Deadlock (1)

David A. Eckhardt
School of Computer Science
Carnegie Mellon University

de0u@andrew.cmu.edu

Status Rendezvous

Partner selection for Project 2

- Largely complete
  - If you are unpartnered, you got mail last night

Project 2

- Out: today
- In: Wednesday, February 19

Outline

Textbook

- Chapter 8

Deadlock

- What it is
- How to get one
- One approach to *not* getting one as a gift

Definition of Deadlock

Deadlock

- Set of N processes
- Each waiting for an event
- caused by another process in the set

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 point finger
  - but the problem is there anyway

Deadlock Requirements

Mutual exclusion

- resources must be “owned”, not simultaneously shared

Hold & Wait

- a process can hold one resource while waiting to get another

No preemption

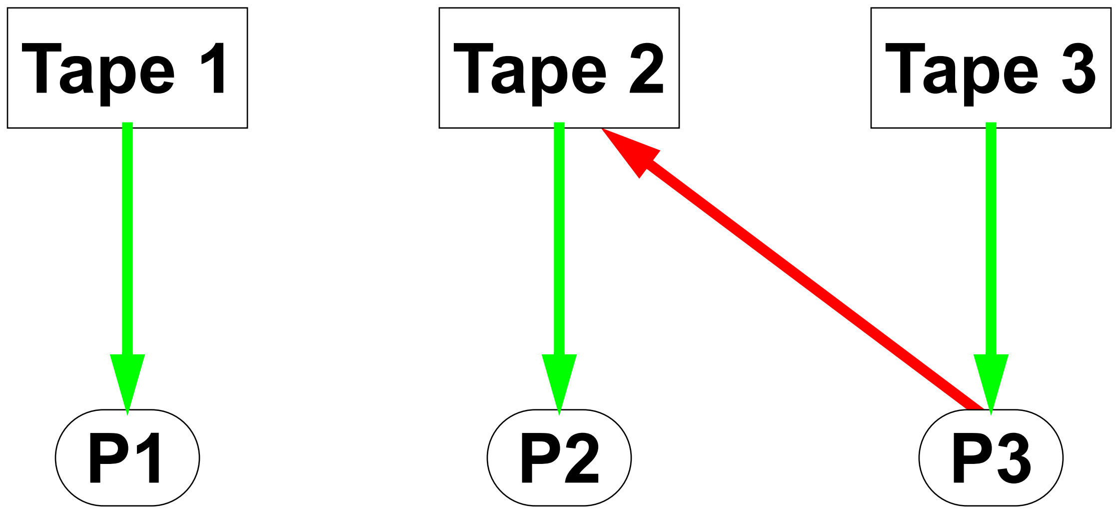
- no way to force a process to yield a resource it has

Circular Wait

- process 0 needs something process 4 has
- process 4 needs something process N has
- process N needs something process M has
- process M needs something process 0 has

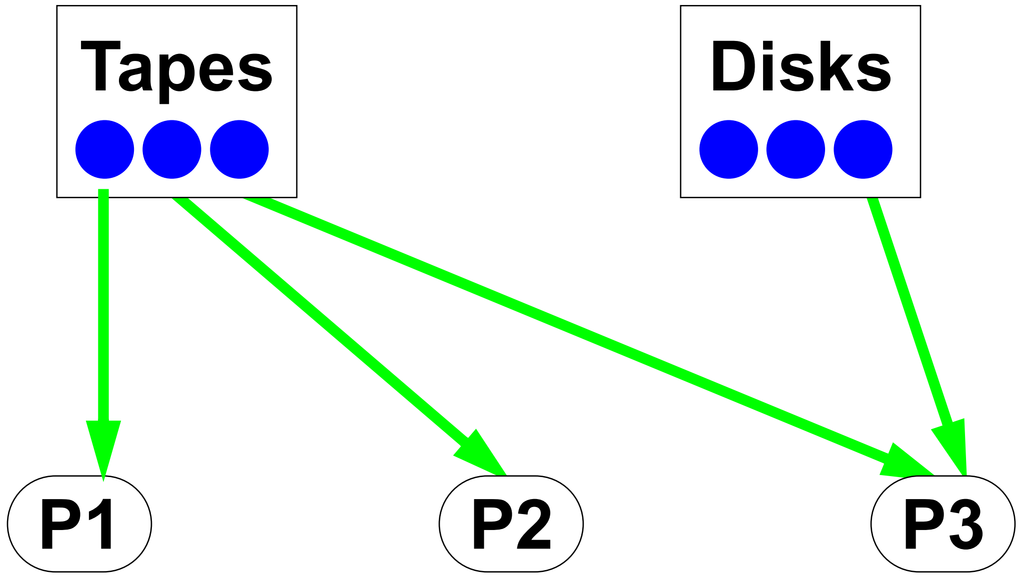
Deadlock requires *all four*

Process/Resource graph

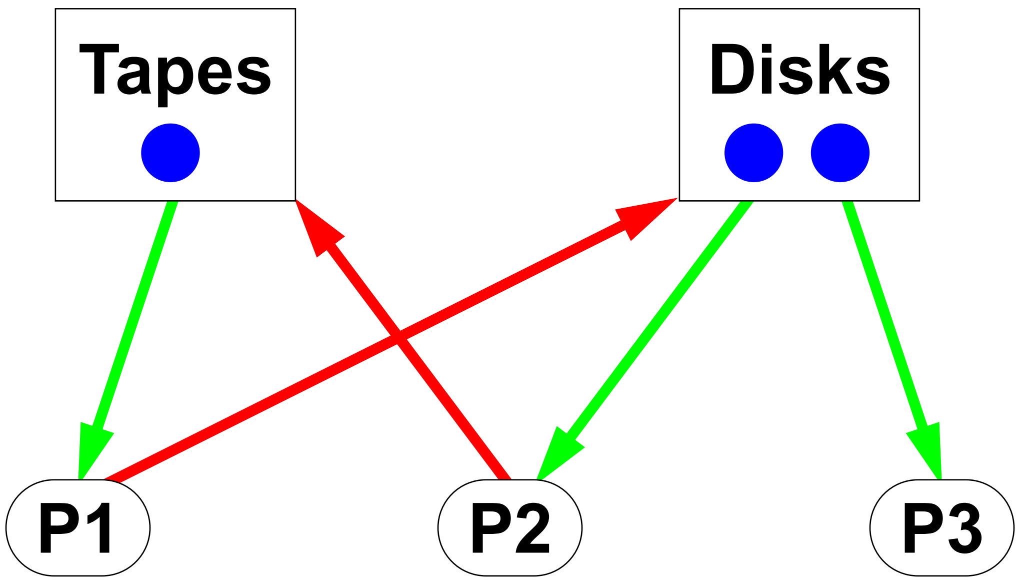


Allocation: arrow from resource to process (green)
Request: arrow from process to resource (red)

Interchangeable resources



Some Cycles are Ok

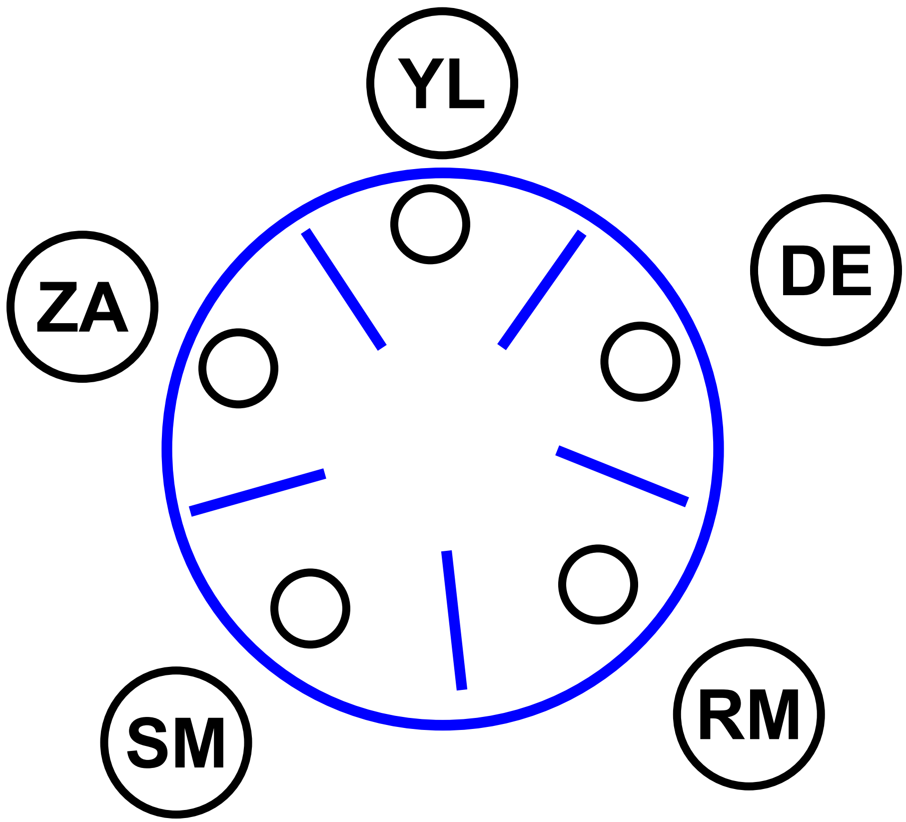


Only *rescuer-free* cycles are deadlocks

Dining Philosophers

The scene

- 412 staff at a Chinese restaurant
- a little short on cutlery



Dining Philosophers

Processes

- 5, one per person

Resources

- 5 bowls
  - each dedicated to a diner (ignore)
- 5 chopsticks
  - 1 between every adjacent pair of diners

Who cares?

- illustrates contention, starvation, deadlock

Dining Philosophers

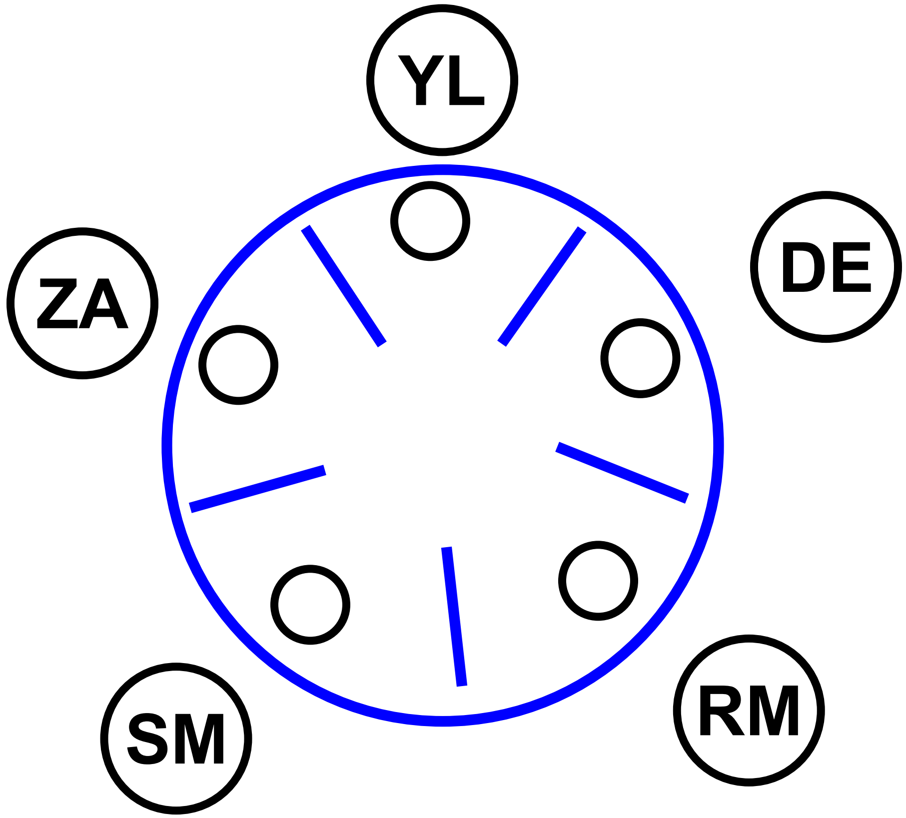
```
int stick[5] = { -1 };
condition want[5]; /* stick */
mutex table = { true };

start_eating(int diner)
 right = (diner + 1) % 5;
 left = (diner + 4) % 5;
 mutex_lock(table);
 while (stick[right] != -1)
  condition_wait(want[right], table);
 stick[right] = diner;
 while (stick[left] != -1)
  condition_wait(want[left], table);
 stick[left] = diner;
 mutex_unlock(table);
```

Dining Philosophers

```
done_eating(int diner)
 right = (diner + 1) % 5;
 left = (diner + 4) % 5;
 mutex_lock(table);
  stick[diner] = stick[diner] = -1;
  condition_signal(want[right]);
  condition_signal(want[left]);
 mutex_unlock(table);
```

Dining Philosophers Deadlock



What if everybody reaches right *at the same time*?

Deadlock - What to do?

Prevention

- restrict behavior or resources
- violate one of the 4 conditions

Avoidance

- dynamically examine requests
- keep system in "safe state"

Detection/Recovery

- maybe deadlock won't happen today
- gee, it seems quiet
- oops, here is a cycle
- abort some processes

Just reboot when it gets "too quiet"

Prevention - 1

Violate mutual exclusion

- Don't *have* single-user resources

Problem

- Not going to work out for chopsticks

Prevention - 2

Violate Hold&Wait

- Acquire resources all-or-none

```
start_eating(int diner)
 right = (diner + 1) % 5;
 left = (diner + 4) % 5;
 done = false;
 mutex_lock(table);
 while (1)
  if (stick[left] == -1 && stick[right] == -1)
   stick[left] = stick[right] = diner
   mutex_unlock(table)
   return
  condition_wait(somebody_finished, table);
```

Violating Hold&Wait

Problem - starvation

- Larger resource set makes grabbing harder
- No guarantee a diner eats in bounded time

Problem - low utilization

- Must allocate 2 chopsticks *and* waiter
- Nobody else can use waiter while you eat

Problem - not everybody knows in advance

Prevention - 3

Violate non-preemption

- steal resources from sleeping processes

```
start_eating(int diner)
 right = (diner + 1) % 5;
 rright = (diner + 2) % 5;
 left = (diner + 4) % 5;
 lleft = (diner + 3) % 5;
 mutex_lock(table);
 while (1)
  if (stick[right] == -1)
  stick[right] = diner
  else if (stick[rright] != rright)
  /* right cannot be eating */
  /* take right's stick */
  stick[right] = diner
  ...same for left...
 mutex_unlock(table);
```

Violating Non-preemption

Problem

- Some resources cannot be cleanly preempted

Prevention - 4

Avoid circular wait

- impose total order on all resources
- require acquisition in strictly increasing order
  - static: allocate memory, then files
  - dynamic: ooops, need resource 0; dump all, start over

Assigning a Total Order

```
start_eating(int diner)
if diner == 4
 right = (diner + 4) % 5;
 left = (diner + 1) % 5;
else
 right = (diner + 1) % 5;
 left = (diner + 4) % 5;
 mutex_lock(table);
 while (stick[right] != -1)
  condition_wait(want[right], table);
 stick[right] = diner;
 while (stick[left] != -1)
  condition_wait(want[left], table);
 stick[left] = diner;
 mutex_unlock(table);
```

Assigning a Total Order

Problem

- may not be possible to force allocation order
  - some trains go east, some go west

Deadlock Prevention problems

Typical resources require mutual exclusion

Allocation restrictions can be painful

- all-at-once
  - hurts efficiency
  - may starve
- resource needs may be unpredictable
- preemption may be impossible
  - or may lead to starvation
- ordering restrictions may not be feasible

Deadlock prevention summary

Great if you can find a tolerable approach

Awfully tempting to just let processes try their luck