

The Deadlock Problem

Law passed by the Kansas Legislature in early 20th century:

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone.”

Neil Groundwater has the following to say about working with Unix at Bell Labs in 1972:

... the terminals on the development machine were in a common room ... when one wanted to use the line printer. There was no spooling or lockout. `pr myfile > /dev/lp` was how you sent your listing to the printer. If two users sent output to the printer at the same time, their outputs were interspersed. Whoever shouted. “line printer!” first owned the queue.¹

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
 - Several processes may compete for a finite set of resources
 - Processes request resources and if a resource is not available, enter a wait state
 - Requested resources may be held by other waiting processes
 - Require divine intervention to get out of this problem
- A significant problem in real systems
- Little attention paid to the study of the problem because
 - Most multiprogramming systems limit parallelism to some system processes only, and only on a limited basis
 - Systems allocate resources to processes statically
- Deadlock problem becoming more important because of increasing use of multiprocessing systems (like real-time, life support, vehicle monitoring)
- Important in answering the question about the completion of a process
- Deadlocks can occur with
 - Serially reusable resources – printer, tape drive, memory
 - Serially consumable resources – messages

Examples of Deadlocks in Computer Systems

- File Sharing
 - Consider two processes p_1 and p_2
 - They update a file F and require a scratch tape during the updating
 - Only one tape drive T available
 - T and F are serially reusable resources, and can be used only by *exclusive access*
 - p_2 needs T immediately prior to updating
 - *request* operation
 - * Blocks the process requesting the resource
 - * Puts the process on the wait queue
 - * The process is to remain blocked until the requested resource is available
 - * If the resource is available, the process is granted an exclusive access to it.

¹Peter H. Salus. *A Quarter Century of UNIX*. Addison Wesley, Reading, MA. 1994

- *release* operation
 - * Returns the resource being released to the system
 - * Wakes up the process waiting for the resource, if any
- p_1 and p_2 may run as follows

$p_1:$ \vdots request(F); $r_1:$ request(T); \vdots \vdots release(T); release(F); \vdots	$p_2:$ \vdots request(T); \vdots $r_2:$ request(F); \vdots \vdots release(F); release(T); \vdots
--	--

- p_1 can block on T holding F while p_2 can block on F holding T

- Single Resource Sharing
 - Deadlock due to no memory being available and existing processes requesting more memory
 - Fairly common cause of deadlock
- Locking in Database Systems
 - Locking required to preserve the *consistency* of databases
 - Problem when two records to be updated by two different processes are locked
- Self-deadlock
 - Attempt to obtain a “lock” by a process that is already owned by it
- Deadlocking by nefarious users
 - Given by R. C. Holt


```
void deadlock ( task )
{
    wait (event);
} /* deadlock */
```
- Effective Deadlocks
 - Exemplified by Shortest Job Next Scheduling
- Deadlocks problem characterization
 - Deadlock Detection
 - * Process resource graphs
 - Deadlock Recovery
 - * “Best” ways of recovering from a deadlock
 - Deadlock Prevention
 - * Not allowing a deadlock to happen

A Systems Model

- Finite number of resources in the system to be distributed among a number of competing processes
- Partition the resources into several classes
- Identical resources assigned to the same class (CPU cycles, memory space, files, tape drives, printers)
- Allocation of any instance of resource from a class will satisfy the request
- State of the OS – allocation status of various resources
- Process actions
 - request a resource
 - * request a device
 - * open a file
 - * allocate memory
 - acquire/use a resource
 - * read from/write to a device
 - * read/write a file
 - * use the memory
 - release a resource
 - * release a device
 - * close a file
 - * free memory
- Resources acquired and used only through system calls
- Allocation record to be maintained as a *system table*
- State of the OS changed only by the process actions
- Processes to be modeled as nondeterministic entities
- Deadlock when every process is waiting for an event that can be caused by only one of the waiting processes
- System $\langle \sigma, \pi \rangle$
 - $\sigma - \{S, T, U, V, \dots\}$ – system states
 - $\pi - \{p_1, p_2, \dots\}$ – processes
- Process p_i – a partial function from system states into nonempty subsets of system states

$$p_i : \sigma \rightarrow \{\sigma\}$$

$S \xrightarrow{*} W$ implies

- $S = W$
- $S \xrightarrow{i} W$ for some p_i
- $S \xrightarrow{i} T$ for some p_i and T , and $T \xrightarrow{*} W$

- Process blocked if it cannot change state of the system

$$\nexists T | S \xrightarrow{i} T$$

- Process deadlocked in S if
 - Process is blocked in S

- No operations can make the process to be *unblocked*

p_i is deadlocked in S if

$$\forall T | S \xrightarrow{*} T$$

p_i is blocked in T

- Deadlock state S if $\exists p_i$ deadlocked in S
- *Safe state* S if

$$\forall T | S \xrightarrow{*} T$$

T is not a deadlock state

Deadlock Characterization

- Necessary conditions for deadlocks – Four conditions to hold simultaneously
 - Mutual exclusion – At least one resource must be held in a non-sharable mode
 - Hold and wait – Existence of a process holding at least one resource and waiting to acquire additional resources currently held by other processes
 - No preemption – Resources cannot be preempted by the system
 - Circular wait – Processes waiting for resources held by other waiting processes

Deadlock with Serially Reusable Resources

- *Serially reusable resource* – A finite set of identical units
 - The number of units is constant
 - Each unit is allocated to one and only one process
 - A process may release a unit only if it has previously acquired it

Deadlocks in Unix

- Possible deadlock condition that cannot be detected
- Number of processes limited by the number of available entries in the process table
- If process table is full, the `fork` system call fails
- Process can wait for a random amount of time before forking again
- Examples:
 - 10 processes creating 12 children each
 - 100 entries in the process table
 - Each process has already created 9 children
 - No more space in the process table \Rightarrow deadlock
 - Deadlocks due to open files, swap space
- Another cause of deadlock can be due to the inode table becoming full in the filesystem

Resource Allocation Graph

- Directed graph to describe deadlocks

- Set of vertices V consisting of
 - $P = P_1, P_2, \dots$ – Set of processes
 - Represent process nodes as circles
 - $R = R_1, R_2, \dots$ – Set of resource types
 - Represent resource nodes as squares with a dot (\cdot) representing each instance of the resource
- Set of edges E
 - Directed edge from P_i to R_j
 - * *request edge*
 - * denoted by $P_i \rightarrow R_j$
 - * P_i has requested for an instance of R_j and is currently waiting for that resource
 - Directed edge from R_j to P_i
 - * *assignment edge*
 - * denoted by $R_j \rightarrow P_i$
 - * an instance of R_j has been allocated to P_i
- No cycles in the graph \Rightarrow no deadlock
- Cycle in the graph \Rightarrow deadlock
- Each process involved in a cycle is deadlocked
- Cycle in the resource graph is necessary and sufficient condition for the existence of a deadlock
- If a graph contains several instances of a resource type, a cycle is not a sufficient condition for a deadlock but it still is a necessary condition

Deadlock Detection

- Simulate the *most favored execution* of each unblocked process
 - An unblocked process may acquire all the needed resources
 - Run and then release *all* the acquired resources
 - Remain dormant thereafter
 - Released resources may wake up some previously blocked process
 - Continue the above steps as long as possible
 - If any blocked processes remain, they are deadlocked
- Reduction of resource graphs
 - Process blocked if it cannot progress by either of the following operations
 - * Request
 - * Acquisition
 - * Release
 - Reduction of resource graph
 - * Reduced by a process p_i
 - by removing all edges to and from p_i
 - p_i is neither blocked nor isolated node
 - p_i becomes an isolated node
 - * Irreducible if the graph cannot be reduced by any process
 - * Completely reducible if a sequence of reductions deletes *all* the edges in the graph

– **Lemma 1.** *All reduction sequences of a given resource graph lead to the same irreducible graph.*

• Algorithms for Deadlock Detection with SR Resources

– **The Deadlock Theorem.** *S is a deadlock state if and only if the resource graph of S is not completely reducible.*

– Representation of resource graph

* Matrix representation – Two $n \times m$ matrices

· Allocation matrix A – processes as rows and resources as columns

$A_{ij}, i = 1, \dots, n, j = 1, \dots, m$ gives the number of units of resource R_j allocated to process p_i

· Request matrix B – Similar to A

B_{ij} gives the number of units of resource R_j requested by process p_i

* Linked list structure – Four lists

· Resources allocated to processes

$$p_i \rightarrow (R_x, a_x) \rightarrow (R_y, a_y) \rightarrow \dots \rightarrow (R_z, a_z)$$

· Resources requested by processes

· Allocation list of processes with respect to a resource

· Request list of processes with respect to a resource

* Available units vector (r_1, \dots, r_m)

– Deadlocks detected by looping through the process request lists, making reductions where possible

– Worst case execution time – mn^2

– Algorithm deadlock

```
// Check if the request for process pnum is less than or equal to available
// vector
```

```
bool req_lt_avail ( const int * req, const int * avail, const int pnum, \
                   const int num_res )
```

```
{
    int i ( 0 );
    for ( ; i < num_res; i++ )
        if ( req[pnum*num_res+i] > avail[i] )
            break;
    return ( i == num_res );
}
```

```
bool deadlock ( const int * available, const int m, const int n, \
                const int * request, const int * allocated )
```

```
{
    int work[m];           // m resources
    bool finish[n];       // n processes

    for ( int i ( 0 ); i < m; work[i] = available[i++] );
    for ( int i ( 0 ); i < n; finish[i++] = false );

    int p ( 0 );
    for ( ; p < n; p++ ) // For each process
    {
        if ( finish[p] ) continue;
        if ( req_lt_avail ( request, work, p, m ) )
        {
```

```

    finish[p] = true;
    for ( int i ( 0 ); i < m; i++ )
        work[i] += allocated[p*m+i];
    p = 0;
}
}

for ( p = 0; p < n; p++ )
    if ( ! finish[p] )
        break;

return ( p != n );
}

```

– Example

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	0	0
p_1	2	0	0	2	0	2			
p_2	3	0	3	0	0	0			
p_3	2	1	1	1	0	0			
p_4	0	0	2	0	0	2			

No deadlock with the sequence $\langle p_0, p_2, p_3, p_1, p_4 \rangle$

– Consider that p_2 makes an additional request for an instance of type C

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	0	0
p_1	2	0	0	2	0	2			
p_2	3	0	3	0	0	1			
p_3	2	1	1	1	0	0			
p_4	0	0	2	0	0	2			

deadlock with processes $\langle p_1, p_2, p_3, p_4 \rangle$

- $\text{reach}(a)$ – Set of nodes in the graph reachable from a.
- **Theorem 2. The Cycle Theorem.** A cycle in a resource graph is a necessary condition for deadlock.
- **Theorem 3.** If S is not a deadlock state and $S \xrightarrow{i} T$, then T is a deadlock state if and only if the operation by p_i is a request and p_i is deadlocked in T .
- Special Cases of Resource Graphs
 - **Knot:** A knot in a directed graph $\langle N, E \rangle$ is a subset of nodes $M \subseteq N$ such that $\forall a \in M, \text{reach}(a) = M$
 - Immediate Allocation
 - * Expedient States – All processes having requests are blocked
 - * Expedient state \Rightarrow A knot in the corresponding resource graph is a sufficient condition for deadlock
 - Single-Unit Resources – Cycle is sufficient and necessary condition for deadlock
- Recovery from Deadlock
 - Recovery by process termination
 - * Terminate deadlocked processes in a systematic way
 - * When enough processes terminated to recover from deadlock, stop terminations

- * Problems with the approach
 - If the process is in the midst of updating a file, its termination may leave the file in an incorrect state
 - If the process is in the midst of printing, the printer must be reset
- * Processes should be terminated based on some criterion/policy
 - Priority of a process
 - CPU time used and expected usage before completion
 - Number and type of resources being used (can they be preempted easily?)
 - Number of resources needed for completion
 - Number of processes needed to be terminated
 - Are the processes interactive or batch?
- * Minimum cost recovery
- * Cost of recovery
 - Cost of destroying a process
 - Cost of recovery from the next process state
- Recovery by resource preemption
 - * Enough resources to be preempted from processes and made available to deadlocked processes to resolve the deadlock
 - * Selecting a victim
 - * Rollback
 - * Prevention of starvation – Ensure that the resources are not always preempted from the same process
- Deadlock Prevention
 - Each process must request and acquire *all* the needed resources at the same time
 - Deny one of the required conditions for a deadlock
 - * Mutual Exclusion
 - Cannot be done for non-sharable resources (like printers)
 - Sharable resources (read-only files) do not require mutually exclusive access \Rightarrow cannot be involved in deadlock
 - Cannot deny mutual exclusion as some resources are inherently non-sharable
 - * Hold and Wait
 - Processes can request and acquire all the resources at one time
 - Request resources only if the process is holding none
If the process is holding any resources, they must be released before requests can be granted
 - Disadvantages
 1. Low resource utilization – resources may get allocated but not used for a long time
 2. Possibility of starvation – on popular resources
 - * No Preemption
 - If a process holding resources requests for another resource that cannot be immediately allocated, all currently held resources are preempted
 - Process restarted only when it regains *all* the resources
 - Suitable for resources whose state can be easily saved – CPU registers, memory
 - * Circular Wait
 - Impose a total ordering on all resource types
 - Each process requests resources in an increasing order of enumeration
 - If several instances of a resource required, a single request must be issued for all of them
- Deadlock Prevention based on Maximum Claims

- Also called Deadlock Avoidance
- A priori knowledge of maximum possible claims for each process
- Dynamically examine the resource allocation status to ensure that no circular wait condition can exist
- Resource allocation state
 - * Defined by the number of available and allocated resources, and the maximum demands of the processes
 - * Safe, if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock
- System in safe state only if there exists a *safe sequence*
- All unsafe states are not deadlock states
- An unsafe state may lead to a deadlock
- Example

* System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	2	2
p_2	9	2	7

Current availability : 3

Safe sequence: $\langle p_1, p_0, p_2 \rangle$

- * Possible to go from a safe state to an unsafe state
- Let the state after allocating two tapes to process p_1 be

System with 12 magnetic tape drives

Process	Max needs	Allocation	Current needs
p_0	10	5	5
p_1	4	4	0
p_2	9	2	7

Current availability : 1

Let p_2 request and acquire the last remaining tape drive

- * Mistake in allocating one more tape drive to p_2
- Problem: To detect the possibility of unsafe state and deny requests even if resources are still available
- Banker's Algorithm
 - * Based on banking system that never allocates its available cash such that it can no longer satisfy the needs of all its customers

• Deadlock Avoidance

- Requires a process to declare the maximum instances of each resource type needed
- Upon request, the system must determine whether the allocation will leave the system in a safe state
- Number of processes in the system – n
- Number of resource classes – m
- Data structures
 - * available
 - A vector of length m
 - Number of available resources of each type
 - $available[j] = k \Rightarrow k$ instances of resource class R_j are available
 - * maximum
 - An $n \times m$ matrix
 - Defines maximum demand for each process
 - $maximum[i, j] = k \Rightarrow$ process p_i may request at most k instances of resource class R_j
 - * allocation

- An $n \times m$ matrix
- Defines the number of resources of each type currently allocated to each process
- $\text{allocation}[i, j] = k \Rightarrow$ process p_i is currently allocated k instances of resource class R_j
- * need
 - An $n \times m$ matrix
 - Indicates the remaining resource need of each process
 - $\text{need}[i, j] = k \Rightarrow$ process p_i may need k more instances of resource type R_j in order to complete its task
 - $\text{need}[i, j] = \text{maximum}[i, j] - \text{allocation}[i, j]$
- Banker's Algorithm
 - * request_i
 - Request vector for process p_i
 - $\text{request}_i[j] = k \Rightarrow$ process p_i wants k instances of resource class R_j
 - * Upon request for resources, the following actions are taken


```

if requesti > needi then
  raise error condition
else
  if requesti ≤ available then
    { available -= requesti
      allocationi += requesti
      needi -= requesti
    }
  else
    wait (pi)
          
```
 - * If resulting resource-allocation state is safe, transaction is completed and process p_i is allocated its resources
 - * If the new state is unsafe, p_i must wait for request_i and the old allocation state is restored
- Safety Algorithm
 - * Finds out whether or not a system is in a safe state


```

var
  work : integer vector [1..m]
  finish : boolean vector [1..n]

work = available
for ( i = 1; i < n; i++ )
  finish[i] = false;
x :find an i such that
  { finish[i] == false
    needi ≤ work
  }
  if there is no such i then
    { if finish[i] == true for all i then
      system is in a safe state
    }
  else
    { work += allocationi
      finish[i] = true
      go to x
    }
          
```
- Example
 - * System with five processes

	Allocation			Maximum			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	7	5	3	3	3	2
p_1	2	0	0	3	2	2			
p_2	3	0	2	9	0	2			
p_3	2	1	1	2	2	2			
p_4	0	0	2	4	3	3			

* Matrix need

	Need		
	A	B	C
p_0	7	4	3
p_1	1	2	2
p_2	6	0	0
p_3	0	1	1
p_4	4	3	1

* Sequence $\langle p_1, p_3, p_4, p_2, p_0 \rangle$ satisfies the safety criterion

* Let process p_1 request one additional instance of resource class A and two additional instances of resource class C

$$\text{request}_1 = (1, 0, 2)$$

* $\text{request}_1 \leq \text{available}$ is true

* New state

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	7	4	3	2	3	0
p_1	3	0	2	0	2	0			
p_2	3	0	2	6	0	0			
p_3	2	1	1	0	1	1			
p_4	0	0	2	4	3	1			

* Sequence $\langle p_1, p_3, p_4, p_0, p_2 \rangle$ satisfies the safety criterion

* Request for (3, 3, 0) by p_4 cannot be granted