Chapter 3: Deadlocks
Overview

- Resources
- Why do deadlocks occur?
- Dealing with deadlocks
  - Ignoring them: ostrich algorithm
  - Detecting & recovering from deadlock
  - Avoiding deadlock
  - Preventing deadlock
Resources

- Resource: something a process uses
  - Usually limited (at least somewhat)
- Examples of computer resources
  - Printers
  - Semaphores / locks
  - Tables (in a database)
- Processes need access to resources in reasonable order
- Two types of resources:
  - Preemptable resources: can be taken away from a process with no ill effects
  - Nonpreemptable resources: will cause the process to fail if taken away
When do deadlocks happen?

- **Suppose**
  - Process 1 holds resource A and requests resource B
  - Process 2 holds B and requests A
  - Both can be blocked, with neither able to proceed

- **Deadlocks occur when …**
  - Processes are granted exclusive access to devices or software constructs (resources)
  - Each deadlocked process needs a resource held by another deadlocked process

![Diagram of deadlocks](image-url)
Using resources

- Sequence of events required to use a resource
  - Request the resource
  - Use the resource
  - Release the resource

- Can’t use the resource if request is denied
  - Requesting process has options
    - Block and wait for resource
    - Continue (if possible) without it: may be able to use an alternate resource
    - Process fails with error code

- Some of these may be able to prevent deadlock…
What is a deadlock?

- Formal definition:
  “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 release of a currently held resource

- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened
Four conditions for deadlock

- **Mutual exclusion**
  - Each resource is assigned to at most one process

- **Hold and wait**
  - A process holding resources can request more resources

- **No preemption**
  - Previously granted resources cannot be forcibly taken away

- **Circular wait**
  - There must be a circular chain of 2 or more processes where each is waiting for a resource held by the next member of the chain
Resource allocation graphs

- Resource allocation modeled by directed graphs
- Example 1:
  - Resource R assigned to process A
- Example 2:
  - Process B is requesting / waiting for resource S
- Example 3:
  - Process C holds T, waiting for U
  - Process D holds U, waiting for T
  - C and D are in deadlock!
Dealing with deadlock

- How can the OS deal with deadlock?
  - Ignore the problem altogether!
    - Hopefully, it’ll never happen…
  - Detect deadlock & recover from it
  - Dynamically avoid deadlock
    - Careful resource allocation
  - Prevent deadlock
    - Remove at least one of the four necessary conditions

- We’ll explore these tradeoffs
Getting into deadlock

A
Acquire R
Acquire S
Release R
Release S

B
Acquire S
Acquire T
Release S
Release T

C
Acquire T
Acquire R
Release T
Release R

Deadlock!
Not getting into deadlock…

- Many situations *may* result in deadlock (but don’t have to)
  - In previous example, A could release R before C requests R, resulting in no deadlock
  - Can we always get out of it this way?

- Find ways to:
  - Detect deadlock and reverse it
  - Stop it from happening in the first place
The Ostrich Algorithm

- Pretend there’s no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness
Detecting deadlocks using graphs

- Process holdings and requests in the table and in the graph (they’re equivalent)
- Graph contains a cycle => deadlock!
  - Easy to pick out by looking at it (in this case)
  - Need to mechanically detect deadlock
- Not all processes are deadlocked (A, C, F not in deadlock)

<table>
<thead>
<tr>
<th>Process</th>
<th>Holds</th>
<th>Wants</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>D</td>
<td>U</td>
<td>S,T</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>V</td>
</tr>
<tr>
<td>F</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>G</td>
<td>V</td>
<td>U</td>
</tr>
</tbody>
</table>
Deadlock detection algorithm

- General idea: try to find cycles in the resource allocation graph
- Algorithm: depth-first search at each node
  - Mark arcs as they’re traversed
  - Build list of visited nodes
  - If node to be added is already on the list, a cycle exists!
- Cycle == deadlock

For each node N in the graph {
    Set L = empty list
    unmark all arcs
    Traverse (N, L)
}

If no deadlock reported by now, there isn’t any

define Traverse (C, L) {
    If C in L, report deadlock!
    Add C to L
    For each unmarked arc from C {
        Mark the arc
        Set A = arc destination
        /* NOTE: L is a local variable */
        Traverse (A, L)
    }
}
Resources with multiple instances

- Previous algorithm only works if there’s one instance of each resource

- If there are multiple instances of each resource, we need a different method
  - Track current usage and requests for each process
  - To detect deadlock, try to find a scenario where all processes can finish
  - If no such scenario exists, we have deadlock
Deadlock detection algorithm

current=avail;
for (j = 0; j < N; j++) {
  for (k=0; k < N; k++) {
    if (finished[k])
      continue;
    if (want[k] < current) {
      finished[k] = 1;
      current += hold[k];
      break;
    }
    if (k==N) {
      printf "Deadlock!\n";
      // finished[k]==0 means process is in
      // the deadlock
      break;
    }
  }
}

Note: want[j], hold[j], current, avail are arrays!
Recovering from deadlock

- Recovery through preemption
  - Take a resource from some other process
  - Depends on nature of the resource and the process

- Recovery through rollback
  - Checkpoint a process periodically
  - Use this saved state to restart the process if it is found deadlocked
  - May present a problem if the process affects lots of “external” things

- Recovery through killing processes
  - Crudest but simplest way to break a deadlock: kill one of the processes in the deadlock cycle
  - Other processes can get its resources
  - Preferably, choose a process that can be rerun from the beginning
    - Pick one that hasn’t run too far already
Resource trajectories

Two process resource trajectories
Safe and unsafe states

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

Free: 3

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

Free: 2

Demonstration that the first state is safe

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>-0</td>
</tr>
</tbody>
</table>

Free: 0

Demonstration that the second state is unsafe
Banker's Algorithm for a single resource

Bankers’ algorithm: before granting a request, ensure that a sequence exists that will allow all processes to complete

- Use previous methods to find such a sequence
- If a sequence exists, allow the requests
- If there’s no such sequence, deny the request

Can be slow: must be done on each request!
### Banker's Algorithm for multiple resources

#### Example of banker's algorithm with multiple resources

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Resources assigned**

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Resources still needed**

- E = (6342)
- P = (5322)
- A = (1020)
Preventing deadlock

- Deadlock can be completely prevented!
- Ensure that at least one of the conditions for deadlock never occurs
  - Mutual exclusion
  - Circular wait
  - Hold & wait
  - No preemption
- Not always possible…
Eliminating mutual exclusion

- Some devices (such as printer) can be spooled
  - Only the printer daemon uses printer resource
  - This eliminates deadlock for printer
- Not all devices can be spooled
- Principle:
  - Avoid assigning resource when not absolutely necessary
  - As few processes as possible actually claim the resource
Attacking “hold and wait”

- Require processes to request resources before starting
  - A process never has to wait for what it needs
- This can present problems
  - A process may not know required resources at start of run
  - This also ties up resources other processes could be using
    - Processes will tend to be conservative and request resources they might need
- Variation: a process must give up all resources before making a new request
  - Process is then granted all prior resources as well as the new ones
  - Problem: what if someone grabs the resources in the meantime—how can the process save its state?
Attacking “no preemption”

- This is not usually a viable option
- Consider a process given the printer
  - Halfway through its job, take away the printer
  - Confusion ensues!
- May work for some resources
  - Forcibly take away memory pages, suspending the process
  - Process may be able to resume with no ill effects
Attacking “circular wait”

- Assign an order to resources
- Always acquire resources in numerical order
  - Need not acquire them all at once!
- Circular wait is prevented
  - A process holding resource $n$ can’t wait for resource $m$ if $m < n$
  - No way to complete a cycle
    - Place processes above the highest resource they hold and below any they’re requesting
    - All arrows point up!
Deadlock prevention: summary

- Mutual exclusion
  - Spool everything

- Hold and wait
  - Request all resources initially

- No preemption
  - Take resources away

- Circular wait
  - Order resources numerically
Example: two-phase locking

- **Phase One**
  - Process tries to lock all data it needs, one at a time
  - If needed data found locked, start over
  - (no real work done in phase one)

- **Phase Two**
  - Perform updates
  - Release locks

- Note similarity to requesting all resources at once
- This is often used in databases
- It avoids deadlock by eliminating the “hold-and-wait” deadlock condition
“Non-resource” 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
- Semaphores could be thought of as resources…
Starvation

- Algorithm to allocate a resource
  - Give the resource to the shortest job first
- Works great for multiple short jobs in a system
- May cause long jobs to be postponed indefinitely
  - Even though not blocked

Solution

- First-come, first-serve policy

Starvation can lead to deadlock

- Process starved for resources can be holding resources
- If those resources aren’t used and released in a timely fashion, shortage could lead to deadlock