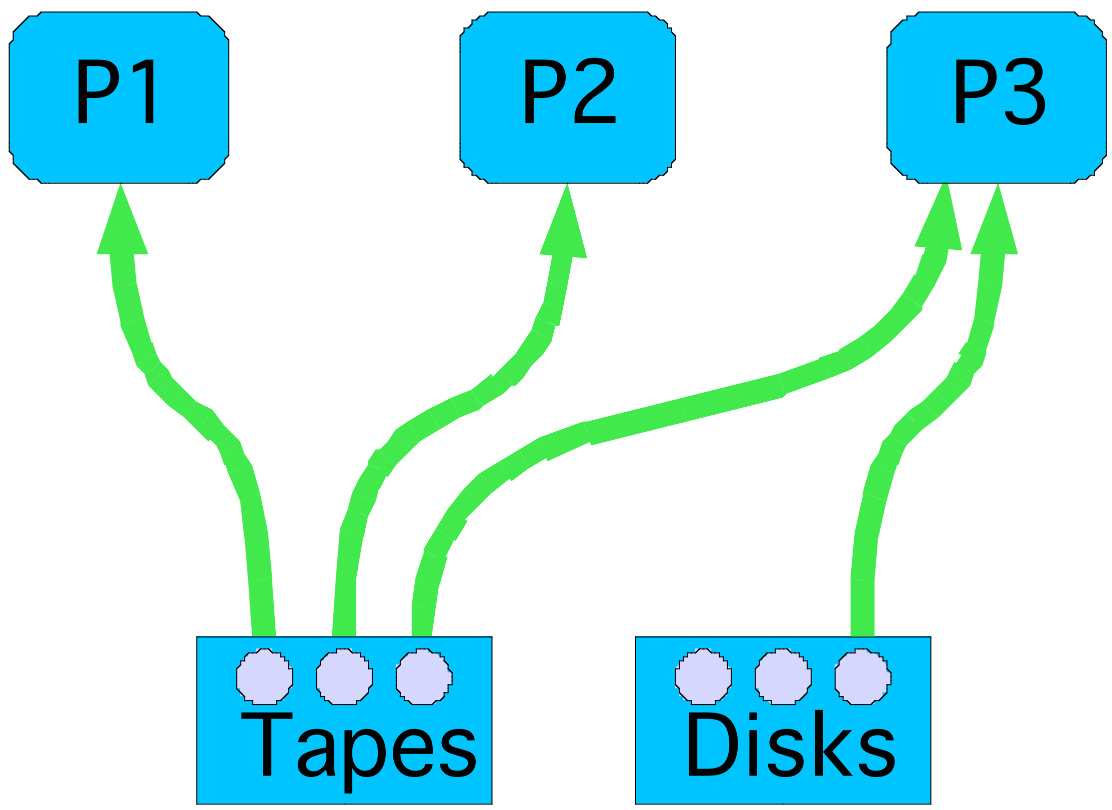
Release



Reallocation



Multi-instance Resources



Definition of Deadlock

- **A deadlock**
  - **Set of N processes**
  - **Each waiting for an event**
    - **...which can be caused** *only by another process in the set*
- **Every process will wait forever**

Deadlock Examples

- Simplest form
  - Process 1 owns printer, wants tape drive
  - Process 2 owns tape drive, wants printer
- Less-obvious
  - Three tape drives
  - Three processes
    - Each has one tape drive
    - Each wants “just” one more
  - Can't blame anybody, but problem is still there

Deadlock Requirements

- **Mutual Exclusion**
- **Hold & Wait**
- **No Preemption**
- **Circular Wait**

Mutual Exclusion

- Resources aren't “thread-safe” (“reentrant”)
- Must be allocated to one process/thread at a time
- Can't be shared
  - Programmable Interrupt Timer
    - Can't have a different reload value for each process

Hold & Wait

- **Process holds some resources while waiting for more**

```
mutex_lock(&m1);
mutex_lock(&m2);
mutex_lock(&m3);
```

- **This locking behavior is *typical***

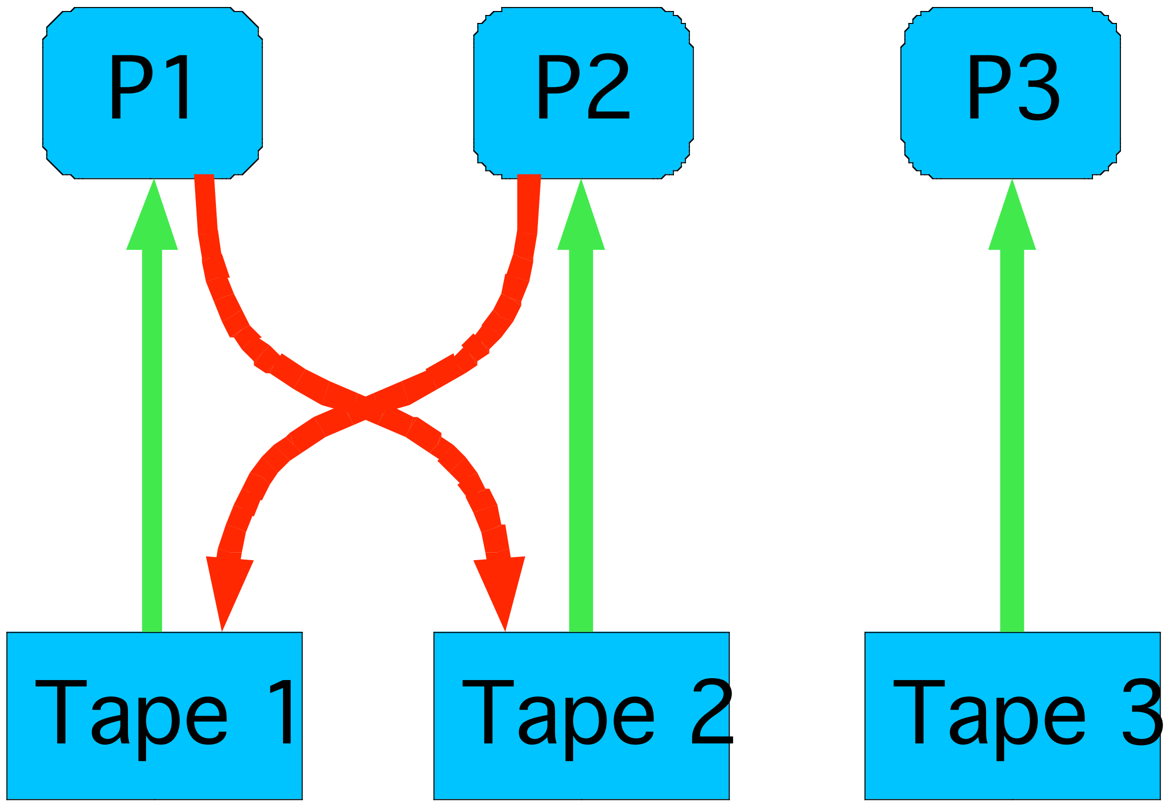
No Preemption

- Can't force a process to give up a resource
- Interrupting a CD-R burn creates a “coaster”
  - So don't do that
- Obvious solution
  - CD-R device driver forbids second simultaneous `open()`
  - If you can't open it, you can't pre-empt it...

Circular Wait

- Process 0 needs something process 4 has
  - Process 4 needs something process 7 has
  - Process 7 needs something process 1 has
  - Process 1 needs something process 0 has – uh-oh...
- Described as "cycle in the resource graph"

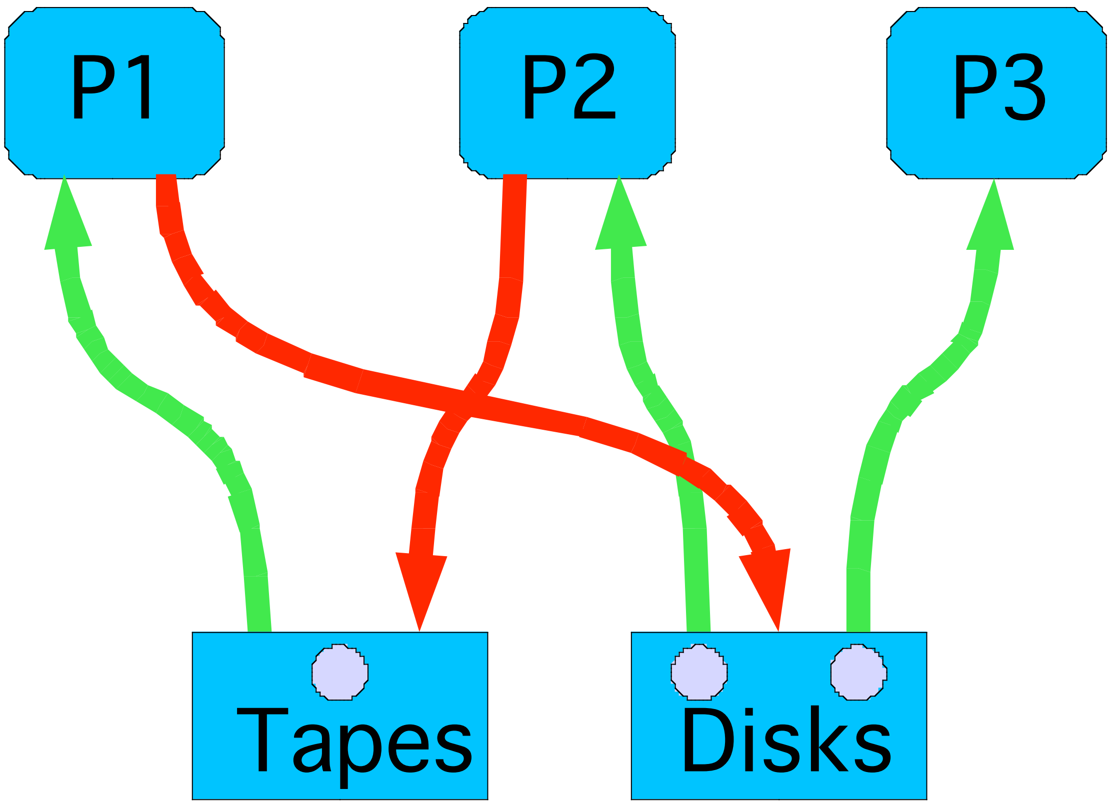
Cycle in Resource Graph



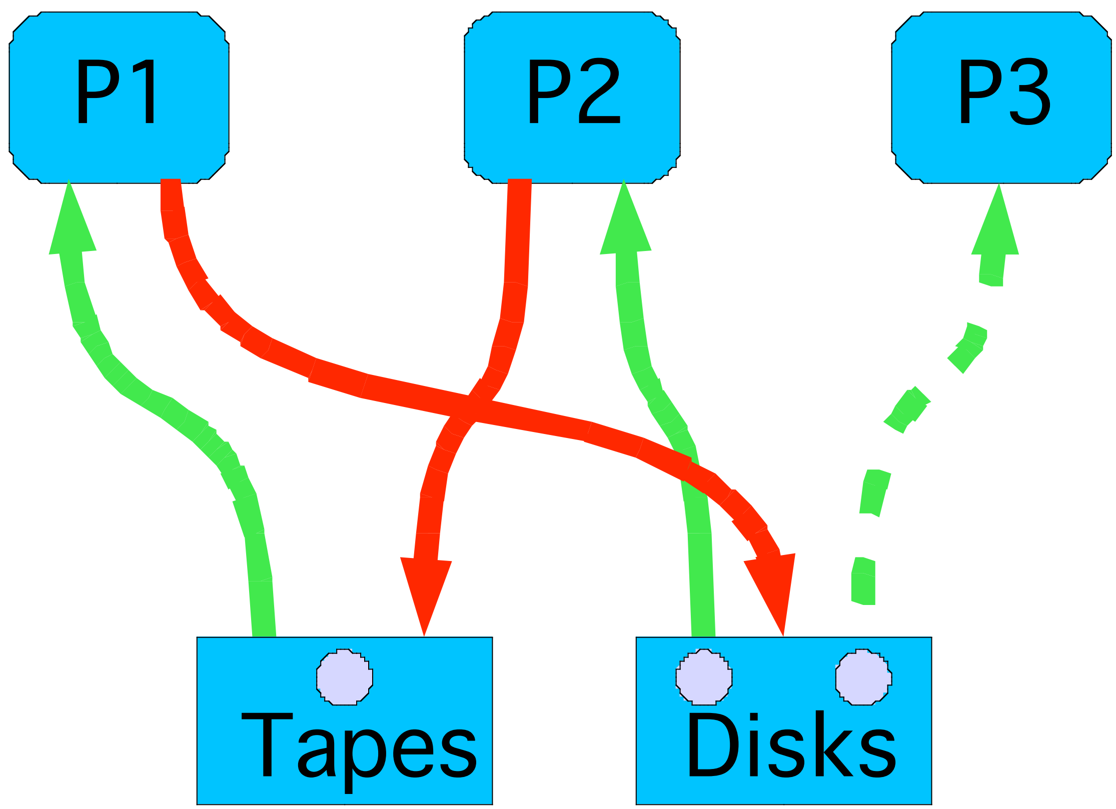
Deadlock Requirements

- **Mutual Exclusion**
- **Hold & Wait**
- **No Preemption**
- **Circular Wait**
- *Each deadlock* **requires** *all four*

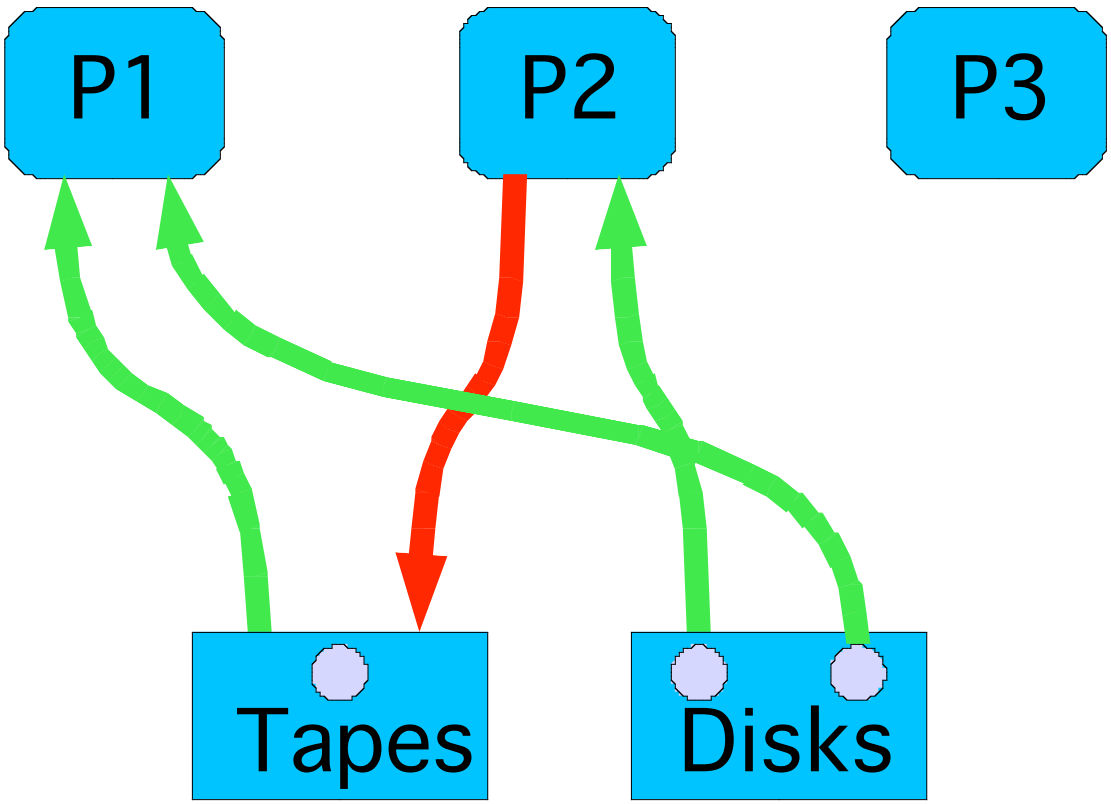
Multi-Instance Cycle



Multi-Instance Cycle *(With Rescuer!)*



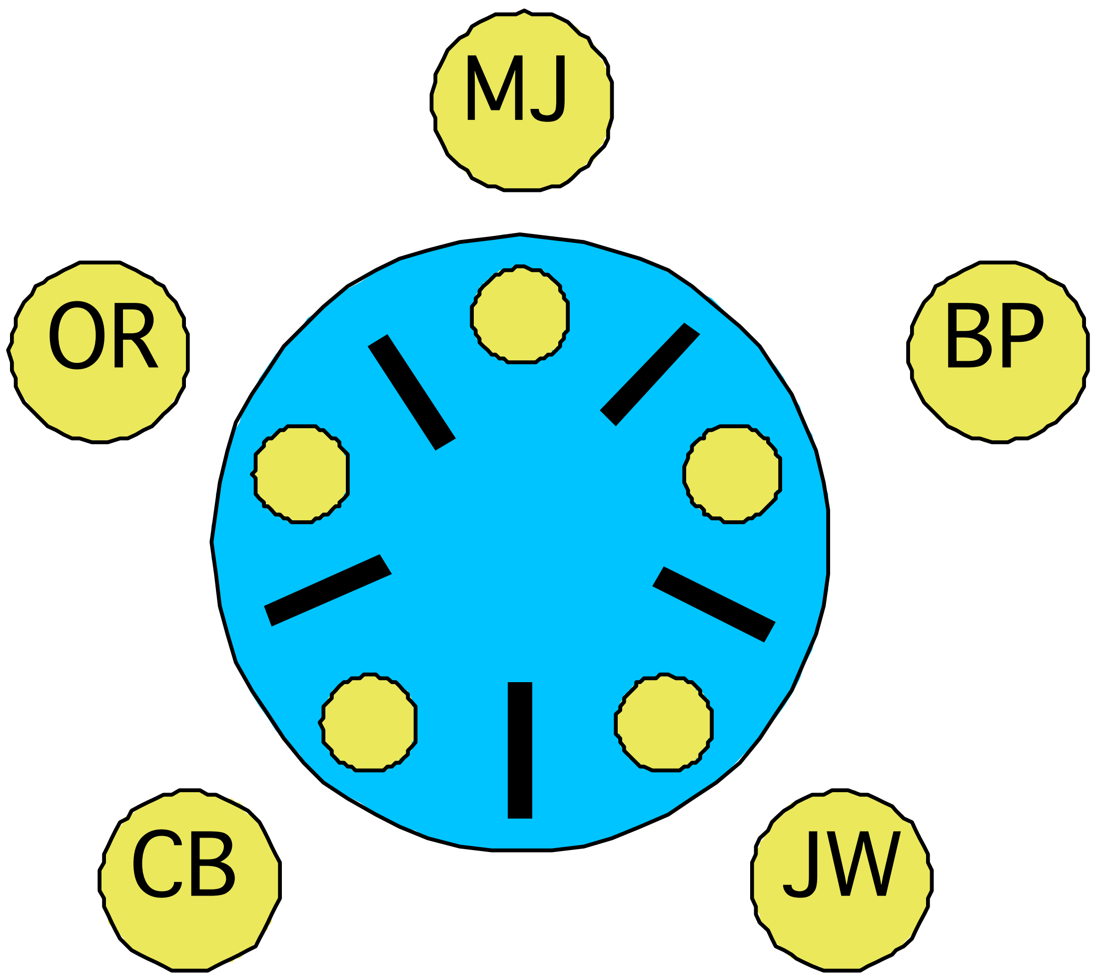
Cycle Broken



Dining Philosophers

- The scene
  - 410 staff at a Chinese restaurant
  - A little short on utensils

Dining Philosophers



Dining Philosophers

- Processes
  - 5, one per person
- Resources
  - 5 bowls (dedicated to a diner: no contention: ignore)
- 5 chopsticks
  - 1 between every adjacent pair of diners
- Contrived example?
  - Illustrates contention, starvation, deadlock

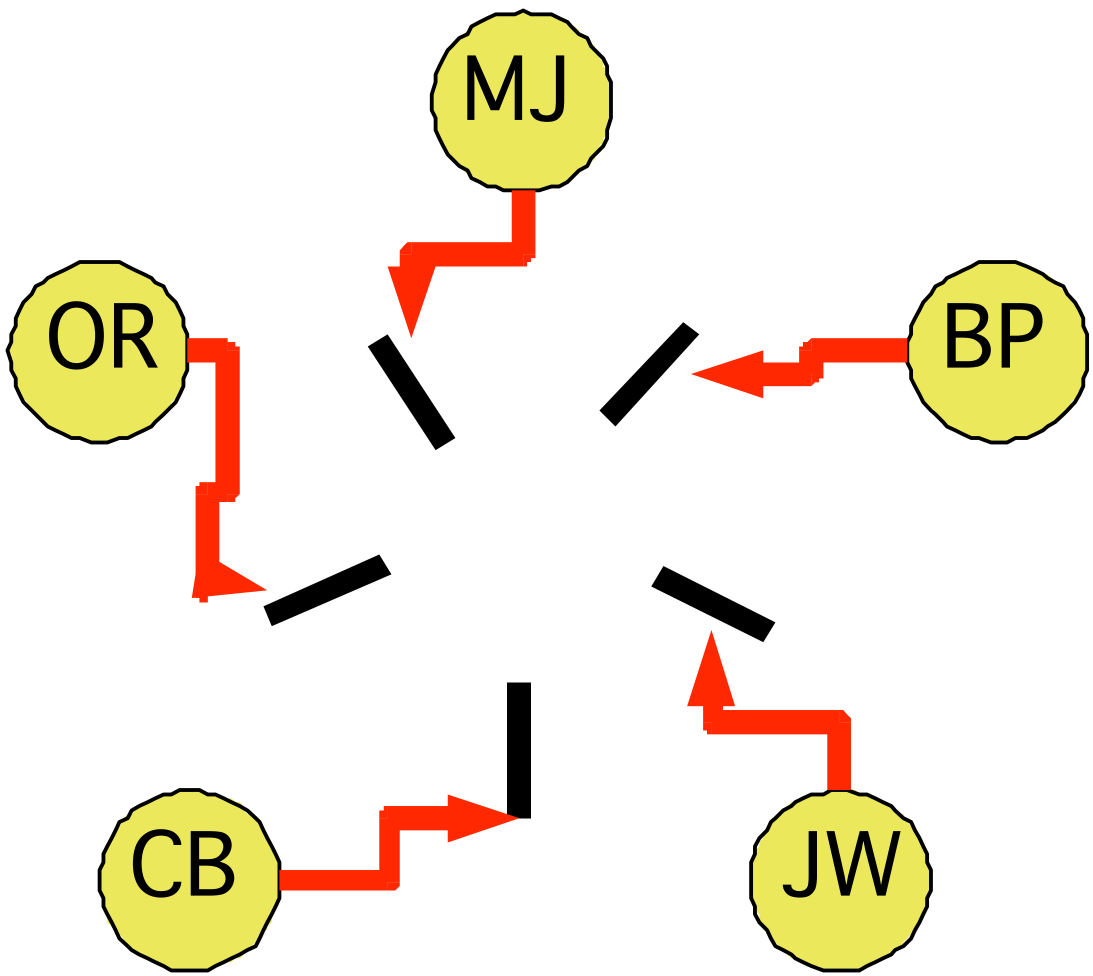
Dining Philosophers

- A simple rule for eating
  - Wait until the chopstick to your right is free; take it
  - Wait until the chopstick to your left is free; take it
  - Eat for a while
  - Put chopsticks back down

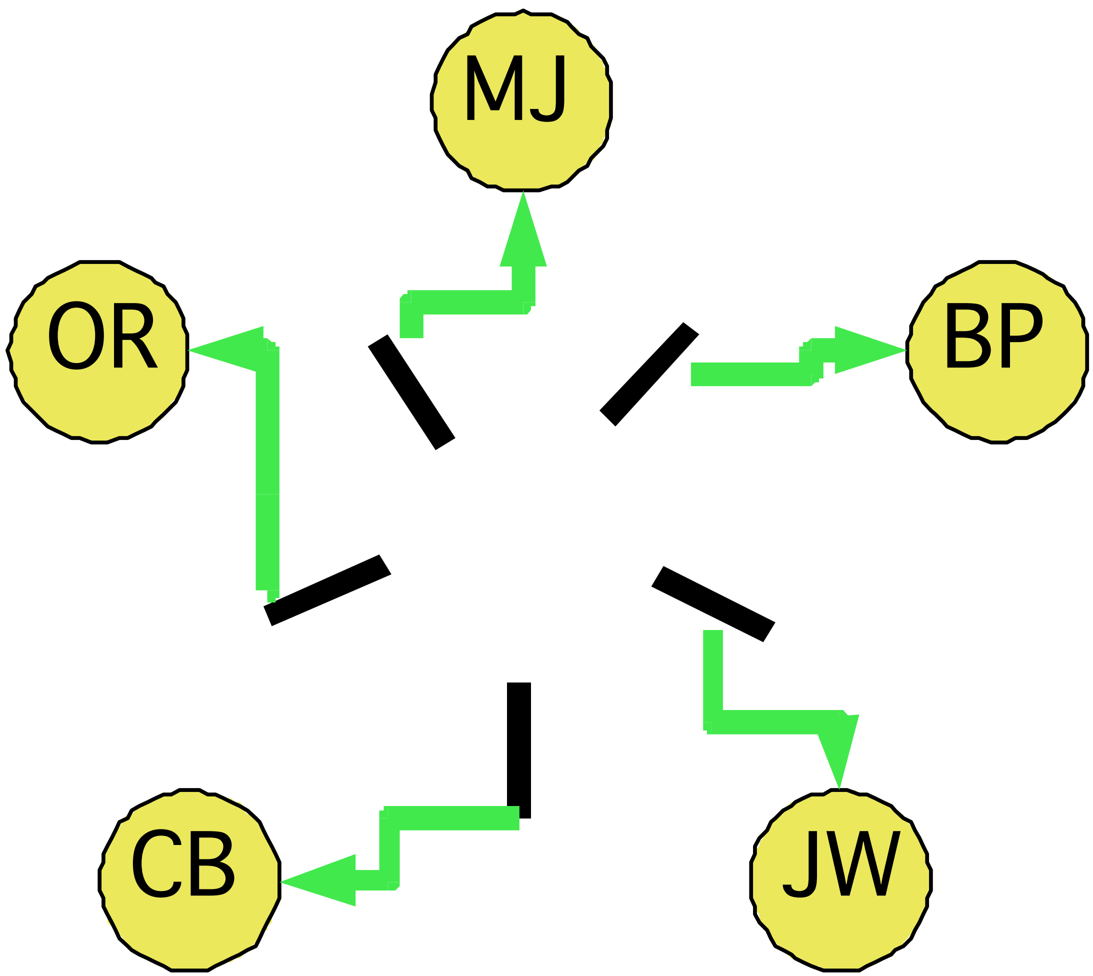
Dining Philosophers Deadlock

- Everybody reaches right...
  - ...at the same time?

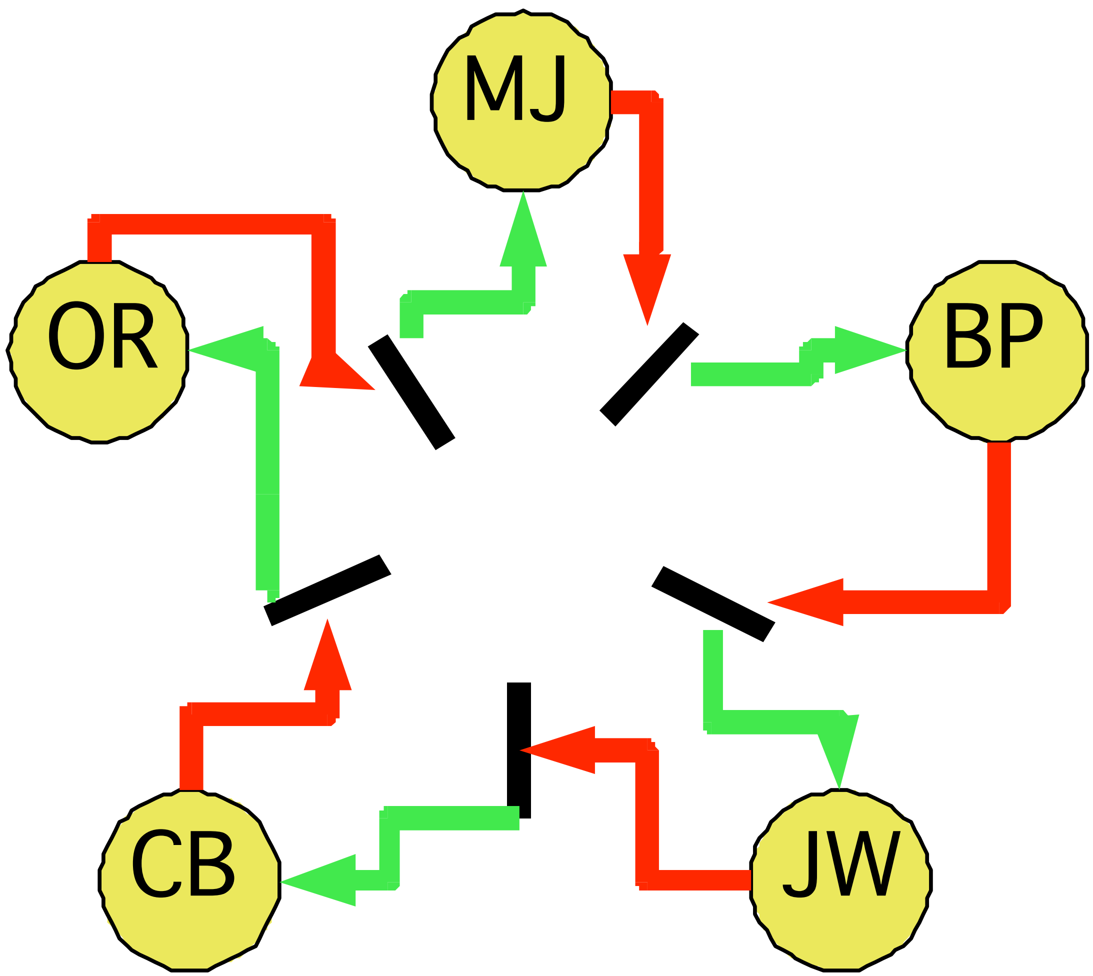
Reaching Right



Process graph



Deadlock!



Dining Philosophers – State

```
int stick[5] = { -1 }; /* owner */
condition avail[5]; /* newly avail. */
mutex table = { available };

/* Right-handed convention */
right = diner;                /* 3 ⇒ 3 */
left = (diner + 4) % 5; /* 3 ⇒ 7 ⇒ 2 */
```

start_eating(int diner)

```
mutex_lock(table);

while (stick[right] != -1)
  condition_wait(avail[right], table);
stick[right] = diner;

while (stick[left] != -1)
  condition_wait(avail[left], table);
stick[left] = diner;

mutex_unlock(table);
```

done_eating(int diner)

```
mutex_lock(table);

stick[left] = stick[right] = -1;
condition_signal(avail[right]);
condition_signal(avail[left]);

mutex_unlock(table);
```

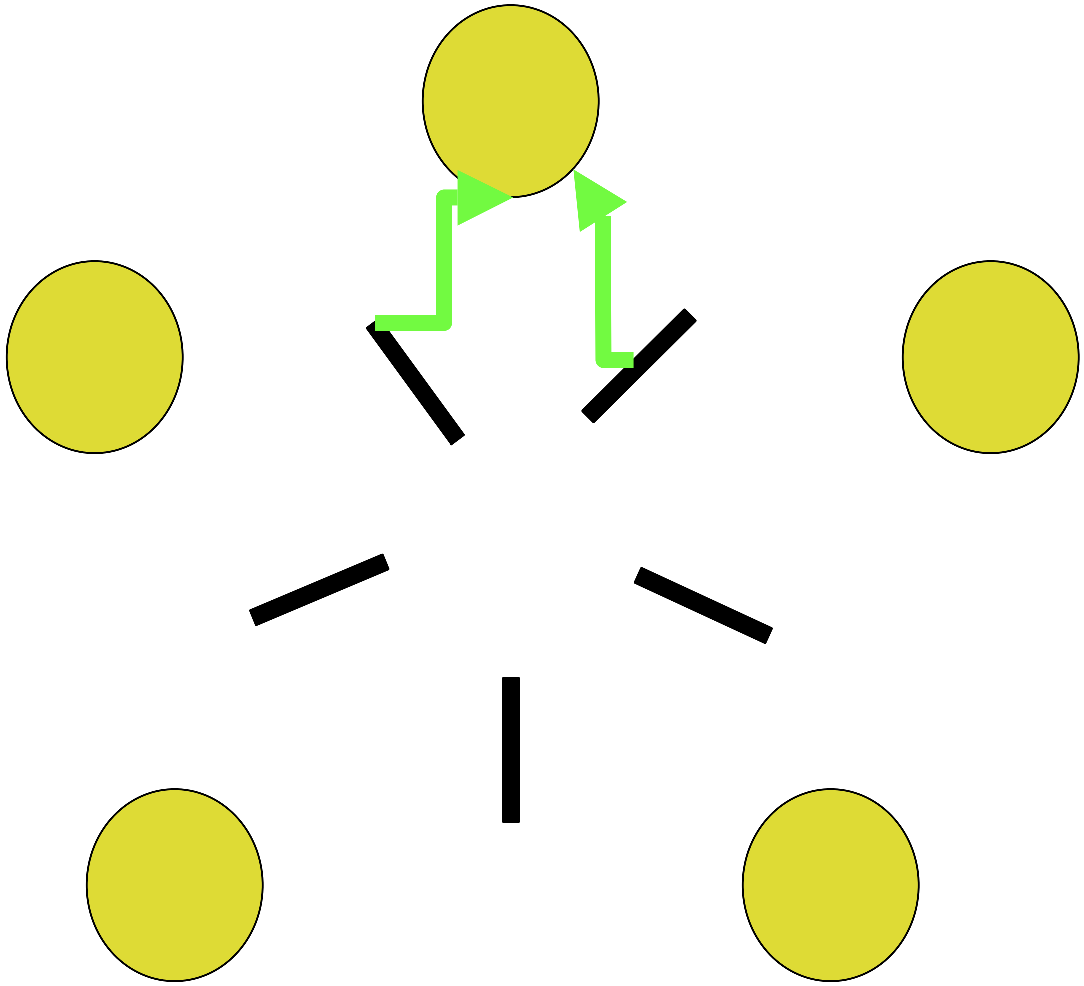
Can We Deadlock?

- At first glance the table mutex protects us
  - Can't have “everybody reaching right at same time”...
  - ...mutex means only one person can access table...
  - ...so allows only one reach at the same time, right?

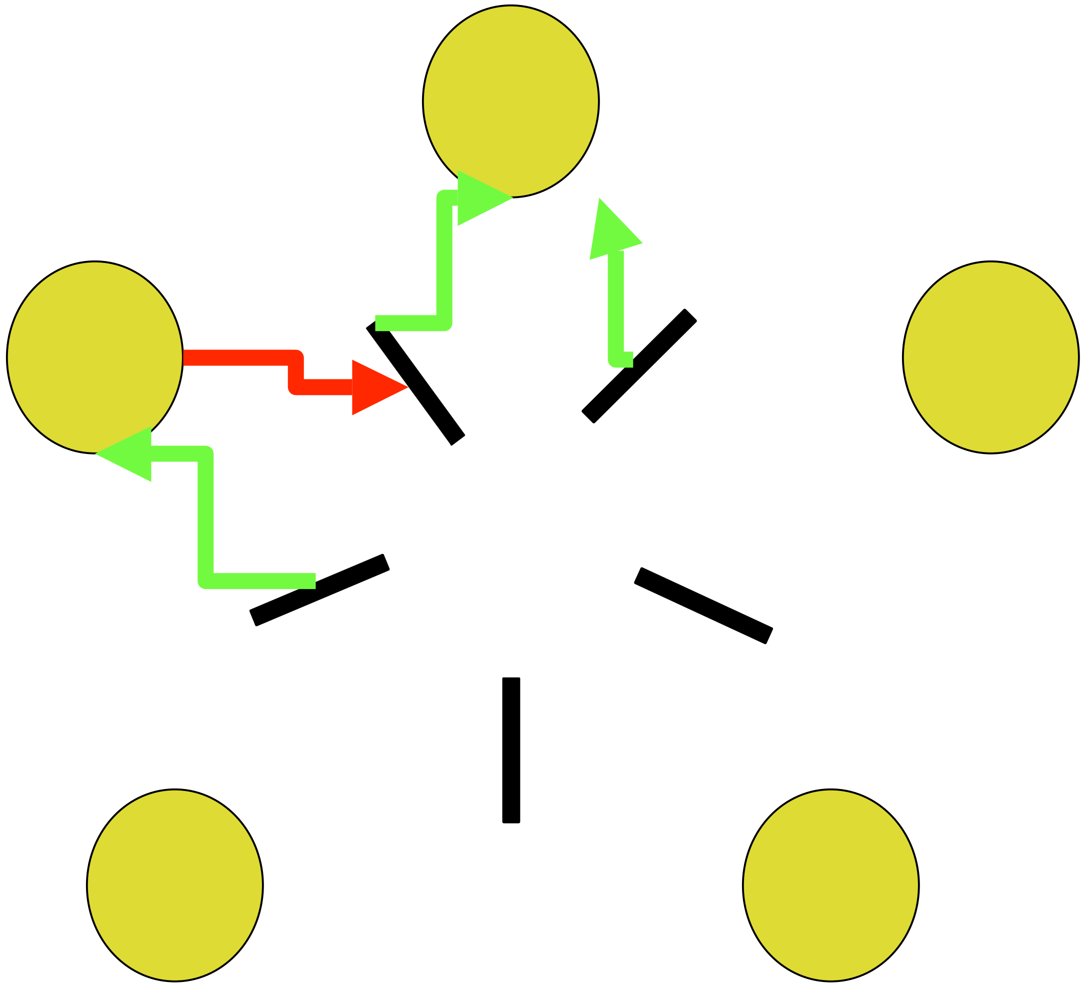
Can We Deadlock?

- At first glance the table mutex protects us
  - Can't have “everybody reaching right at same time”...
  - ...mutex means only one person can access table...
  - ...so allows only one reach at the same time, right?
- Maybe we can!
  - *condition_wait()* is a “reach”
  - Can everybody end up in condition_wait()?

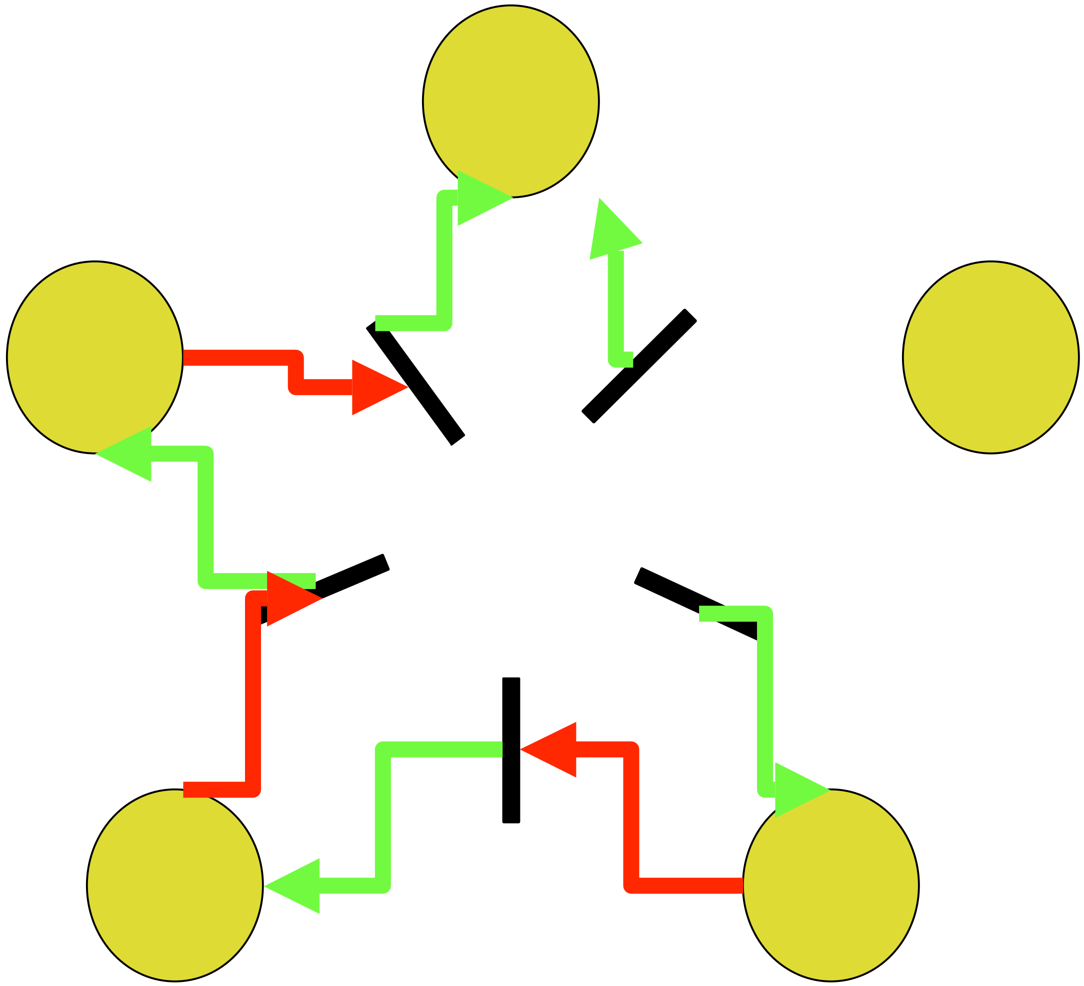
First diner gets both chopsticks



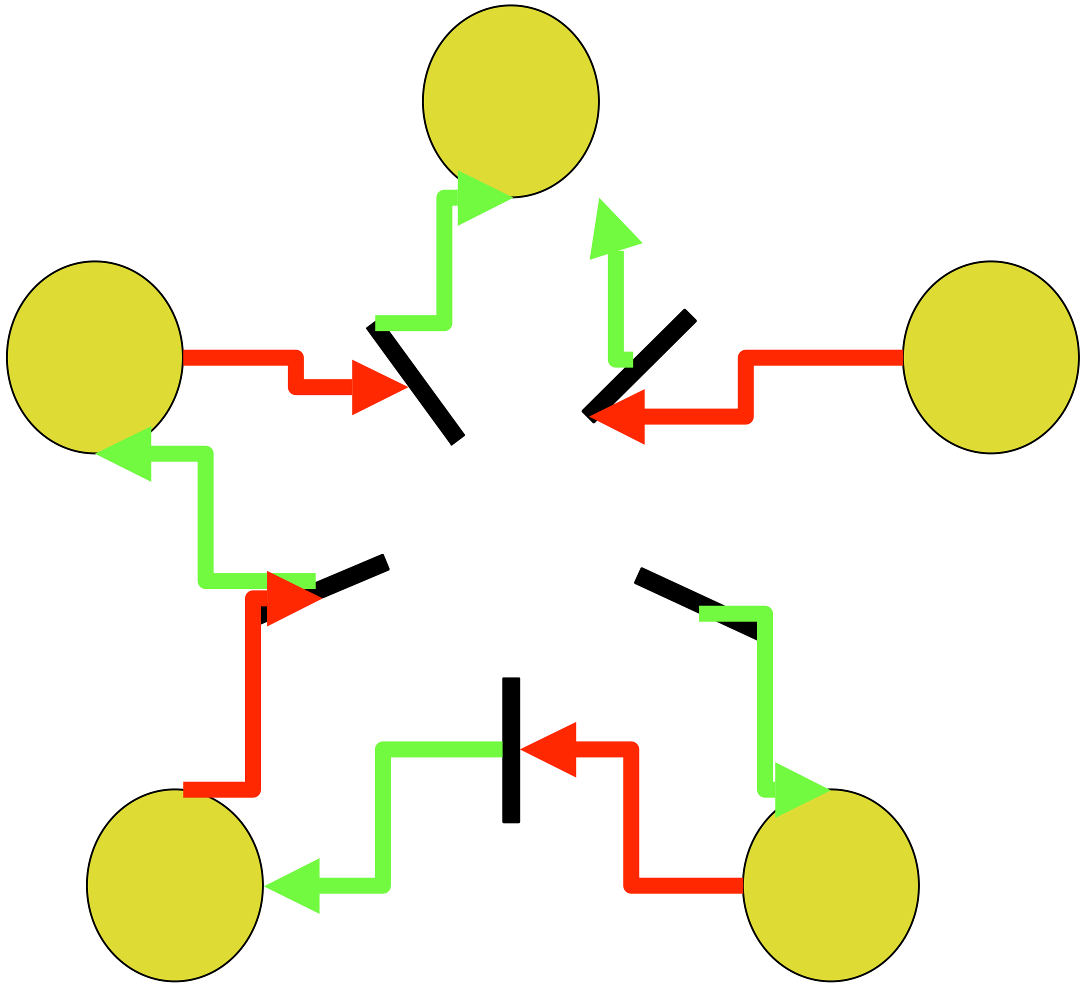
Next gets right, waits on left



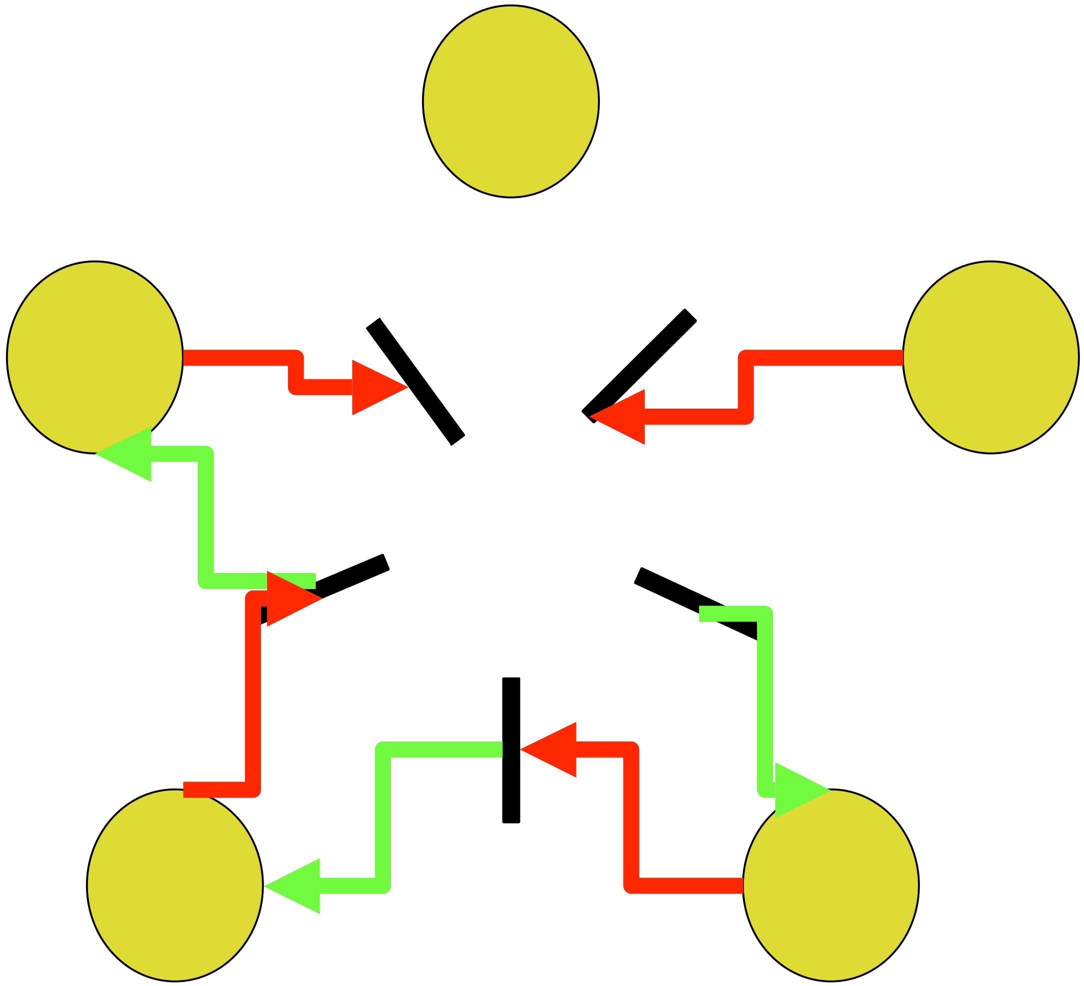
Next two get right, wait on left



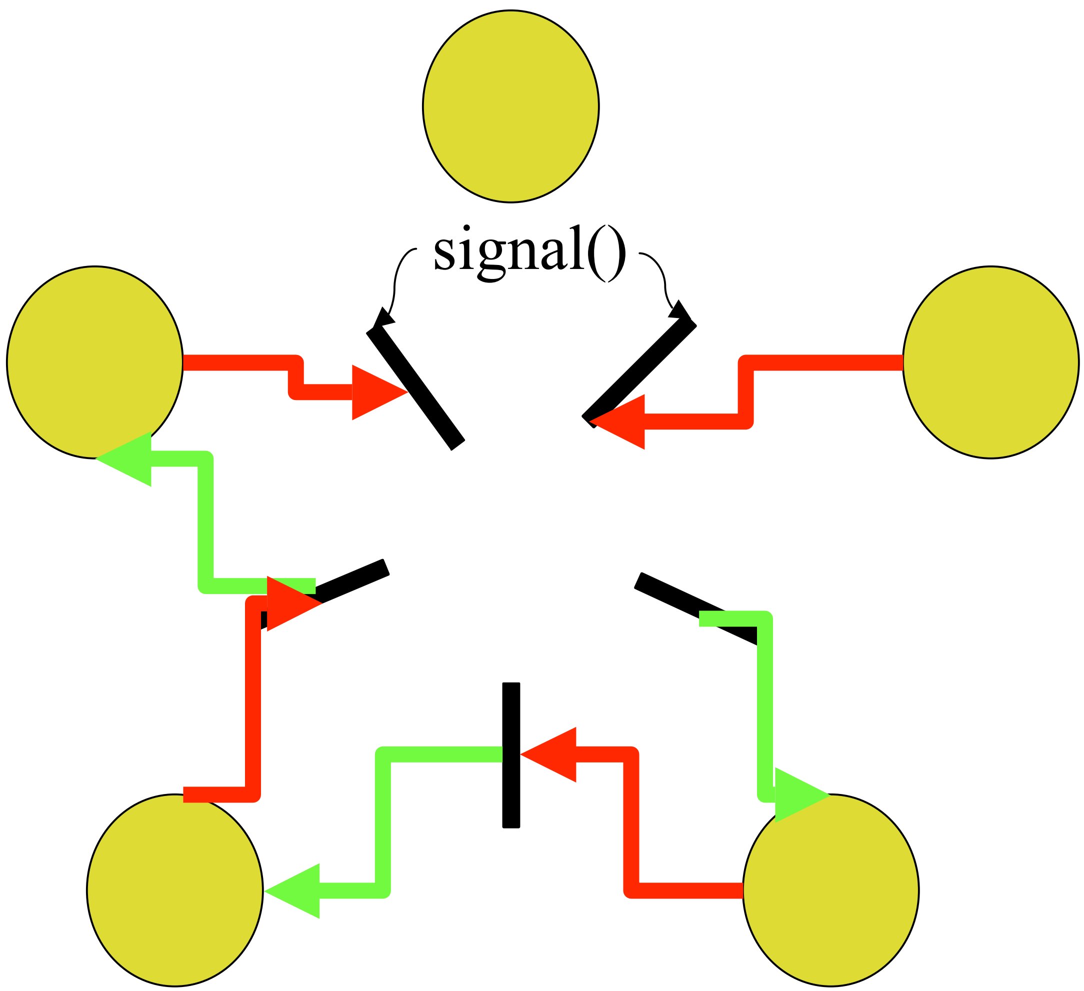
Last waits on right



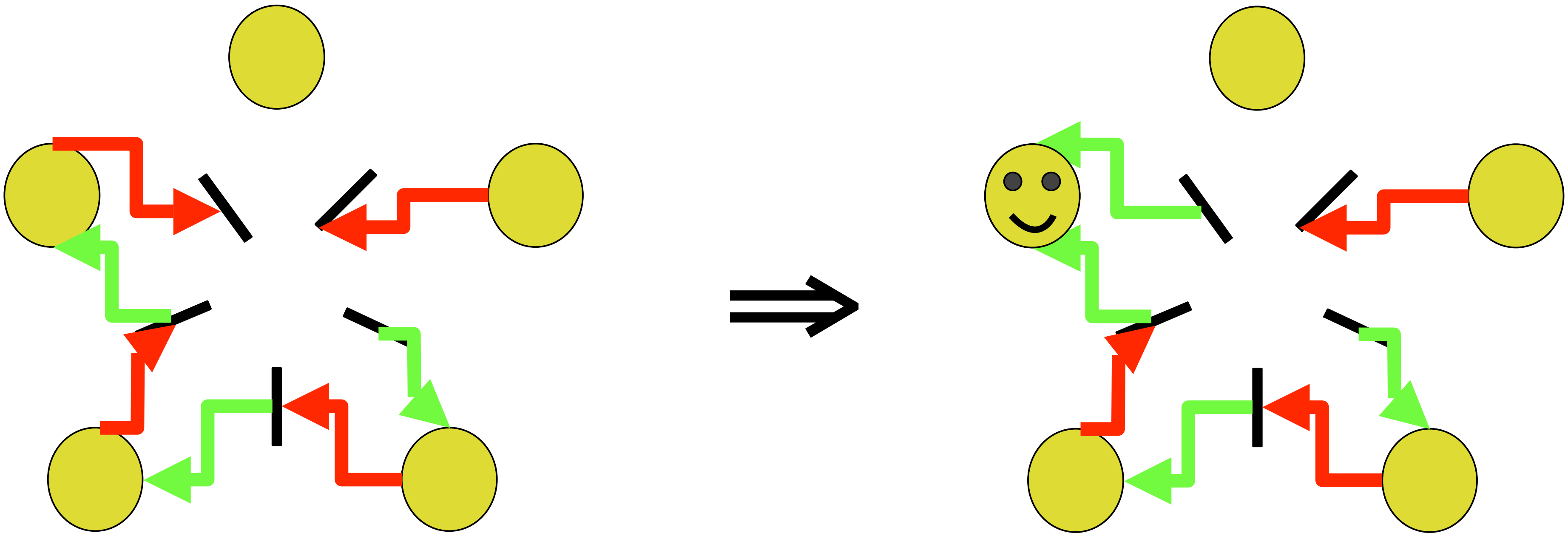
First diner stops eating - *briefly*



First diner stops eating - *briefly*

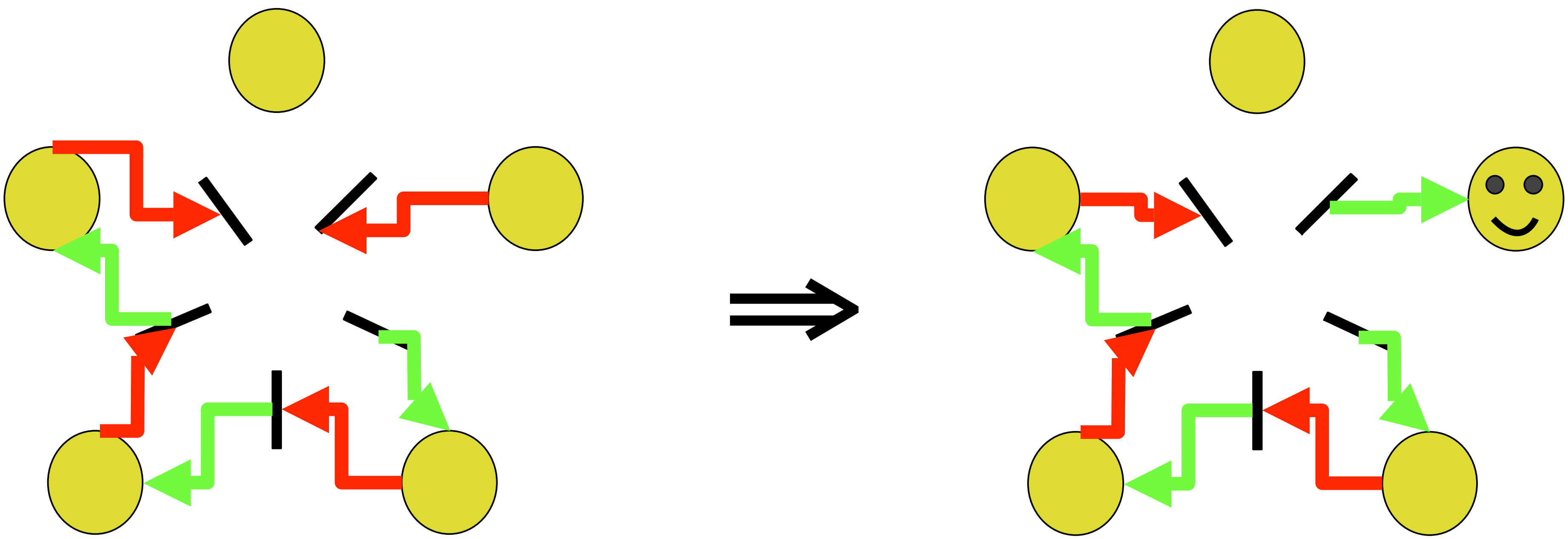


Next Step – *One* Possibility



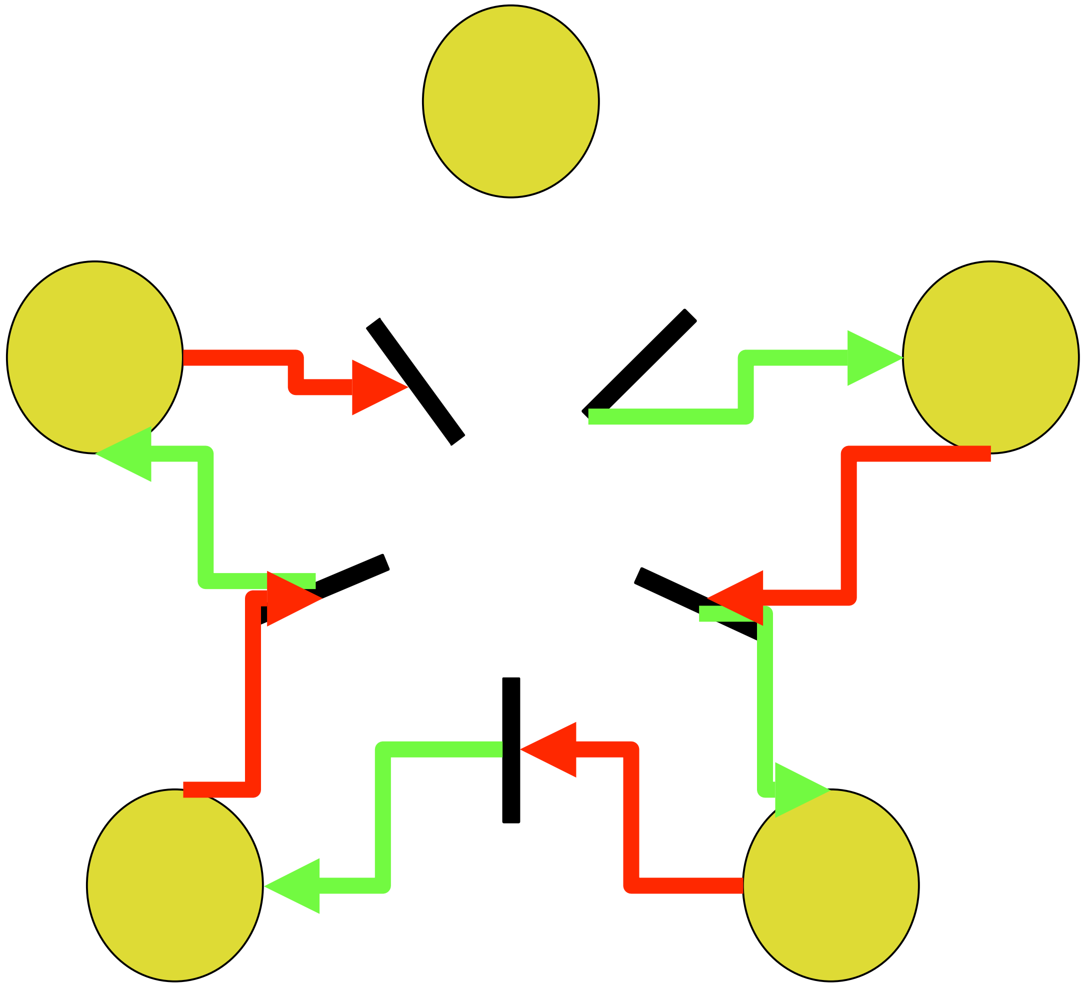
“Natural” –
longest-waiting diner progresses

Next Step – *Another* Possibility

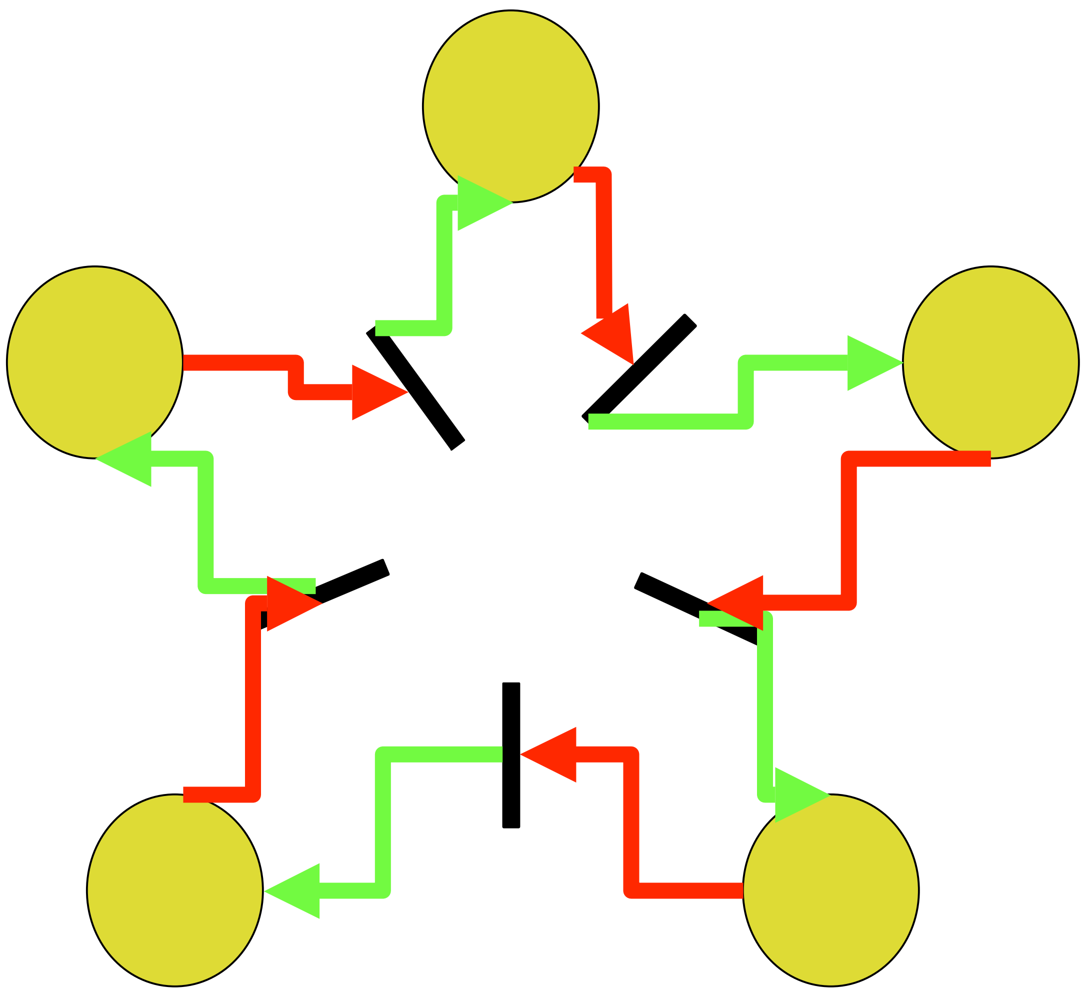


Or –
somebody else!

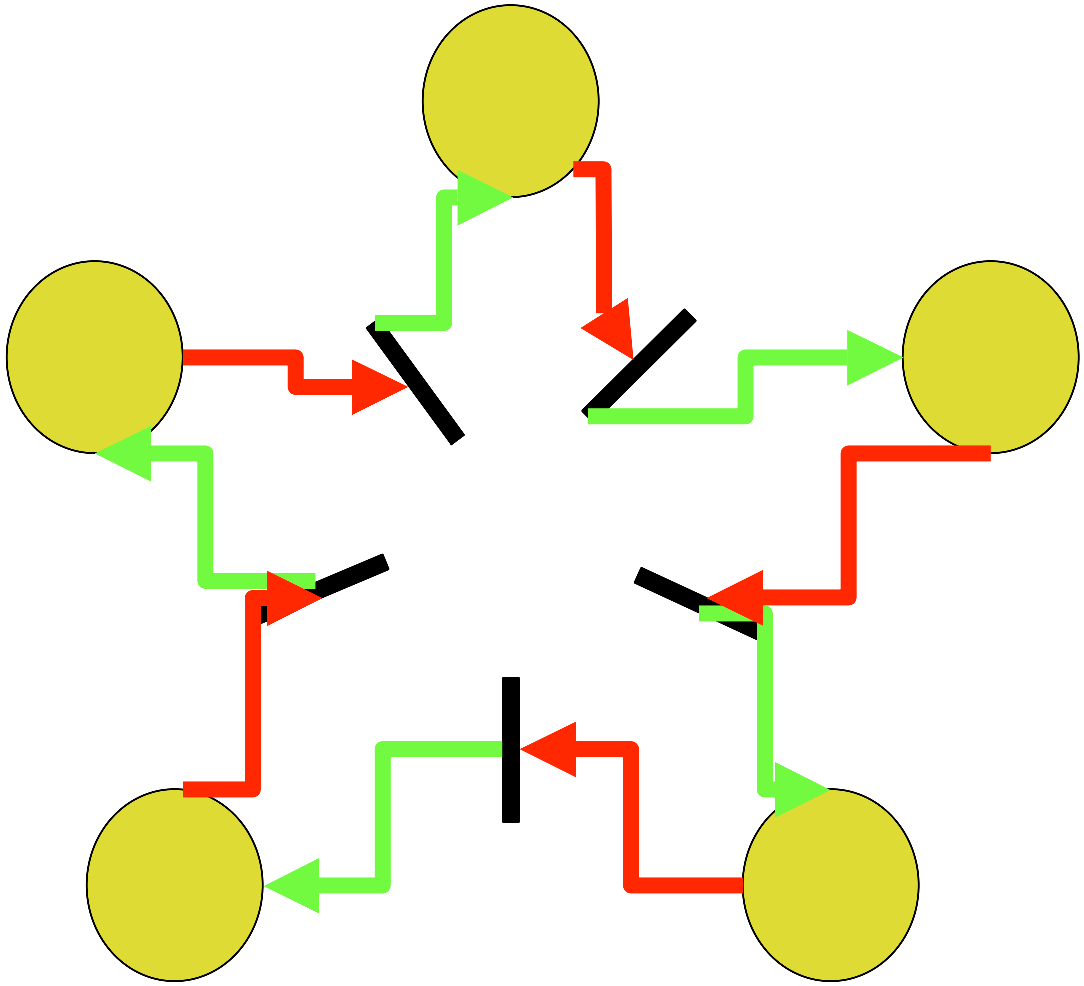
Last diner gets right, waits on left



First diner gets right, waits on left



Now things get boring



Deadlock - What to do?

- Prevention
- Avoidance
- Detection/Recovery
- Just reboot when it gets "too quiet"

1: Prevention

- Restrict behavior or resources
  - Find a way to violate one of the 4 conditions
    - To wit...?
- What we will talk about today
  - 4 conditions, 4 possible ways

2: Avoidance

- Processes *pre-declare* usage patterns
- Dynamically examine requests
  - Imagine what other processes could ask for
  - Keep system in “safe state”

3: Detection/Recovery

- **Maybe deadlock won't happen today...**
- **...Hmm, it seems quiet...**
- **...Oops, here is a cycle...**
- *Abort some process*
  - **Ouch!**

4: Reboot When It Gets “Too Quiet”

- **Which systems would be so simplistic?**

Four Ways to Forgiveness

- *Each deadlock* **requires** *all four*
  - **Mutual Exclusion**
  - **Hold & Wait**
  - **No Preemption**
  - **Circular Wait**
- **“Deadlock Prevention” - this is a technical term**
  - *Pass a law* **against one (pick one)**
  - **Deadlock happens only if somebody** *transgresses!*

Outlaw Mutual Exclusion?

- **Approach:** *ban* **single-user resources**
  - **Require all resources to "work in shared mode"**
- **Problem**
  - **Chopsticks???**
  - **Many resources don't work that way**

Outlaw Hold&Wait?

- **Acquire resources** *all-or-none*

```
start_eating(int diner)

mutex_lock(table);
while (1)
  if (stick[lt] == stick[rt] == -1)
    stick[lt] = stick[rt] = diner
    mutex_unlock(table)
    return;
  condition_wait(released, table);
```

Problems

- "Starvation"
  - Larger resource set makes grabbing everything harder
    - No guarantee a diner eats in bounded time
- Low utilization
  - Larger peak resource needs hurts whole system always
    - Must allocate 2 chopsticks (and waiter!)
    - Nobody else can use waiter while you eat

Outlaw Non-preemption?

- **Steal resources from sleeping processes!**

```
start_eating(int diner)
right = diner;   rright =
 (diner+1)%5;
mutex_lock(table);
while (1)
  if (stick[right] == -1)
    stick[right] = diner
  else if (stick[rright] != rright)
    /* right person can't be eating:
 take! */
    stick[right] = diner;
...same for left...wait() if must...
mutex unlock(table);
```

Problem

- Some resources cannot be cleanly preempted
  - CD burner

Outlaw Circular Wait?

- Impose *total order* on all resources
- Require acquisition in *strictly increasing order*
  - Static order may work: allocate memory, then files
  - Dynamic – may need to “start over” sometimes
    - Traversing a graph
      - lock(4), visit(4) /* 4 has an edge to 13 */
      - lock(13), visit(13) /* 13 has an edge to 0 */
      - lock(0)?
        - Nope!
        - unlock(4), unlock(13)
        - lock(0), lock(4), lock(13), ...

Assigning Diners a Total Order

- **Lock order: 4, 3, 2, 1, 0 ≡ right chopstick, then left**
  - **Diner 4 ⇒ lock(4); lock(3);**
  - **Diner 3 ⇒ lock(3); lock(2);**

Assigning Diners a Total Order

- **Lock order: 4, 3, 2, 1, 0 ≡ right chopstick, then left**
  - **Diner 4 ⇒ lock(4); lock(3);**
  - **Diner 3 ⇒ lock(3); lock(2);**
  - **Diner 0 ⇒ lock(0); lock(4); /* invalid lock order! */**
    - **Requires special-case locking code to get order right**

```
if diner == 0
  right = (diner + 4) % 5;
  left = diner;
else
  right = diner;
  left = (diner + 4) % 5;
...
```

Problem

- **May not be possible to force allocation order**
  - **Some trains go east, some go west**

Deadlock Prevention problems

- Typical resources *require* mutual exclusion
- All-at-once allocation can be *painful*
  - Hurts efficiency
  - May starve
  - Resource needs may be unpredictable
- Preemption may be *impossible*
  - Or may lead to starvation
- Ordering restrictions may be *impractical*

Deadlock Prevention

- **Pass a law against one of the four ingredients**
  - **Great if you can find a tolerable approach**
- *Very* **tempting to just let processes try their luck**

Deadlock is not...

- **...a simple synchronization bug**
  - **Deadlock remains even when those are cleaned up**
  - **Deadlock is a resource usage design problem**
- **...the same as starvation**
  - **Deadlocked processes don't ever get resources**
  - **Starved processes don't ever get resources**
  - **Deadlock is a “progress” problem; starvation is a “bounded waiting” problem**
- **....that “after-you, sir” dance in the corridor**
  - **That's “livelock” – continuous changes of state without forward progress**

Next Time

- Deadlock Avoidance
- Deadlock Recovery