Deadlock

CS 450 : Operating Systems
Michael Lee <lee@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

mtx_A.lock()
mtx_B.lock()

    # critical section

mtx_B.unlock()
mtx_A.unlock()

mtx_B.lock()
mtx_A.lock()

    # critical section

mtx_B.unlock()
mtx_A.unlock()
§ Necessary conditions for Deadlock
i.e., what conditions need to be true (of some system) so that deadlock *is possible*? (not the same as *causing* deadlock!)
I. Mutual Exclusion

- resources can be held by processes in a mutually exclusive manner
II. Hold & Wait

- while holding one resource (in mutex), a process can request another resource
III. **No Preemption**

- one process can not force another to give up a resource; i.e., releasing is *voluntary*
IV. Circular Wait

- resource requests and allocations create a cycle in the resource allocation graph
Resource Allocation Graphs
Process

Resource

Request

Allocation
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 shareable
- 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 can not 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
$P_1$ requests $R_2$
if we convert to an allocation edge ...
cycle involving claim edges!
means that if processes fulfill their claims, we cannot avoid deadlock!
i.e., $P_1 \rightarrow R_1$, $P_2 \rightarrow R_2$
$P_1 \rightarrow R_2$ 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
$P_2$ releases $R_1$
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 simulated state is safe
some assumptions we need to make:

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

2. it is known *a priori* how many instances of each resource a given process needs
Safe State

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

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

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

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

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 request array

request[j] = num of resource Rj 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:
   - $\text{available} \leftarrow \text{available} - \text{request}$
   - $\text{allocated}[k] \leftarrow \text{allocated}[k] + \text{request}$
   - $\text{need}[k] \leftarrow \text{need}[k] - \text{request}$
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></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P₁</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P₂</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₄</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Safe state: \(<P₁, P₃, P₀, P₂, P₄>\)
- P₃ requests \(<0, 0, 1>\)
- P₀ requests \(<0, 3, 0>\)
Banker’s algorithm discussion
1. Efficiency?
   - how fast is it?
   - how often is it run?
1. finish[i] ← false ∀ i ∈ 1...n
   work ← available

For up to N processes, check M resources

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

3. work ← work + allocated[i]; finish[i] ← true
   Go to 2. loop for N processes

4. Safe state iff finish[i] = true ∀ i

\[ O(N \cdot N \cdot M) = O(N^2 \cdot M) \]
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 to 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
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 decision problems:

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 \( \mathbf{P} \): 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 $\text{NP}$ can be reduced to another problem in the $\text{NP}$-complete class,

and all problems in $\text{NP}$-complete can be reduced to each other)
if you can prove that any \textbf{NP-complete} problem is in \textbf{P}, then \textbf{all NP} problems are in \textbf{P}!

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

(motivation: you still win $1M$)
a decision problem
deadlock prevention

resources available, request & allocations, running programs

will the system deadlock?

yes

no

deadlock prevention
the halting problem

description of a program and its inputs

will the system \textit{halt} (or run forever)?

yes

no

the halting problem
e.g., write the function:

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

- 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 halt(f):
    # your code here

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, \textit{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
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 have no 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. 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.
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>Request</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₂ requests</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Available</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Not deadlocked: <P₀, P₂, P₁, P₃, P₄>
- P₂ requests <0, 0, 1>
Discussion
1. Speed?
1. \( \text{finish}[i] \leftarrow \text{all} \_\text{nil}?(\text{allocated}[i]) \quad \forall \ i \in 1 \ldots n \)
   \( \text{work} \leftarrow \text{available} \)

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

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

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

\[ \text{Still } O(N \cdot N \cdot M) = O(N^2 \cdot M) \]
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

cool, 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-purpose 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

- # processes
- completed run-time
- # resources held / needed
- arbitrary priority (no killing system processes!)
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!