Deadlocks

ECE595
Sep 20

Y. Charlie Hu

Review: Preemptive CPU Scheduling

- What is in it?
  - Mechanism + policy
  - Mechanisms fairly simple
  - Policy choices harder

Review: Evolution of CPU scheduling polices

- Don't know future → optimal policy is hard
- FIFO, Round-Robin, SJF all have merits
  - Tradeoffs are tricky to analyze

  → occationally we can prove things
- Need a general framework to encompass all

  → Priority scheduling
- But coming up with priorities is tricky

  → Multiple queue scheduling
- But statically assigning queues not flexible

  → multi-level feedback queue scheduling

[lec5] 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

[lec5] 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

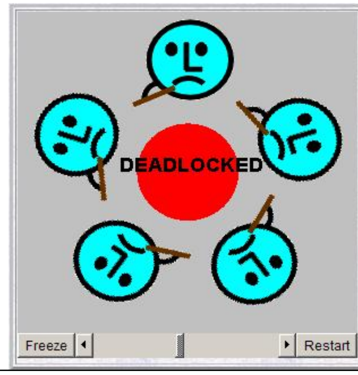
wait(fulls)
wait(mutex)
get full buffer from pool of fulls
signal(mutex)

consume data in buffer

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

empties = 0; fulls = N

[lec8] Dining Philosophers Problem (using semaphores)



Phil(i)

think;

wait(i.right)
wait(i.left);

eat;

signal(i.left);
signal(i.right);

The essence of our cond var solution?

Deadlock Example

- A law passed by the Kansas legislature early in the 20th century (in part):

"When two trains approach each other at a crossing, both come to a full stop and neither should start up again until the other has done."

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 $R_1, R_2, \ldots, R_m$
    - *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, ...

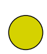
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Resource-Allocation Graph

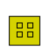
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow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

Resource-Allocation Graph (Cont.)

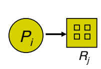
- Process



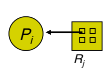
- Resource type with 4 instances



- P_i requests instance of R_j

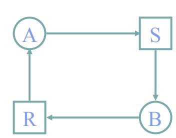


- P_i is holding an instance of R_j

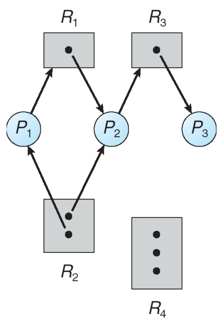


Example of a Resource Allocation Graph – one instance per type

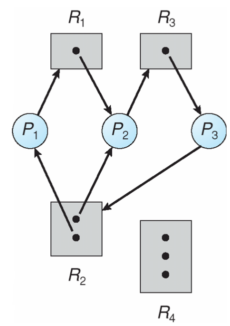
- What happens if there is a cycle in the resource allocation graph?



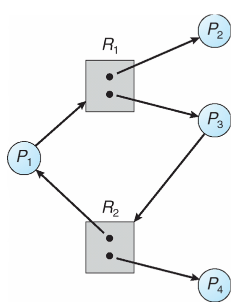
Example of a Resource Allocation Graph – multiple instances / type



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



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



Basic Facts

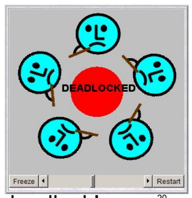
- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

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*

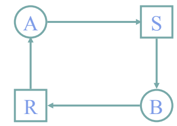
Program behavior



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

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!

4.1 Prevention: Remove Mutual Exclusion

- Some resources can be made sharable
  - Read-only files, memory, etc
- Some resources are not sharable
  - Printer, tape, mutex, etc
- Dining philosophers problem?

4.2 Prevention: (change app) Remove Hold and Wait

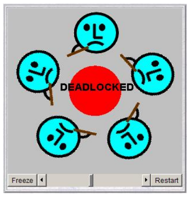
- Two-phase locking
  - Phase I:
    - Try to lock all needed resources at the beginning
  - Phase II:
    - If successful, use the resources & release them
    - If not, release all resources and start over
- This is how telephone company prevents deadlocks
- Dining philosophers problem? (use TSA)
- 2 Problems with this approach?

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
- 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 lockng?



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