Concurrency Problems (Bugs)

ECE469, Feb 2

Yiying Zhang

Reading assignment

- Dinosaur chapter 7
- Comet Ch 32 (Ch 33 for last lecture)
- Homework from this lecture (ungraded)
- Quiz 1 next Tue

[lec6] Producer & Consumer – Is the order of waits important?

- Producer

wait(empties)
wait(mutex)
get empty buffer from pool of empties
signal(mutex)

produce data in buffer

wait(mutex)
add full buffer to pool of fulls
signal(mutex)
signal(fulls)

- Consumer

wait(fulls)
wait(mutex)
get full buffer from pool of fulls
signal(mutex)

consume data in buffer

wait(mutex)
add empty buffer to pool of empties
signal(mutex)
signal(empties)

empties = 0; fulls = N

[lec6] Producer & Consumer – Is the order of waits important?

- Producer

  wait(mutex)
  wait(empties)
  get empty buffer from pool of empties
  signal(mutex)

  produce data in buffer

  wait(mutex)
  add full buffer to pool of fulls
  signal(mutex)
  signal(fulls)

- Consumer

  wait(fulls)
  wait(mutex)
  get full buffer from pool of fulls
  signal(mutex)

  consume data in buffer

  wait(mutex)
  add empty buffer to pool of empties
  signal(mutex)
  signal(empties)

empties = 0; fulls = N

[lec7] Dining Philosopher’s Problem

- Dijkstra 1971

- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

[lec7] What can go wrong?

- Primarily, we worry about:
  - Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  - Deadlock: A policy that leaves all the philosophers “stuck”, so that nobody can do anything at all
  - Livelock: A policy that makes them all do something endlessly without ever eating!

[lec7] A flawed conceptual solution

```
void getforks() {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
}

void putforks() {
        sem_post(forks[left(p)]);
        sem_post(forks[right(p)]);
}
```

Oops! Subject to deadlock if they all pick up their “right” fork simultaneously!

[lec7] Dijstra's Solution

```
void getforks() {
        if (p == 4) {
                sem_wait(forks[right(p)]);
                sem_wait(forks[left(p)]);
        } else {
                sem_wait(forks[left(p)]);
                sem_wait(forks[right(p)]);
        }
}
```

[lec7] Solutions are less interesting than the problem itself!

- In fact the problem statement is why people like to talk about this problem!
- Rather than solving Dining Philosophers, we should use it to understand properties of solutions that work and of solutions that can fail!

Deadlock Example

Deadlocks

- Definition: in a multiprogramming environment, a process is waiting forever for a resource held by another waiting process

- Topics:
  - Conditions for deadlocks
  - Strategies for handling deadlocks

System Model

- Resources
  - Resource types R_1, R_2, . . ., R_m
    - *CPU cycles, memory space, I/O devices, mutex*
  - Each resource type R_i has W_i instances
  - *Preemptable:* can be taken away by scheduler, e.g. CPU
  - *Non-preemptable:* cannot be taken away, to be released voluntarily, e.g., mutex, disk, files, ...

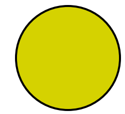
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Resource-Allocation Graph

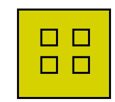
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

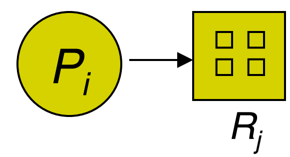
- Process



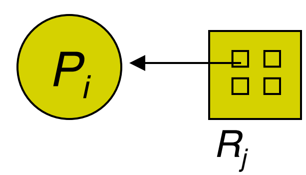
- Resource type with 4 instances



- P_i requests instance of R_j

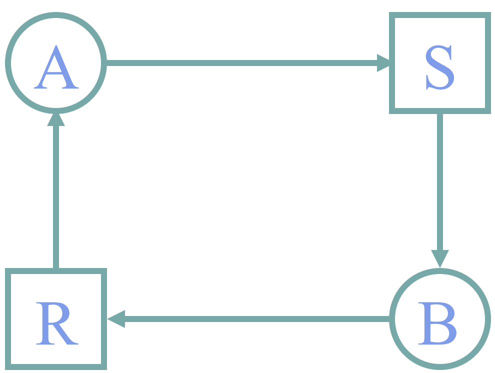


- P_i is holding an instance of R_j

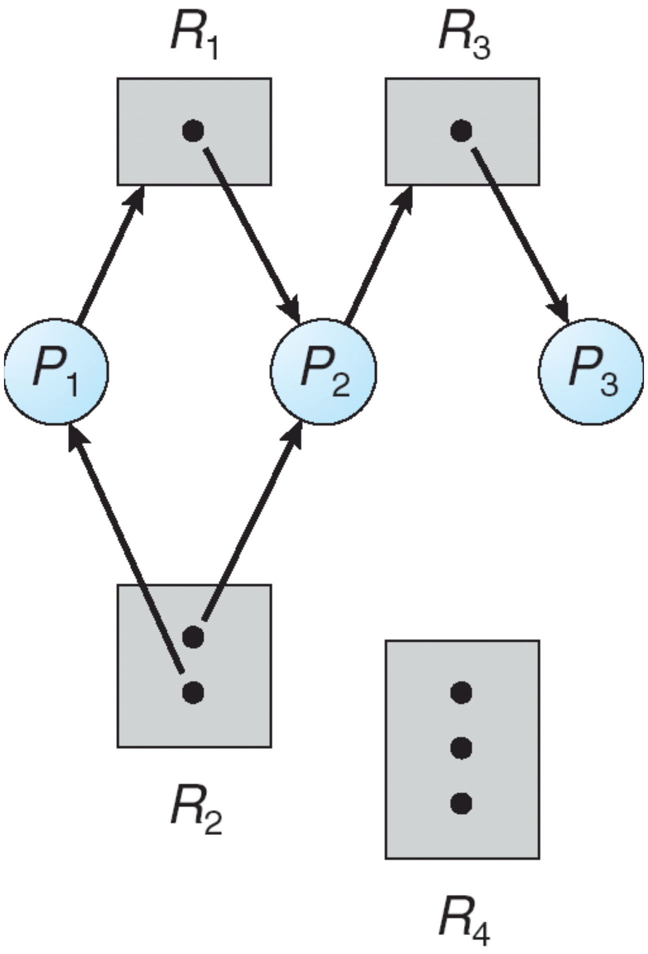


Example of a Resource Allocation Graph – one instance per type

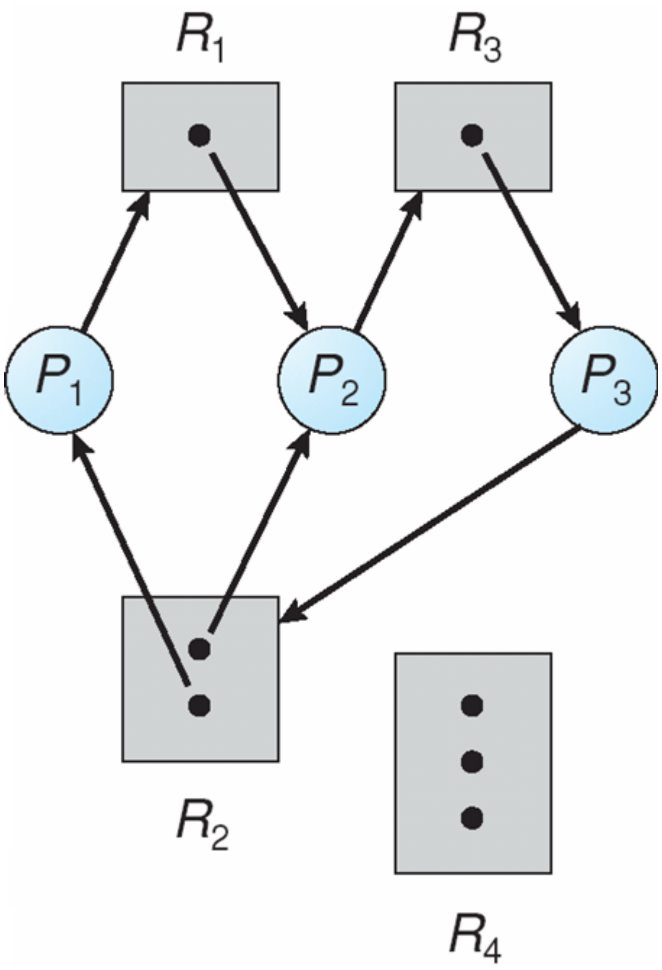
- What happens if there is a cycle in the resource allocation graph?



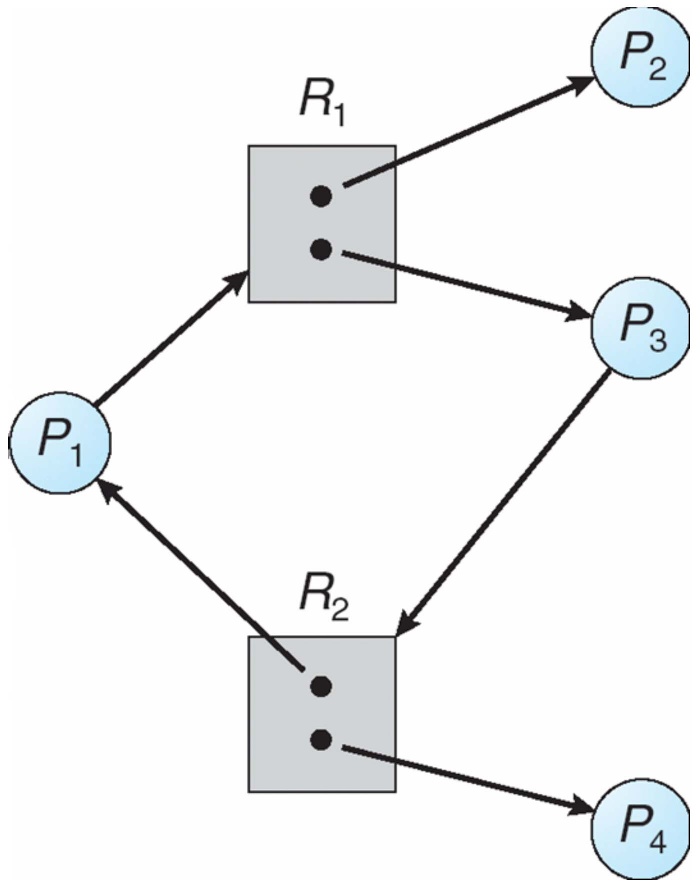
Example of a Resource Allocation Graph – multiple instances / type



Resource Allocation Graph with a cycle – is there a deadlock?



Resource Allocation Graph with a cycle – is there a deadlock?



Basic Facts

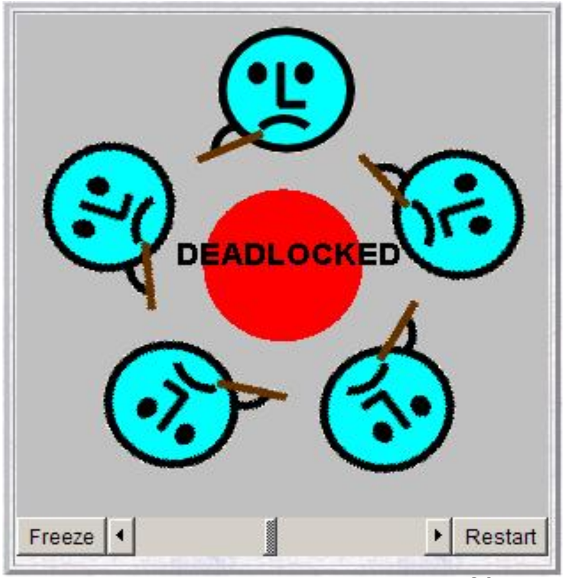
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

4 Necessary Conditions for Deadlock

Resource nature

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

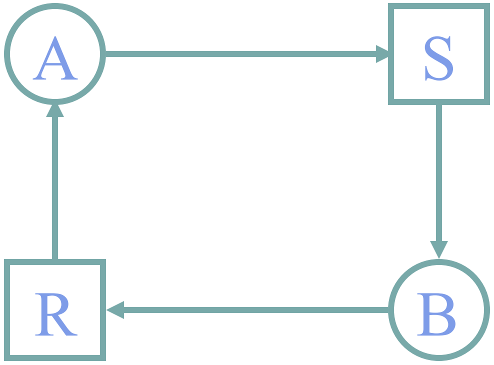
Program behavior



Eliminating ***any*** condition eliminates deadlock!

Eliminate Competition for Resources?

- If running A to completion and then running B, there will be no deadlock
- Generalize this idea for all processes?
- Is it a good idea?



Previous example

Four Possible Strategies

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem after occurring
3. Dynamic avoidance (by OS, programmer help)
   - Careful allocation
4. Prevention (by programmer, practically)
   - Negate one of the four conditions

Four Possible Strategies

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem after occurring
3. Dynamic avoidance (by OS, programmer help)
   - Careful allocation
4. Prevention (by programmer, practically)
   - Negate one of the four conditions

Puzzle time

- Every day, Jack arrives at the train station from work at 5 pm. His wife leaves home in her car to meet him there at exactly 5 pm, and drives him home. One day, Jack gets to the station an hour early, and starts walking home, until his wife meets him on the road. They get home 30 minutes earlier than usual. How long was he walking?

- Distances are unspecified. Speeds are unspecified, but constant. Give a number which represents the answer in minutes.

Four Necessary Conditions for Deadlock

Resource nature

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Program behavior

Eliminating ***any*** condition eliminates deadlock!

4.1 Prevention: Remove Mutual Exclusion

- Some resources can be made sharable
  - Read-only files, memory, etc
- Some resources are not sharable
  - Printer, mutex, etc
- Reduce/remove as much ME as possible

- How?
  - Using hardware primitives (e.g., TAS, ldl&stc)
  - Wait-free synchronization

- Dining philosophers problem? (work out on your own)

4.2 Prevention: (change app) Remove Hold and Wait

- Two-phase locking
  - Phase I:
    - Try to lock all needed resources at the beginning, atomically (how?)
  - Phase II:
    - If successful, use the resources & release them
    - If not, release all resources and start over

- This is how telephone company prevents deadlocks
- 2 Problems with this approach?

- Dining philosophers problem? (work out on your own)

4.3 Prevention: Preemption (w/o changing app)

- Make scheduler aware of resource allocation
- Method
  - If a request from a process holding resources cannot be satisfied, preempt the process and release all resources
  - Schedule it only if the system satisfies all resources
- Applicability?
  - Preemptable resources:
    - CPU registers, physical memory
  - Difficult for OS to understand app intention

4.3 Prevention: Preemption (changing app using hw primitive)

- Trylock approach

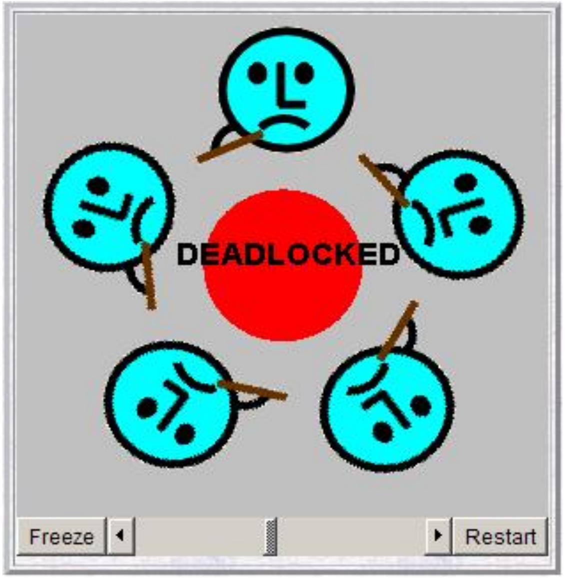
```
top:
lock(L1);
if (trylock(L2) == -1) {
        unlock(L1);
        goto top;
}
```

- Problems? Livelock

- Dining philosophers problem?

4.4 Prevention: (change app) No Circular Wait

- Impose some order of requests for all resources
- How?
- Does it always work?
- Can we prove it?



- How is this different from two-phase locking?

Dining philosophers?

Four Possible Strategies

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem afterwards
3. Dynamic avoidance (by OS & programmer)
   - Careful allocation
4. Prevention (by programmer & OS)
   - Negate one of the four conditions

3. Deadlock Avoidance

Definition:

An algorithm that is run by the OS whenever a process requests resources, the algorithm avoids deadlock by <u>denying or postponing</u> the request

if

it finds that accepting the request <u>could</u> put the system in an <u>unsafe state</u> (one where deadlock could occur).

Deadlock Avoidance

- Requirement:
  - each process declares the *maximum number* of resources of each type it *may* need

- Key idea:
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure there can never be a deadlock condition
  - *No matter what future requests will be*

Limitations of deadlock avoidance

- Needs to know the entire set of tasks that must be run and the locks that they need
- Reduce concurrency
- Not used widely in practice
  - E.g., used in embedded systems

Four Possible Strategies

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem afterwards
3. Dynamic avoidance (by OS)
   - Careful allocation
4. Prevention (mostly by programmer)
   - Negate one of the four conditions (mostly 2 and 4)

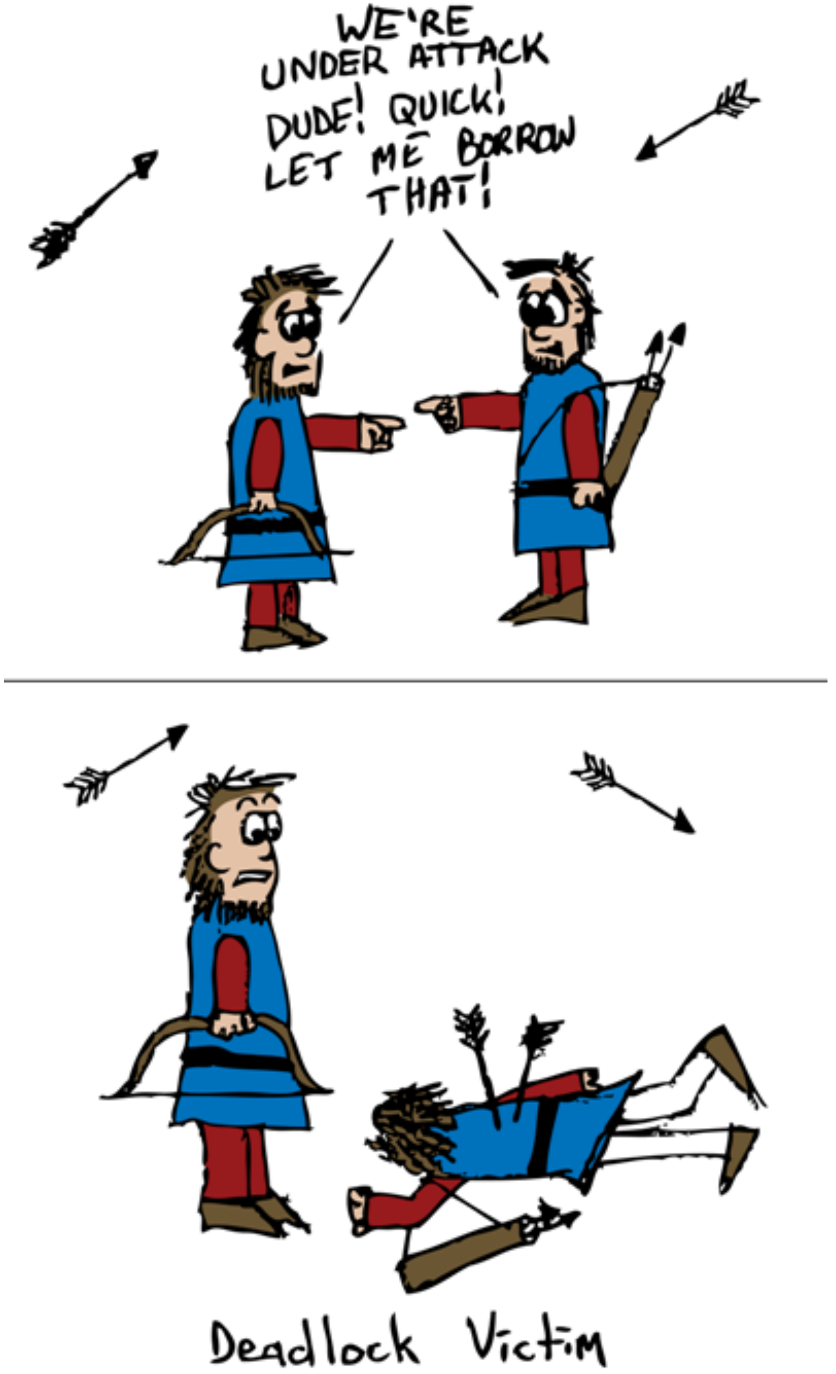
2. Deadlock Detection

- Programmer does nothing
- Allow system to enter deadlock state
- Run some detection algorithm
  - E.g., build a resource graph to check for cycles
- Try to recovery somehow
  - E.g., reboot the machine

2. Deadlock Detection

- If deadlock is a rare case, this makes a lot of sense
- TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)
  - "Not everything worth doing is worth doing well"

Deadlock examples



Conditions for Deadlock

Guns don't cause deadlocks – people do

Non-Deadlock Bugs

- Atomicity-Violation Bugs
  - The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).
  - Real example in MySQL

Thread 1::

```
if (thd->proc_info) {
        ...
        fputs(thd->proc_info, ...);
        ...
}
```

Not Atomic!

Thread 2::

```
thd->proc_info = NULL;
```

Non-Deadlock Bugs

- Order-Violation Bugs
  - The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)

```
Thread 1::
void init() {
    ...
    mThread =
    PR_CreateThread(mMain, ...);
        ...
}
```

```
Thread 2::
void mMain(...) {
        ...
        mState = mThread->State;
        ...
}
```

Non-Deadlock Bugs

- Real study found that non-deadlock bugs happen more often than deadlock bugs and are the majority of concurrency bugs!