Deadlock

Comp 305 Lecture 4

This Week

- Lecture on transactions and deadlock
- Tutorial on Lab Exercise 2
- Homework

    Chap 6, problems 18 and 19;
    Chap 7, problems 4, 7 and 9.

Transactions

- A critical section ensures mutual exclusion and sequential execution.
- An atomic transaction gives the effect of a single atomic action.

```
begin transaction
 read X
 if (X >= a) then {
  read Y
  X = X - a
  Y = Y + a
  write X
  write Y
 }
end transaction
```

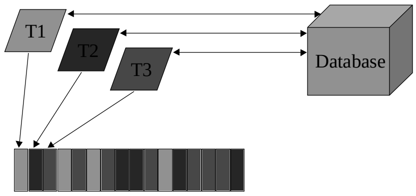

Assumptions

- Three types of memory
  - Volatile
  - Non-volatile
  - Stable
- Commit
  - All changes become visible
- Abort
  - Nothing has happened

Log Based Transactions

```
begin transaction         → <Ti,start>
 read X
 if (X >= a) then {
  read Y
  X = X - a
  Y = Y + a
  write X
  write Y                 → <Ti,X,oldV,newV>
 }                        → <Ti,Y,oldV,newV>
end transaction
                          → <Ti,commit>
```


How It Works

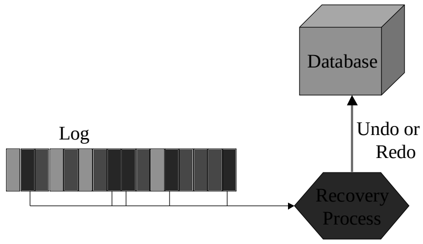


Log

Recovery

- Operations
  - undo(Ti)
  - redo(Ti)
- Possible states of Ti
  - Committed, but were values written?
  - Aborted, but were values restored?
  - Unfinished
- Checkpoints

Recovery At Work



Concurrent Transactions

```
T1 begin transaction
  read X
  modify X
  write X
  ...                    T2 begin transaction
  do some more             read X
end transaction            read Y
                           modify Y using X
                           write Y
                         end transaction
```


Conflicting Operations

```
T1 begin transaction
  read X
  modify X
  write X
  ...                    T2 begin transaction
  do some more             read X
end transaction            read Y
                           modify Y using X
                           write Y
                         end transaction
```

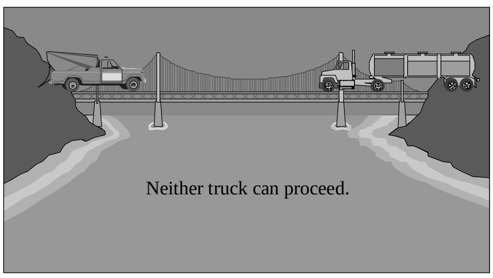
abort restores X
incorrect Y written

Locks

- A shared lock allows multiple reads
- An exclusive lock allows single write
- Serial schedule

  All locks are acquired before any are released.

What is Deadlock?



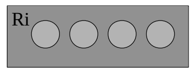
Deadlock Defined

- *A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.*
- Necessary Conditions -
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

Resource Allocation Graph

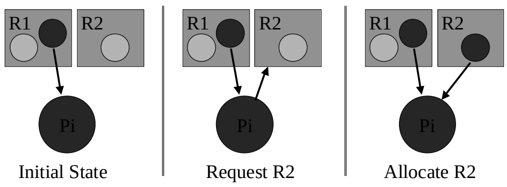
- Set of processes [P1, P2, ...]

- Set of resources [R1, R2, ...]



Resource Allocation Graph/2

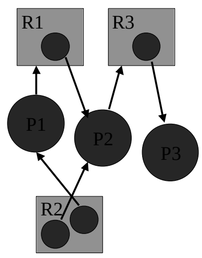
- Construct a graph by drawing
  - Edges from Pj to Ri on request (Pj, Ri)
  - Edges from Ri to Pj on allocation (Ri, Pj)



Initial State | Request R2 | Allocate R2

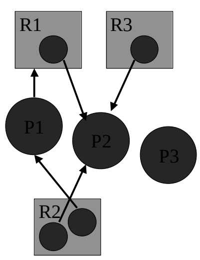
Cycles and Deadlock

- Deadlock?
  - P1 waiting
  - P2 waiting
  - P3 runnable



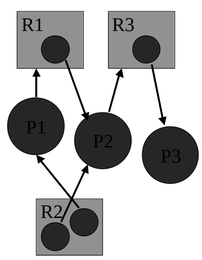
Cycles and Deadlock

- Deadlock?
  - P1 waiting
  - P2 waiting
  - P3 runnable
- P3 releases R3
  - P2 may run



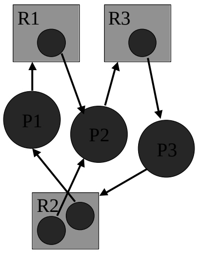
Cycles and Deadlock

- Deadlock?
  - P1 waiting
  - P2 waiting
  - P3 runnable



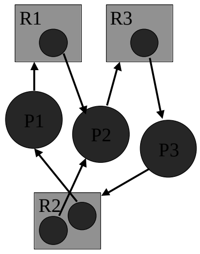
Cycles and Deadlock

- Deadlock?
  - P1 waiting
  - P2 waiting
  - P3 runnable
- P3 requests R2



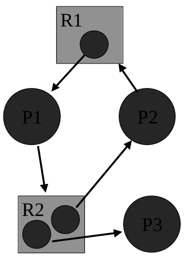
Cycles and Deadlock

- Deadlock?
  - P1 waiting
  - P2 waiting
  - P3 runnable
- P3 requests R2
- Circular wait
- Deadlock



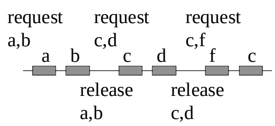
Is Cycle Sufficient?

- Cycle exists
- P3 may release R2
- P1 may continue
- Cycle broken
- No deadlock



Deadlock Prevention

- Eliminate 1 of 4 necessary conditions.
- Mutual exclusion
- Hold and wait



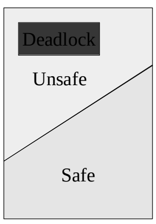
- No preemption
- Circular wait

Deadlock Algorithms

- Straightforward prevention is inefficient.

	max request	alloc
P0	10	5
P1	4	2
P2	9	2

- System has 12
- <P1, P0, P2> is SAFE



A Formal View

- *A state is SAFE iff there exists a sequence of processes <P0, P1, ...> such that all processes can complete.*
- All other states are UNSAFE.
- All DEADLOCK states are UNSAFE.
- From a SAFE state system can allocate resource to avoid DEADLOCK

Bankers Algorithm

- Overview
  - Each process declares maximum need
  - Request granted iff system remains in safe state
- Definitions
  - Available[1..M]
  - Max[1..N][1..M]
  - Allocation[1..N][1..M]
  - Need[1..N][1..M] = Max - Allocation
  - Request[1..M]

The Bankers Algorithm

1 IF NOT *Request* <= NEED[*i*] THEN *stop*

2 WHILE *Available* < *Request* DO *wait*

3 Construct a new state by

Available = Available - Request
Allocation[*i*] = *Allocation*[*i*] + *Request*
Need[*i*] = *Need*[*i*] - *Request*

4 IF *new state is unsafe* THEN *wait*

The Safety Algorithm

1 *Work = Available*
Finish = FALSE

2 Find i such that Pi hasn't finished but could
Finish[*i*] = *FALSE*
Need[*i*] <= *Work*

3 Assume Pi completes
Work = Work + Allocation[*i*]
Finish[*i*] = *TRUE*
goto 2

4 IF *for all i Finish*[*i*] = *TRUE* THEN *SAFE*

Deadlock Detection

- If we don't avoid it we must detect deadlock.
- Run Safety Algorithm on current state

2 substitute
Request[*i*] <= *work* for *need*[*i*]<= *Work*

4 If *Finish*[*i*] = *FALSE* THEN *Pi is deadlocked*

A Deadlock Detection Algorithm

1 *Work = Available*
Finish = FALSE

2 Find i such that Pi hasn't finished but could
Finish[*i*] = *FALSE*
Request[*i*] <= *Work*

3 Assume Pi completes
Work = Work + Allocation[*i*]
Finish[*i*] = *TRUE*
goto 2

4 If *Finish*[*i*] = *FALSE* THEN *Pi is deadlocked*

Breaking a Deadlock

- Break circular wait
  - Kill all deadlocked processes
  - Kill deadlocked processes one-by-one
  - Some processes may resist death
- Preempt resources
  - Implies rollback to a checkpoint