Deadlocks (cont)

ECE595
Apr 14

Y. Charlie Hu

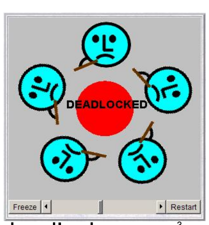
Deadlocks

- 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

4 Necessary Conditions for Deadlock

- *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*



Eliminating ***any*** condition eliminates deadlock

Four Possible Strategies

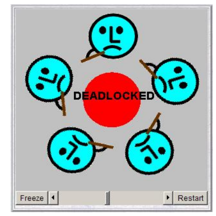
1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem afterwards
3. Dynamic avoidance (by OS)
   - Careful allocation
4. Prevention (mostly by programmer)
   - Negate one of the four conditions (mostly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

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
- 2 Problems with this approach?
- Dining philosophers problem?

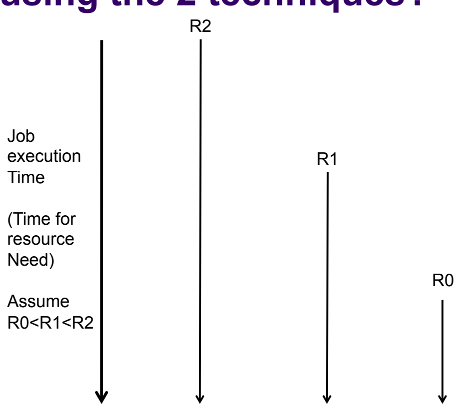
4.4 Prevention: (change app) No Circular Wait

- Impose some order of requests for all resource types
  - Single req for instances of same type



- How?
- Does it always work?
- Can we prove it?
- How is this different from two-phase lockng?

Example: how to program using the 2 techniques?



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 (by programmer & OS)
   - Negate one of the four conditions

3. Deadlock Avoidance

Definition:

The algorithm is run by the OS whenever a process requests resources, the algorithm avoids deadlock by denying or postponing the request

if

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

How Deadlock Avoidance works

- 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*

Resource-Allocaition State

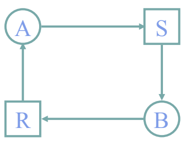
- Resource-allocation state is defined by
  - number of available resources
  - number of allocated resources
  - maximum demands of each process

Avoidance algorithms

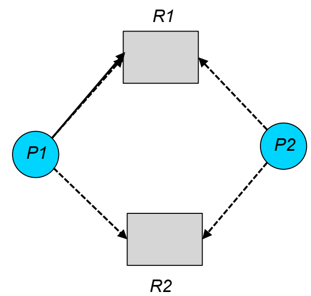
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use Banker's algorithm

Example of a Resource Allocation Graph – one inst per type

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

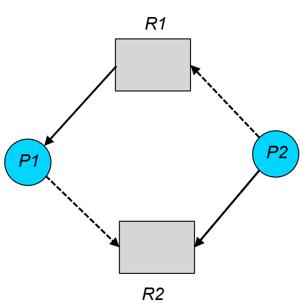


Deadlock Avoidance – 1 instance/type



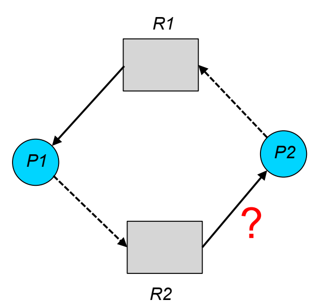
Claim edge – maximal resource requests

Deadlock Avoidance – 1 instance/type



Claim edge

(if granted) Unsafe State In Resource-Allocation Graph



Claim edge

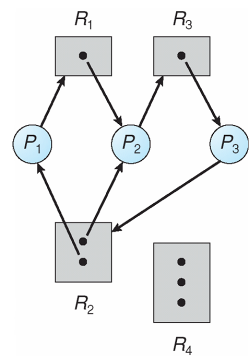
Why is this avoidance conservative?

Avoidance algorithms

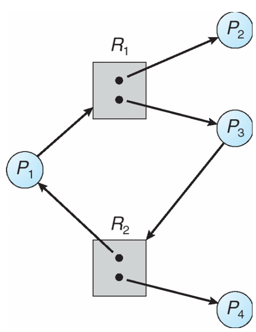
- Single instance of a resource type
  - Detect cycles in the resource-allocation graph

→ Multiple instances of a resource type
  - Use the banker's algorithm

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



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



Cycle detection is not enough! ☹

Safe State

OS can control future
When future comes

Worst case scenario

- System is in safe state if it can allocate resources to each process, up to its maximum, in some order and still avoid a deadlock.
  - Meaning: even in the worse case, there will be a way out
- System is in a safe state if there exists a safe sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that
  - for each P_i, the resources that P_i may still request can be satisfied by currently available resources + resources held by all P_j, with $j < i$

Safe State

- System is in a safe state if there exists a safe sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that
  - for each P_i, the resources that P_i may still request can be satisfied by currently available resources + resources held by all P_j, with $j < I$
- What is the way out?
  - P1, P2, …., Pn

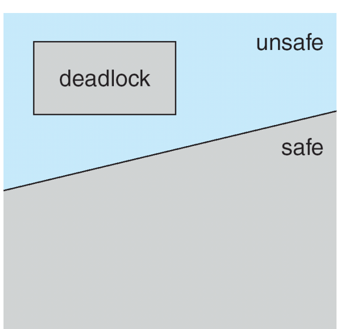
Example of safe state

- System has 12 drives

	Maximum needs	Currently held	Remaining needs	Available drives: 3
P0	10	5	5	
P1	4	2	2	
P2	9	2	7	

- Is it safe now?
- Is it safe to allocate 1 to P2?

Safe, Unsafe, Deadlock State



Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
  - Deadlock happens under worse case expectation: that *Pi* will request for all its remaining needs next
- Avoidance ⇒ ensure that a system will never enter an unsafe state

Banker's Algorithm (by Dijkstra)

- Multiple instances per resource type
- Each process must a priori claim maximum use
- When a process requests some resources
  - Algorithm checks if granted, system still in safe state?
    - If yes, grant resources
    - else; put the process to wait
  - Safe state checking algo (section 7.5.3.1) runs in polynomial time

Data Structures

- Available [m]: available resources of each type
- Max [n][m]: maximum demand of each resource by each process
- Allocation [n][m]: num of resources currently allocated to processes
- Need [n][m] = Max [n][m] – Allocation [n][m]

Safe State

- System is in safe state if there exists a safe sequence < $P_1, P_2, \ldots, P_n$ > of ALL the processes in the systems such that
  - for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

Safety checking Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work[ ] = Available [ ]
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find an *i* such that both:
   (a) *Finish* [*i*] = *false*
   (b) $Need_i$ [ ] ≤ *Work [ ]*
   If no such *i* exists, go to step 4
3. *Work* [ ] = *Work [ ]* + $Allocation_i$*[ ]*
   Finish[*i*] = *true*
   go to step 2
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state, else not safe

Does the ordering matter?

Finish

F	F	…	F	F

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types:
  - A (10 inst), B (5 inst), and C (7 inst)

Snapshot at time T_0:

	Allocation	*Max*	*Need*	*Available*
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

System in a safe state since seq < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available, (1,0,2) ≤ (3,3,2)
- If granted:

	Allocation	*Need*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

  - seq. < P_1, P_3, P_4, P_0, P_2> satisfies safety req.

Example: P_4 Request (3,3,0)?

- Check that Request ≤ Available, (3,3,0) ≤ (3,3,2)
- If granted:

	Allocation	*Need*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	0 0 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	3 3 2	1 0 1	

- Is there a safe sequence?

Example: P_0 Request (0,2,0)?

- Check that Request ≤ Available, (0,2,0) ≤ (3,3,2)
- If granted:

	Allocation	*Need*	*Available*
	A B C	A B C	A B C
P_0	0 3 0	7 2 3	3 1 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Is there a safe sequence?

Deep Thinking: Deadlock Avoidance algorithm

- Is it a scheduling algorithm?

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 detection algorithm
- Try to recovery somehow

Detection algorithm (7.6.2)

- Almost identical to safe state checking algorithm
  - Just change *Need* matrix to *Request* matrix
    - If finding a sequence leading to finishing, no deadlock
    - Else there is a deadlock

Several Instances of a Resource Type

- **Available**: A vector of length m indicates the number of available resources of each type.
- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An n x m matrix indicates the current request of each process. If *Request* [i][j] = k, then process P_i is requesting k more instances of resource type. R_j.

Detection Algorithm

1. Initialize *Work* and *Finish* as:
   (a) *Work* = *Available*
   (b) For i = 1,2, …, n, if $Request_i \neq 0$, *Finish*[i] = false; otherwise, *Finish*[i] = *true*
2. Find an index i such that both:
   (a) *Finish*[i] == *false;* (b) $Request_i \leq Work$
   If no such i exists, go to step 4
3. *Work* = *Work* + $Allocation_i$
   Finish[i] = *true*
   go to step 2
4. If *Finish*[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[i] == *false*, then P_i is deadlocked

Example of Detection Algo

- 5 processes P_0 through P_4;
- 3 resource types: A (7 instances), *B* (2), *C* (6)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- <P_0, P_2, P_3, P_1, P_4> will result in finishing

Example 2

- Another snapshot

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 1	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes;
  - Deadlock exists, → processes P_1, P_2, P_3, and P_4

Detection Algorithm: practical questions

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph
  - which of the many deadlocked processes "caused" the deadlock?

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed
  - How much longer to completion
  - Resources the process has used
  - Resources the process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Summary on solving deadlocks

4. Prevention
   - Negate one of the four conditions
     - Promising: avoid hold & wait, avoid cycle

3. Dynamic avoidance
   - Careful allocation (by OS)

2. Detection and recovery (by OS)
   - Fix the problem afterwards

Why do we care about this?

- Two possibilities
  - Bitten all the time by this
  - Strive to never have it
- Why does it happen?
  - Fact of life for some applications (e.g. symmetrical behavior of all processes)
  - Faulty programming in others

Reading assignment

- Read chapter 7