

Deadlocks

Chapter 7

Contents

- Definitions
 - Resources
- Conditions for deadlock
- Deadlock issues and handling them
 - Deadlock prevention: Havender's algorithms
 - Deadlock avoidance
 - Deadlock detection
 - Recovery from deadlock
- Representations for dealing with deadlocks
 - Resource allocation graph
 - Wait-For graphs

Definitions

Indefinite postponement vs. deadlock

- A process is **indefinitely postponed** if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes (different than **deadlock**)
 - i.e., Logically the process may proceed but the system never gives it the CPU
- A process is in **deadlock** if it cannot proceed with computation even if it is given the CPU.

Resources

- A resource is a “commodity” needed by a process
- Examples of computer resources
 - Devices that require exclusive access: Printers, Tape drives, CPU
 - Tables
 - Memory (as in a page frame)
 - Files
 - Synchronization resources: Semaphores, locks, etc.

Resources

- Processes need access to resources in reasonable order
- Suppose a process holds resource A and requests resource B
 - at same time another process holds B and requests A
 - both are blocked and remain so
- Deadlocks occur when ...
 - processes are granted **exclusive access** to resources

Introduction to Deadlocks

- Formal definition of deadlock:
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is the release of a currently held resource
- None of the processes in the set can ...
 - run
 - release resources
 - be awakened

Resources

- Resources can be either:
 - **Serially reusable**: e.g., CPU, memory, disk space, I/O devices, files
 - Acquire-use-release
 - **Consumable**: produced by a process, needed by a process; e.g., messages, buffers of information, interrupts
 - Create-acquire-use (**resource ceases to exist after it has been used, so it is not released**)

Resources

Resources can be either:

- **Preemptible**: e.g., CPU, main memory
- **Non-preemptible**: e.g., Tape drives, printers

And resources can be either:

- **Shared** among several processes
- **Dedicated exclusively** to a single process

Conditions for Deadlock

- The following four conditions are both **necessary** and **sufficient** for deadlock:
 1. Mutual exclusion condition
 2. Hold and wait condition
 3. No preemption condition
 4. Circular wait condition

Mutual Exclusion

- Processes claim **exclusive** control of the resources they require
 - Each resource assigned to only 1 process

Hold-and-Wait (wait-for) Condition

- Processes hold resources already allocated to them while waiting for additional resources

No Preemption Condition

- Resources cannot be removed from the processes holding them until used to completion

Circular Wait Condition

- A **circular chain** of processes exists in which each process holds one or more resources that are requested by the next process in the chain

Deadlock Strategies

- Four main strategies for dealing with deadlocks:
 - prevention
 - negating one of the four necessary conditions (Havender's approach) in the design
 - dynamic avoidance
 - careful resource allocation
 - detection and recovery
 - just ignore the problem altogether (ostrich algorithm)

Increasing amount of concurrency
↓

Deadlock Issues

- **Prevention**: design a system in such a way that deadlocks cannot occur, at least with respect to **serially reusable resources**
- **Avoidance**: impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches

Deadlock Issues

- **Detection:** in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved
- **Recovery:** after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over

Deadlock Prevention

- Havender's Approach
 - If any one of the four necessary conditions for deadlock (with reusable resources) is denied, deadlock is impossible.
 - Design system so that one of these conditions can never be met.

Deny Mutual Exclusion

- We don't want to deny this; exclusive use of resources is an important feature. However, sometimes its possible—
 - Virtualize and page it out: effectively preempt
 - Virtual memory
 - Virtual network connections
 - Virtual disk
 - Caches
 - CPU preemption
- } Break deadlocks by virtualizing these resources

Deny Mutual Exclusion

- Some devices (such as printer) can be spooled
 - only the printer daemon uses printer resource
 - thus deadlock for printer eliminated
- Not all devices can be spooled
- Principle:
 - avoid assigning resource when not absolutely necessary
 - as few processes as possible actually claim the resource

Deny Hold-and-Wait Condition

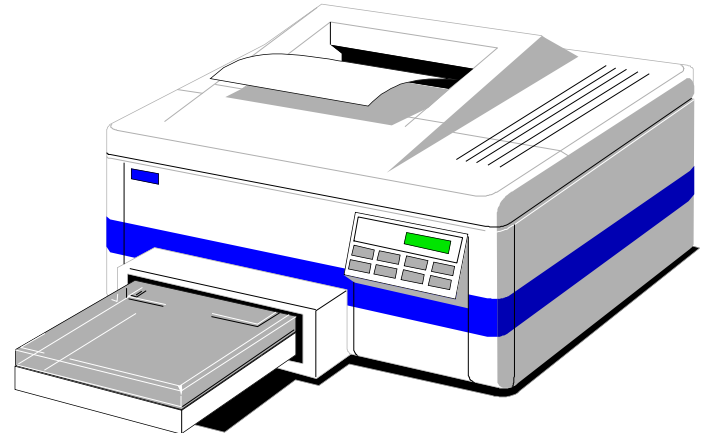
- Force each process to request all required resources at once. It cannot proceed until all resources have been acquired
 - a process never has to wait for what it needs
 - Effectively, everything turns into doing one process at a time sequentially.
- Problems
 - may not know required resources at start of run
 - also ties up resources other processes could be using
- Variation:
 - process must give up all resources
 - then request all immediately needed

Deny No Preemption Condition

- If a process holding some reusable resources makes a further request which is denied, and it wishes to wait for the new resources to become available, it must release all resources currently held and, if necessary, request them again along with the new resources. Thus, resources are removed from a process holding them

Deny No Preemption Condition

- This is not always a viable option
- Consider a process given the printer
 - halfway through its job
 - now forcibly take away printer
 - !!??



Deny Circular Wait Condition

Resource ordering:

- All resource types are numbered.
Processes must request resources in numerical order; if a resource of order N is held, the only resources which can be requested must be of order $> N$

Deny circular wait condition

Example:

- Assume we have the following numbered resources:
 1. Printer
 2. Scanner
 3. Plotter
 4. Tape drive
 5. CD ROM drive
- Now, a process can request an printer (1) followed by a plotter (3)
- But, a process cannot request a plotter (3) first and then request an printer (1).

Example: Solution to cigarette smokers problem

Agent process:

```
/* The shared data structures are */  
Semaphore a[3]; /* initially set to 0 */  
Semaphore agent; /* initially set to 1 */  
/* The agent process code is as follows: */  
Agent() {  
    repeat {  
        /* Set i, j to a value between 0 and 2. */  
        agent.P();  
        a[i].V();  
        a[j].V();  
    } until (false);  
}
```

Example: Solution to cigarette smokers problem

Smoker process:

/* Each smoker process needs two ingredients represented by integers r and s each with value between 0 and 2.

*/

```
Smoker() {  
    repeat  
        a[r].P();  
        a[s].P();  
        smoke();  
        agent.V();  
    until (false);  
}
```

What's the problem with this solution?

Example: Solution to cigarette smokers problem

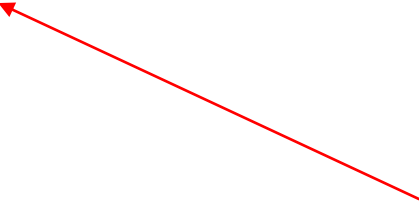
Smoker process:

/* Each smoker process needs two ingredients represented by integers r and s each with value between 0 and 2.

In the following code, **assume $r < s$** , so we have resource ordering and we prevent deadlock situations.

```
*/  
Smoker() {  
    repeat  
        a[r].P();  
        a[s].P();  
        smoke();  
        agent.V();  
    until (false);  
}
```

Resource ordering



Example: Solution to cigarette smokers problem

Write code with resource ordering built-in:

Smoker process:

/* Each smoker process needs two ingredients represented by integers r and s each with value between 0 and 2.

In the following code, **assume $r < s$** , so we have resource ordering and we prevent deadlock situations.

```
*/  
Smoker() {  
    repeat  
        if (r < s) {  
            a[r].P(); // acquire the smaller numbered resource first  
            a[s].P();  
        }  
        else {  
            a[s].P();  
            a[r].P();  
        }  
        smoke();  
        agent.V();  
    until (false);  
}
```

Past exam question

For the following implementation of atomic transfer, say whether it either (i) **works**, (ii) **doesn't work**, or (iii) **is dangerous** — that is, sometimes it works and sometimes it doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work.

The problem statement is as follows: The atomic transfer routine dequeues an item from one queue and enqueues it on another. The transfer must appear to occur atomically: there should be no interval of time during which an external thread can determine that an item has been removed from one queue but not yet placed on another. Note that even though atomic transfer works on two queues at a time, there could be many queues in the system from which atomic transfers can occur. Therefore, the implementation, in addition, must be highly concurrent—it must allow multiple transfers between unrelated queues to happen in parallel. You may assume that `queue1` and `queue2` never refer to the same queue. Also assume that `Append()` and `Remove()` methods for queue and `Acquire()` and `Release()` methods for locks work as normally expected.

`AtomicTransfer()` given on next page:

AtomicTransfer

```
void AtomicTransfer(Queue *queue1, *queue2)
{
    Item thing; /* thing being transferred */
    queue1->lock.Acquire();
    thing = queue1->Remove();
    if (thing != NULL) {
        queue2->lock.Acquire();
        queue2->Append(thing);
        queue2->lock.Release();
    }
    queue1->lock.Release();
}
```

What's wrong with it?

Answer

This is dangerous, since it may (but does not always) lead to deadlock. If one thread transfers from A to B, and another transfers from B to C and another from C to A, then you can get deadlock if they all acquire the lock on the first queue before any of them acquire the second.

Solution: use resource ordering.

Rewritten AtomicTransfer()

```
void AtomicTransfer(Queue *queue1, *queue2)
{
    Item thing; /* thing being transferred */
    if (queue1 < queue2) {
        queue1->lock.Acquire();
        queue2->lock.Acquire();
    } else { // queue2 < queue1
        queue2->lock.Acquire();
        queue1->lock.Acquire();
    }
    thing = queue1->Remove();
    if (thing != NULL) {
        queue2->Append(thing);
    }
    // release order doesn't matter
    queue1->lock.Release();
    queue2->lock.Release();
}
```

Locks are acquired only in increasing order of pointer addresses. That's an example of resource ordering.

Summary of approaches to deadlock prevention

| Condition | Approach |
|------------------|---------------------------------|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resources initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
 - deadlocks occur very rarely
 - cost of prevention is high
- UNIX and Windows take this approach
- It is a trade-off between
 - Convenience/cost
 - correctness

Deadlock prevention

- Previous techniques (prevention)
 - Static → may not result in maximum utilization of resources and maximum concurrency.

Deadlock Avoidance

- A more dynamic approach than deadlock prevention.
- Given: set of resources and set of customers
- Use Banker Rules: look ahead and see if deadlock can occur in the future

Deadlock Avoidance

- Use Banker Rules: look ahead and see if deadlock can occur in the future
 1. Each customer tells banker the maximum number of resources it needs (so things are decidable).
 2. Customer borrows resources from banker
 3. Customer returns resources to banker (Customer eventually pays back loan)

Deadlock Avoidance

- Banker only lends resources if the system will be in a **safe state after the loan.**
- **Safe state** - there is a lending sequence such that all customers can eventually take out loan
 - There is a future avenue where all customers can get a loan
 - If there is a safe state, someone somewhere can make progress.
- **Unsafe state** - a deadlock will occur if customers demand max claims
 - Worst case analysis.
 - There being an unsafe state doesn't mean deadlock will necessarily occur.

How to Compute Safety

Given:

n kinds of resources

p processes

Set **P** of processes

struct {

 resource needs[n],

 alloc[n]


} ToDo[p]

available[n]

Each process' need
(max-allocated)



How many resources
is the process given by
the banker.



How many resources are
Available in the pool.



How to Compute Safety

```
while there exists a p in P such that
  for all i such that (ToDo[p].needs[i] < available[i])
    { do for all j
      available[j] += ToDo[p].alloc[j];
      P = P - p;
    }
```

→ If P is empty then system is safe

This means we found
A future path where all
Processes could get
Max resources they need
And finish. → no deadlock

This means that the process p
Can be done with the resources
And release them. So, we release
The resources in our search for an
Available path by adding them to
The available pool.

Worksheet for Bankers Example

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 2 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

P1 requests 1A, 0 B, 2 C resources so allocate

And Run Safety Test

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 3 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

If P1 requests max resources, can complete

Allocate to P1, Then

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 3 | 2 | 2 | 3 | 2 | 2 | 0 | 0 | 0 | 2 | 1 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Release - P1 Finishes

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 0 | 0 | 0 | 3 | 2 | 2 | 3 | 2 | 2 | 5 | 3 | 2 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Now P3 can acquire max resources and release

Release - P3 Finishes

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 0 | 0 | 0 | 3 | 2 | 2 | 3 | 2 | 2 | 7 | 4 | 3 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Now P4 can acquire max resources and release

Release - P4 Finishes

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 0 | 0 | 0 | 3 | 2 | 2 | 3 | 2 | 2 | 7 | 4 | 5 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | | |
| P4 | 0 | 0 | 0 | 4 | 3 | 3 | 4 | 3 | 3 | | | |

Now P2 can acquire max resources and release

Release - P2 Finishes

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 0 | 0 | 0 | 3 | 2 | 2 | 3 | 2 | 2 | 10 | 4 | 5 |
| P2 | 0 | 0 | 0 | 9 | 0 | 2 | 9 | 0 | 2 | | | |
| P3 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | | |
| P4 | 0 | 0 | 0 | 4 | 3 | 3 | 4 | 3 | 3 | | | |

Now P0 can acquire max resources and release

So P1 Allocation (1 0 2) Is Safe

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 3 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

P0 Requests 2 B

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 3 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Try to Allocate 2 B to P0

Run Safety Test

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 3 | 0 | 7 | 5 | 3 | 7 | 2 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 1 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

No Processes may get max resources and release

So Unsafe State- Do Not Enter

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 3 | 0 | 7 | 5 | 3 | 7 | 2 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 1 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Return to Safe State and do not allocate resource

P0 Suspended Pending Request

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 10 | 5 | 5 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 | 2 | 3 | 0 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

When enough resources become available, P0 can awake

Comment

- If P1 requested 2 Bs, then it's safe to allocate them
- P0 cannot request and have allocated 2 Bs because that would prevent any other process from completing if they need their maximum claim

Just Because Its Unsafe...

- P0 could have been allocated 2 Bs and a deadlock might not have occurred if:
 - P2 say didn't use its maximum resources but finished using the resources it had

If P1 Doesn't Need Max...

| Process | Alloc | | | Max | | | Need | | | Available | | |
|---------|-------|---|---|-----|---|---|------|---|---|-----------|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 3 | 0 | 7 | 5 | 3 | 7 | 2 | 3 | 10 | 5 | 5 |
| P1 | 0 | 0 | 0 | 3 | 2 | 2 | 3 | 2 | 2 | 5 | 1 | 2 |
| P2 | 3 | 0 | 0 | 9 | 0 | 2 | 6 | 0 | 2 | | | |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | | | |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 | | | |

Then P0 would have finished...

Discussion

- The banker's algorithm is conservative --- it reduces parallelism for safety sake
- What is the trade-off?
 - Actual cost is the trade-off
 - Depends on the application. Example:
 - In an airline reservation system the cost of a deadlock and hence a failure of everything would be high.
 - On your PC, the cost may not be so high to have a deadlock and reboot the machine.

Banker Solution Issues

- Process may not terminate (processes have to eventually terminate)
- Process may request more than claim (processes should not ask for more resources than their declared max)
- A process may suffer indefinite postponement
 - Solution is to check for aged process
 - Select an allocation sequence that includes aged process
 - Only select requests that follow that sequence until aged process executes

Deadlock Detection

- **Goal:** max amount of concurrency
- Allocate resources but also check to see if deadlock occurred!
- One resource per type can use wait-for graph to do deadlock detection
 - Look for cycles in wait-for graph

Visualizing/representing Deadlock

- What's a good way of showing in a diagram when processes get into a deadlock?
 - Resource allocation graphs
 - Wait-for graphs

Resource Allocation Graph

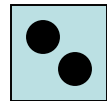
Process



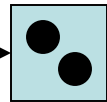
Resource



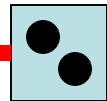
Resource Type



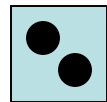
One process, two resources
same type.



Process requests one resource



Process acquires one resource



Process releases resource

Resource Allocation Graph

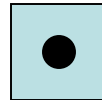
Process



Resource

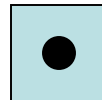


Resource Type

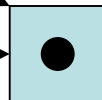
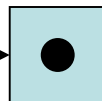


Two processes

Two resources



Two types



Each processes requests
one resource of each type

Resource Allocation Graph

Process



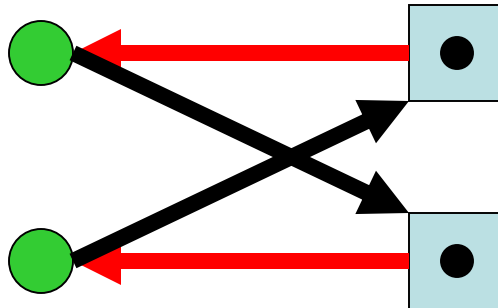
Resource



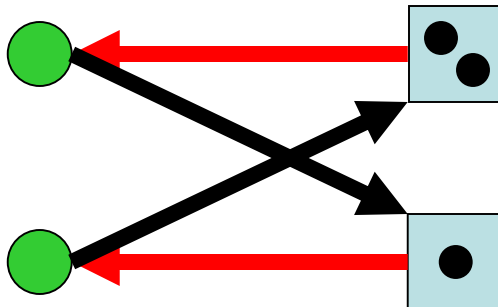
Resource Type



There is a
cycle in the
graph



Each processes acquires
one resource only ---
DEADLOCK



An extra resource would
avoid deadlock

Resource Allocation Graph

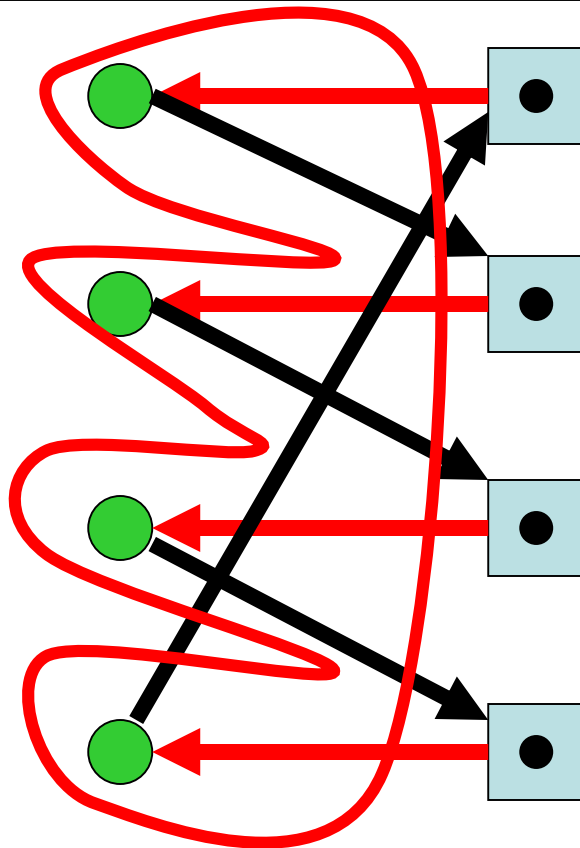
Process



Resource



Resource Type



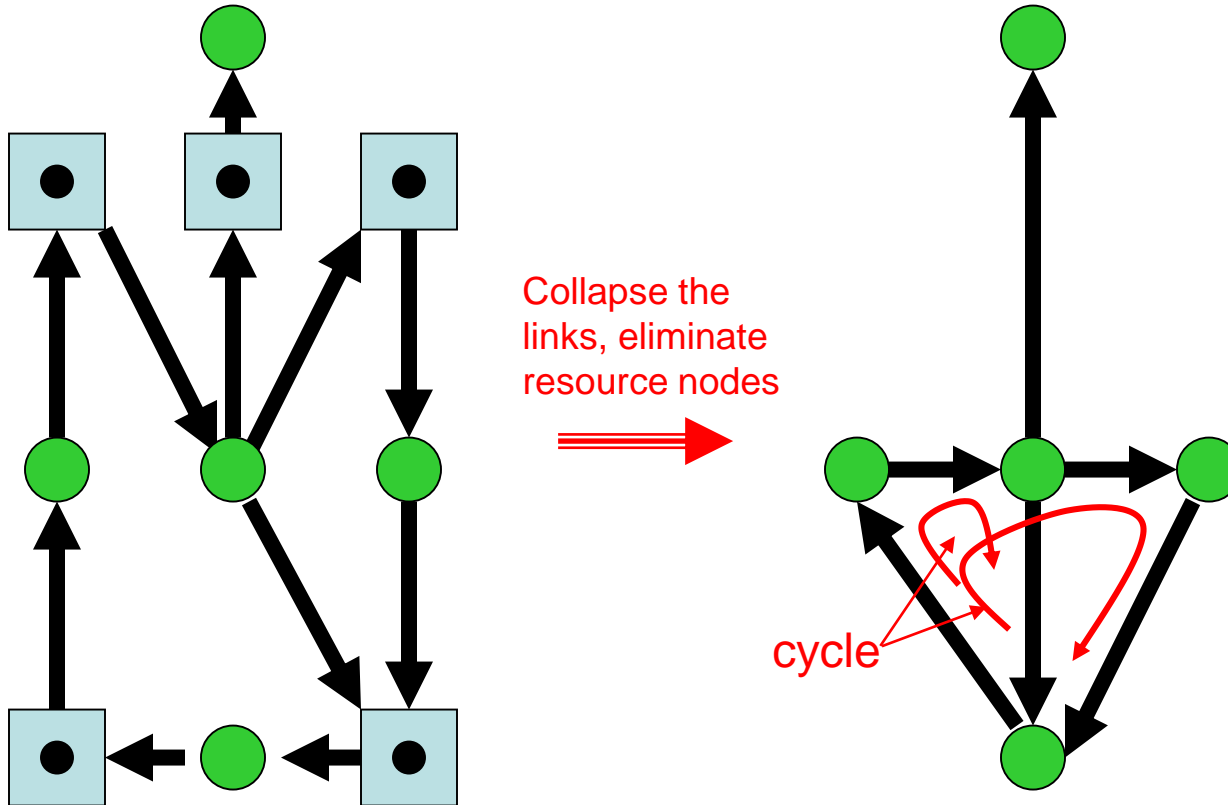
NECESSARY and
SUFFICIENT CONDITION

IF

- Single Resource per Type
and
- Cycle in Resource Graph
THEN DEADLOCK

Wait for Graphs

- Assumption: 1 resource/resource type.



Resource Allocation Graph Corresponding Wait For Graph

Deadlock Detection

- Multiple resources per type
 - Run variant of banker's algorithm to see if processes can finish
 - Optimistic version --- check only that at least one process can finish

Detection with Multiple Resource of Each Type

Resources in existence
($E_1, E_2, E_3, \dots, E_m$)

Snapshot:
 m resources
 n processes

Resources available
($A_1, A_2, A_3, \dots, A_m$)

Current allocation matrix

Request matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row n is current allocation
to process n

Row 2 is what process 2 needs

Every resource is either allocated or available. So, for resource j , the following holds:

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

Detection with Multiple Resource of Each Type

- We will use vector comparisons to do deadlock detection:
 - Define for two vectors **A** and **B**
$$\mathbf{A} \leq \mathbf{B} \text{ holds iff } A_i \leq B_i \text{ for } 1 \leq i \leq m$$
 - That is, if each element of **A** is \leq each element of **B**, then $\mathbf{A} \leq \mathbf{B}$

Detection with Multiple Resource of Each Type

- **Deadlock detection algorithm:**
 1. Scan the rows of **C** matrix looking for an unmarked process, P_i , for which the i -th row of matrix **R**, $\mathbf{R}_i \leq \mathbf{A}$.
 2. If such a process is found,
 - $\mathbf{A} = \mathbf{A} + \mathbf{C}_i$. (Process P_i finishes and releases resources it holds)
 - Mark P_i
 - Go to 1.
 3. If no such process is found, the algorithm terminates.
- When algorithm terminates, all unmarked processes are deadlocked.

Detection with Multiple Resource of Each Type

$$\begin{array}{c} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{CD Roms} \end{array} \\ E = (4 \quad 2 \quad 3 \quad 1)$$

$$\begin{array}{c} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{CD Roms} \end{array} \\ A = (2 \quad 1 \quad 0 \quad 0)$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- No deadlock in the system: process 3 can run and complete, then process 2 can run and complete and then process 1 can complete.

Deadlock Detection

- How often should the detection algorithm be run? It depends—
 - Run every time a resource is allocated
 - Run periodically (every 10 secs)
 - Wait and see if there is enough progress made
- How many processes will be affected by deadlock? Issue is cost
 - Depends on when/how often deadlock detection is done.

Cost of Deadlock Detection Algorithm

- Multiple resources - $O(mn^2)$, n processes
 m resource types
- Wait-for, single resource - $O(n^2)$ -- find cycle in graph

Recovery from Deadlock

- **Recovery through preemption**
 - take a resource from some other process
 - depends on nature of the resource

Recovery From Deadlock

Recovery through killing processes.

- Kill deadlocked processes and release resources
 - May be easier to implement than the next one.
- Kill one deadlocked process at a time and release its resources (hope that the remaining processes now can finish)
- Potential for doing this again and again. No guarantee that deadlock will not occur again.

Recovery From Deadlock

Recovery through rollback

- Rollback all or one of the processes to a checkpoint that occurred before they requested any resources
 - checkpoint a process periodically (intermediate state)
 - use this saved state to restart the process if it is found to be deadlocked
 - Killing is a special case which means rolling back to the initial state.
 - With rollback, it's difficult to prevent indefinite postponement; could keep making progress and rolling back.

Other Issues

Two-Phase Locking

- Phase One
 - process tries to lock all records it needs, one at a time
 - if needed record found locked, release all the locks and start over
 - (no real work done in phase one)
- If phase one succeeds, it starts second phase,
 - performing updates
 - releasing locks
- Note similarity to requesting all resources at once
- Algorithm works where programmer can arrange
 - program can be stopped, restarted

Nonresource Deadlocks

- Possible for two processes to deadlock
 - each is waiting for the other to do some task
- Can happen with semaphores
 - each process required to do a *down()* on two semaphores (*mutex* and another)
 - if done in wrong order, deadlock results

Summary

- Four conditions for deadlock.
- Prevention means prevent one of the four conditions from happening
 - Cost of slowing down system because we may not achieve max parallelism.

Summary

- In general, deadlock detection or avoidance is expensive
 - Costs paid at different times:
 - Avoidance: all the time
 - Detection: only when we get into trouble.

Summary

- Must evaluate cost of deadlock against detection or avoidance costs
 - Deadlock in PC's is not as important to OS vendors than slowing down their OS for deadlock detection.
 - This results in stratified marketplace
 - Reliable OS's
 - Unreliable OS's.
- Deadlock avoidance and recovery may cause indefinite postponement