Deadlock

CS 450: Operating Systems
Sean Wallace <swallac6@iit.edu>
deadlock |ˈdedˌläk|
noun
1 [in sing.] a situation, typically one involving opposing parties, in which no progress can be made: an attempt to break the deadlock.

–New Oxford American Dictionary
Traffic Gridlock
Software Gridlock

mutex_A.lock()
mutex_B.lock()

# critical section

mutex_B.unlock()
mutex_A.unlock()

mutex_B.lock()
mutex_A.lock()

# critical section

mutex_B.unlock()
mutex_A.unlock()}
Necessary Conditions for Deadlock
That is, what conditions need to be true (of some system) so that deadlock is possible?

(Not the same as causing deadlock!)
1. Mutual Exclusion

Resources can be held by process in a mutually exclusive manner
2. Hold & Wait

While holding one resource (in mutex), a process can request another resource
3. No Preemption

One process can not force another to give up a resource; i.e., releasing is voluntary
4. Circular Wait

Resource requests and allocations create a cycle in the resource allocation graph.
Resource Allocation
Graphs
Circular wait is absent = no deadlock
All 4 necessary conditions in place; Deadlock!
In a system with *only* single-instance resources,

necessary conditions $\iff$ deadlock
Cycle without Deadlock!
Not practical (or always possible) to detect deadlock using a graph

— but convenient to help us reason about things
Approaches to Dealing with Deadlock
1. Ostrich algorithm (Ignore it and hope it never happens)

2. Prevent it from occurring (avoidance)

3. Detection & recovery
Deadlock Avoidance
Approach 1:

Eliminate necessary condition(s)
• Mutual exclusion?
  • Eliminating mutex requires that all resources be *sharable*
  • When not possible (e.g., disk, printer), can sometimes use a *spooler process*
• But what about semaphores, file locks, etc.?
  
  • Not all resources are spoolable
  
  • *Cannot eliminate mutex* in general
• Hold & Wait?
  • Elimination requires resource requests to be all-or-nothing affair
  • If currently holding, needs to release all before requesting more
In practice, very inefficient & starvation is possible!

—*cannot eliminate hold & wait*
• No preemption?

• Alternative: allow process to preempt each other and “steal” resources

• Mutex locks cannot be counted on to stay locked!

• In practice, cannot eliminate this either!
Circular Wait is where it’s at.
• Simple mechanism to prevent wait cycles:
  • Order all resources
  • Require that processes request resources in order
But *impractical* — can not count on processes to need resources in a certain order

...and forcing a certain order can result in poor *resource utilization*
Approach 2:

Intelligently prevent circular wait
Possible to create a cycle (with one edge)?
Possible to create a cycle (with one edge)?
It’s quite possible that $P_2$ won’t need $R_2$, or, maybe $P_2$ will release $R_1$ before requesting $R_2$, but we don’t know if/when…
Preventing circular wait means avoiding a state where a cycle is an imminent *possibility*.
To predict deadlock, we can ask processes to “claim” all resources they need in advance.
Graph with “claim edges”
P_2 requests R_1
Convert to allocation edge; no cycle
P₁ requests R₂
If we convert to an allocation edge...
Cycle involving claim edges!
Means that if processes fulfill their claims, we can not avoid deadlock!
i.e., $P_1 \rightarrow R_1$, $P_2 \rightarrow R_2$
$P_1 \rightarrow R_1$ should be blocked by the kernel, even if it can be satisfied with available resources
This is a “safe” state…i.e., no way a process can cause deadlock directly (i.e., without OS alloc)
Idea: if granting an incoming request would create a cycle in a graph with claim edges, deny that request (i.e., block the process)

—approve later when no cycle would occur
P₂ releases R₁
Now OK to approve $P_1 \rightarrow R_2$ (unblock $P_1$)
Should we still deny $P_1 \rightarrow R_2$?
Problem: this approach may incorrectly predict imminent deadlock when resources with multiple instances are involved
Requires a *more general* definition of “safe state”
Banker’s Algorithm

(by Edsger Dijkstra)
Basic idea:

- Define how to recognize system “safety”
- Whenever a resource request arrives:
  - Simulate allocation & check state
  - Allocate iff simulates state is safe
Some assumption we need to make:

1. A non-blocked process holding a resource will eventually release it

2. It is know a priori how many instances of each resource a given process needs
Safe State

• There exists a sequence \(<P_1, P_2, ..., P_n>\), where each \(P_k\) can complete with:
  
  • Currently available (free) resources
  
  • Resources held by \(P_1, ..., P_{k-1}\)
Data Structures

Processes $P_1, \ldots, P_n$, Resources $R_1, \ldots, R_m$:

available$[j]$ = num of $R_j$ available
max$[i][j]$ = max num of $R_j$ required by $P_i$
allocated$[i][j]$ = num of $R_j$ allocated to $P_i$
need$[i][j]$ = max$[i][j]$ - allocated$[i][j]$
Safety Algorithm

1. \( \text{finish}[i] \leftarrow \text{false} \ \forall \ i \in 1...n \)
   
   \( \text{work} \leftarrow \text{available} \)

2. Find \( i : \text{finish}[i] = \text{false} & \text{need}[i][j] \leq \text{work}[j] \ \forall \ j \)
   
   If none, go to 4

3. \( \text{work} \leftarrow \text{work} + \text{allocated}[i] ; \ \text{finish}[i] \leftarrow \text{true} \)
   
   Go to 2.

4. Safe state iff \( \text{finish}[i] = \text{true} \ \forall \ i \)
Incoming request represented by \( \text{request array} \)

\[ \text{request}[j] = \text{num of resource } R_j \text{ requested} \]

(A process can require multiple instances of more than one resource at a time)
Processing Request from $P_k$

1. If $\text{request}[j] \leq \text{need}[k][j] \ \forall \ j$, continue, else error

2. If $\text{request}[j] \leq \text{available}[j] \ \forall \ j$, continue, else block

3. Run safety algorithm with:
   1. $\text{available} \leftarrow \text{available} - \text{request}$
   2. $\text{allocated}[k] \leftarrow \text{allocated}[k] + \text{request}$
   3. $\text{need}[k] \leftarrow \text{need}[k] - \text{request}$
If safety algorithm fails, do not allocate, even if resources are available!

—either deny request or block caller
3 resources: A (10), B (5), C (7)

<table>
<thead>
<tr>
<th>Max</th>
<th>Allocated</th>
<th>Available</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
</tr>
<tr>
<td>P₀</td>
<td>7  5  3</td>
<td>0  1  0</td>
<td>3  3  2</td>
</tr>
<tr>
<td>P₁</td>
<td>3  2  2</td>
<td>2  0  0</td>
<td>7  4  3</td>
</tr>
<tr>
<td>P₂</td>
<td>9  0  2</td>
<td>3  0  2</td>
<td>1  2  2</td>
</tr>
<tr>
<td>P₃</td>
<td>2  2  2</td>
<td>2  1  1</td>
<td>6  0  0</td>
</tr>
<tr>
<td>P₄</td>
<td>4  3  3</td>
<td>0  0  2</td>
<td>0  1  1</td>
</tr>
</tbody>
</table>

- Safe state: \(<P₁, P₃, P₀, P₂, P₄>\)
- \(P₃\) needs \(<0, 0, 1>\) additional
- \(P₀\) needs \(<0, 3, 0>\) additional
Banker’s Algorithm Discussion
1. Efficiency?

• How fast is it?

• How often is it run?
1. \( \text{finish}[i] \leftarrow \text{false} \quad \forall \ i \in 1 \ldots n \)

\( \text{work} \leftarrow \text{available} \)

\textit{for up to N processes, check M resources}

2. \( \textbf{Find i : finish}[i] = \text{false} \quad \& \quad \text{need}[i][j] \leq \text{work}[j] \quad \forall \ j \)

If none, go to 4

3. \( \text{work} \leftarrow \text{work} + \text{allocated}[i]; \quad \text{finish}[i] \leftarrow \text{true} \)

Go to 2. \textit{loop for N processes}

4. Safe state iff \( \text{finish}[i] = \text{true} \quad \forall \ i \)

\[ \mathcal{O}(N \cdot N \cdot M) = \mathcal{O}(N^2 \cdot M) \]
• How often to run?

• Need to run on every resource request

• Can’t relax this, otherwise system might become unsafe!
2. Assumption #1: processes will *eventually* release resources
• Assuming well-behaved processes

• Not 100% realistic, but what else can we do?
3. Assumption #2: a priori knowledge of max resource requirements
• Highly unrealistic

• Process resource needs are dynamic!

• Without this assumption, deadlock prevention becomes much harder…
Aside:
Decision problems, complexity theory, & the halting problem
A decision problem

input

decision algorithm

yes

no
• E.g.:
  • Is X evenly divisible by Y?
  • Is N a prime number?
  • Does string S contain pattern P?
A lot of important problems can be reworded as a decision problem:

E.g., traveling salesman problem (find the shortest tour through a graph)

⇒ is there a tour shorter than $L$?
Complexity theory *classifies* decision problems by their *difficulty*, and draws *relationships* between those problems & classes.
Class **P**: solutions to these problems can be found in polynomial time (e.g., $O(N^2)$)
Class **NP**: solutions to these problems can be *verified* in polynomial time

— but finding solutions may be harder! (i.e., superpolynomial)
Big open problem in CS:

\[ P = NP? \]
Why is this important?
All problems in NP can be reduced to another problem in the NP-complete class, and all problem in NP-complete can be reduced to each other.
If you can prove that *any* NP-complete problem is in $P$, then *all* NP problems are in $P$!

(More motivation: you also win $1M$)
If you can prove that $P \neq NP$, we can stop looking for fast solutions to many hard problems

(Motivation: you still win $1M)$
A decision problem

input

decision algorithm

yes

no
Will the system deadlock?

- **resources available request & allocations, running programs**
- **yes**
- **no**

Deadlock prevention
Will the system *halt* (or run forever)?

- yes
- no

The halting problem
E.g., write the function:

\[
\text{half}(f) \rightarrow \text{bool}
\]

- return true if \( f \) will halt
- return false otherwise
def halt(f):
    # your code here

def loop_forever():
    while True:
        pass

def just_return():
    return True

halt(loop_forever)  # => False
halt(just_return)    # => True
def gotcha():
    if halt(gotcha):
        loop_forever
    else:
        just_return

halt(gotcha)
Does this program halt?

Yes

No
Proof by contradiction:
the halting problem is undecidable
Generally speaking, deadlock prediction can be reduced to the halting problem.
I.e., determining if a system is deadlocked is, in general, *provably impossible*!!!
Deadlock Detection & Recovery
Basic approach: cycle detection
E.g., Tarjan’s strongly connected components algorithm; $O(|V| + |E|)$
Need only run on mutex resources and “involved” processes

...still, would be nice to reduce the size of the resource allocation graph
Actual resources involved are unimportant — only care about relationships between processes
Resource Allocation Graph
“Wait-for” Graph

P1 → P2 → P2 → P2
Substantial optimization!
...but not very useful when we have multi-instance resources (false positives are likely)
Deadlock Detection Algorithm
Important: do away with requirement of a priori resource need declarations
New assumption: processes can complete with *current allocation + all pending requests*

I.e., no future requests

Unrealistic!
(But we don’t have a crystal ball)
Keep track of all pending requests in:

\[ \text{request}[i][j] = \text{num of } R_j \text{ requested by } P_i \]
Detection Algorithm

1. \( \text{finish}[i] \leftarrow \text{all\_nil?}(\text{allocated}[i]) \quad \forall \; i \in 1 \ldots n \)
   \( \text{work} \leftarrow \text{available} \)

2. Find \( i : \text{finish}[i] = \text{false} \) \& \( \text{request}[i][j] \leq \text{work}[j] \quad \forall \; j \)
   If none, go to 4.

3. \( \text{work} \leftarrow \text{work} + \text{allocated}[i] ; \; \text{finish}[i] \leftarrow \text{true} \)
   Go to 2.

4. If \( \text{finish}[i] \neq \text{true} \quad \forall \; i \), system is deadlocked.

ignore processes that aren’t allocated anything
3 resources: A (7), B (2), C (6)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P₁</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- Not deadlocked: <P₀, P₂, P₁, P₃, P₄>
- P₂ requests <0, 0, 1>
Discussion
1. Speed?
1. `finish[i] ← all_nil?(allocated[i]) ∀ i ∈ 1…n`  
   `work ← available`

2. Find `i : finish[i] = false & request[i][j] ≤ work[j] ∀ j`  
   If none, go to 4.

3. `work ← work + allocated[i]; finish[i] ← true`  
   Go to 2.

4. If `finish[i] ≠ true ∀ i`, system is deadlocked.

Still $O(N \cdot N \cdot M) = O(N^2 \cdot M)$
2. When to run?
...as seldom as possible!

Tradeoff: the longer we wait between checks, the messier resulting deadlocks might be
3. Recovery?
• One or more processes must release resources:
  • Via forced termination
  • Resource preemption
  • System rollback

Sure, but how?
• Resource preemption only possible with certain types of resources

• No intermediate state

• Can be taken away and returned (while blocking process)

• E.g., mapped VM page
• Rollback requires process *checkpointing*:
  • Periodically autosave/reload process state
  • Cost depends on process complexity
  • Easier for special-purposes systems
• How many to terminate/preempt/rollback?

• At least one for each disjoint cycle

• Non-trivial to determine how many cycles and which processes!
• Selection criteria (who to kill) = minimize cost
  • # of processes
  • Completed run-time
  • # of resources held/needed
  • Arbitrary priority (no killing system process!)
Dealing with deadlock is hard!
• Moral of this and the concurrency material:

• Be careful with concurrent resource sharing

• Use concurrency mechanisms that avoid explicit locking whenever possible!