Deadlocks

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.
"Slides 3SH3 '12" - Sanzheng Qiao

Feb. 2021

System Model

System consists of a finite number of resources of different types R_1, R_2, ..,R_m.

Resource types: CPU cycles, memory space, I/O devices.

Each resource type R_i has W_i identical instances - if system has 2 CPUs, then resource type CPU has two instances.

Thread requests an instance of a resource type, the allocation of any instance of the type should satisfy the request.

Each process utilizes a resource as follows:

1. Request the resource
2. Use the resource
3. Release the resource

Kernel-managed Resource

OS checks to make sure that the thread has requested and has been allocated the resource.

A system table - for each resource that is allocated, the table records the thread to which it is allocated.

Queue of waiting threads.

A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set.

Livelock occurs when a thread continuously attempts an action that fails - less common than deadlock.

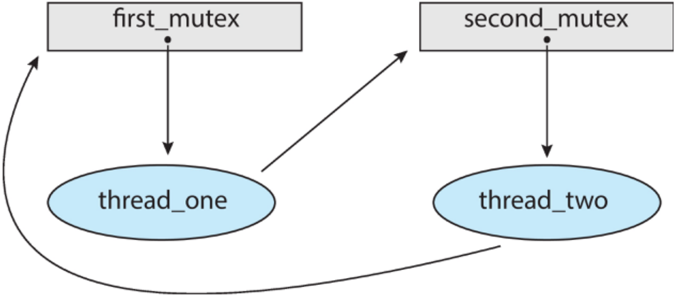
Example: Deadlock in Multithreaded Application

```
/* two mutex locks are created an initialized */
pthread_mutex_t first_mutex;
pthread mutex_t second_mutex;

pthread mutex_init(&first mutex,NULL);
pthread mutex_init(&second mutex,NULL);

/* thread one runs in this function */
void *do_work_one(void *param) {
  pthread_mutex_lock(&first_mutex);
  pthread_mutex_lock(&second_mutex);
  /* Do some work */
  pthread_mutex_unlock(&second_mutex);
  pthread_mutex_unlock(&first_mutex);
  pthread exit(0);
 }

/* thread two runs in this function */
void *do_work_two(void *param) {
  pthread_mutex_lock(&second_mutex);
  pthread_mutex_lock(&first_mutex);
  /* Do some work  */
  pthread_mutex_unlock(&first_mutex);
  pthread_mutex_unlock(&second_mutex);
  pthread exit(0);
}
```

A Resource Allocation Graph

Deadlock is possible if thread 1 acquires `first_mutex` and thread 2 acquires `second_mutex`.

Thread 1 then waits for `second_mutex` and thread 2 waits for `first_mutex`.



Difficult to identify and test deadlocks that may occur only under certain scheduling circumstances.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- Mutual exclusion: only one process at a time can use a resource.
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait: there exists a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2,...,P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

Deadlocks described more precisely by a system resource-allocation graph.

A set of vertices V and a set of edges E.

V is partitioned into two types:

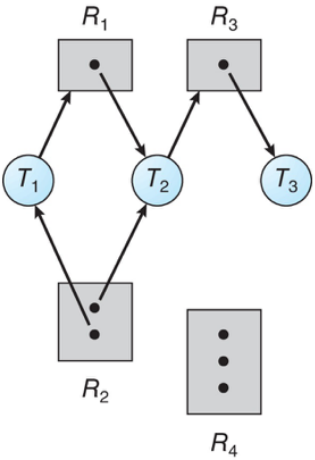
- P = $\{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
- R = $\{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system

request edge - directed edge $P_i \rightarrow R_j$, $i \in \{1, ..n\}$, $j \in \{1, ..m\}$
assignment edge - directed edge $R_j \rightarrow P_i$

Resource Allocation Graph Example

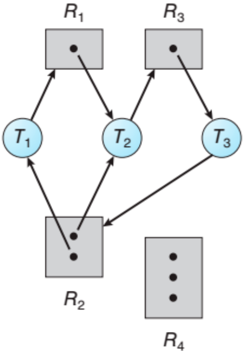
We represent thread T_i as a circle resource type R_i as a rectangle.

- One instance of R_1, two of R_2, one of R_3, three of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 ?
- T_3 ?



$$E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, ...\}$$

Graph With a Deadlock

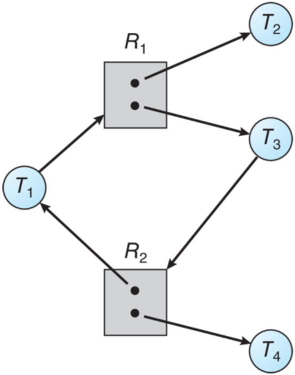


No cycles $\Rightarrow$ no deadlock.

How many cycles exist in the system?

T_1, T_2, T_3 are deadlocked.

Graph With a Cycle But No Deadlock



Cycle: $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

No deadlock! Why?

If there is a cycle, then the system **may** or **may not** be in a deadlocked state.

Methods for Handling Deadlocks

Ensure that the system will never enter a deadlock state:

- Deadlock prevention
- Deadlock avoidance

Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made.

▷ low device utilization

▷ reduced system throughput

Avoiding deadlocks is to require additional information about how resources are to be requested.

Algorithm

What is an algorithm?

Methods for Handling Deadlocks

Ensure that the system will never enter a deadlock state:

- Deadlock prevention
- Deadlock avoidance

Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made.

▷ low device utilization

▷ reduced system throughput

Avoiding deadlocks is to require additional information about how resources are to be requested.

Algorithm

What is an algorithm?
A predetermined set of instructions for solving a specific problem in a limited number of steps.

Deadlock Prevention

Mutual Exclusion - must hold for non-sharable resources.
▷ not required for sharable resources (e.g., read-only files)

Hold and Wait - must guarantee that whenever a process requests a resource, it does not hold any other resources.

- Require process to request and be allocated all its resources before it begins execution.

▷ low resource utilization - resources allocated but not used
▷ starvation possible - waiting to resource forever

Deadlock Prevention (Cont.)

No Preemption - if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.

- Preempted resources are added to the list of resources for which the process is waiting

Circular Wait - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Circular Wait

Invalidating the circular wait condition is most common.

Simply assign each resource $R = \{R_1, R_2, ..., R_n\}$ a unique number, $F : R \to \mathbb{N}$, injective (one-to-one).

Resources must be acquired in order.

If the lock ordering in the Pthread program shown in page 4

`F(first_mutex)=1`

`F(second_mutex)=5`

A thread that wants to use both `first_mutex` and `second_mutex` at the same time must first request `first_mutex` and then `second_mutex`.

Deadlock Avoidance

Requires that the system has some additional **a priori** information available.

▷ In a system with resources R_1 and R_2, the system might need to know that thread P will request first R_1 and then R_2 before releasing both resources, whereas thread Q will request R_2 and then R_1.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

A state is safe if the system can allocate resources to each thread and still avoid a deadlock.

System is in safe state if there exists a sequence $< P_1, P_2, ..., P_n >$ of all the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by:

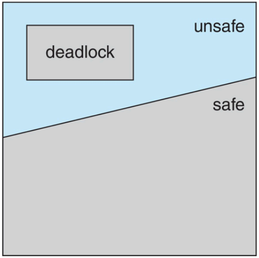
currently available res. + res. held by all the P_j, with $j < i$

Basic Facts

If a system is in safe state $\Rightarrow$ no deadlock

If a system is in unsafe state $\Rightarrow$ possibility of deadlock

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Example

System with twelve resources and three threads.

▷ T_0 - requires ten resources, T_1 - four resources, T_2 - nine resources

Suppose that, at time t_0, thread T_0 is holding five resources, T_1 two resources, T_2 two resources.

	Max. Needs	Current Need
T_0	10	5
T_1	4	2
T_2	9	7

The sequence $<T_1, T_0, T_2>$ satisfies the safety condition - the system in a safe state.

Exercise

Suppose that, at time t_1, thread T_2 requests and is allocated one more resource. Is the system in a safe state?

Avoidance Algorithms

If we have a resource-allocation system with only one instance of each resource type we use

- Resource-allocation graph

For multiple instances of a resource type

- Banker's Algorithm

Resource-Allocation Graph Scheme

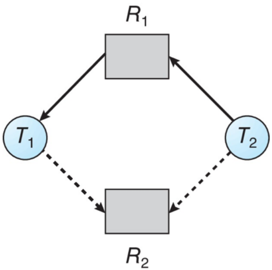
Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j

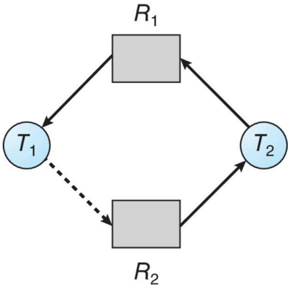
Claim edge converts to request edge when a process requests a resource.

Request edge converted to an assignment edge when the resource is allocated to the process.

Resources must be claimed a priori in the system - before thread T_i starts executing, all its claim edges must already appear in the resource-allocation graph.

Claim edge is represented by a dashed line.





Unsafe State

- ▷ T_2 requests R_2
- ▷ Although R_2 is currently free, we cannot allocate it to T_2, since this action will create a cycle in the graph

Banker's Algorithm

Multiple instances of resources.

Each process must a priori claim maximum use.

When a process requests a resource it may have to wait.

When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

- ▷ *n = number of processes*
- ▷ *m = number of resources* types
  - **Available**: Vector of length m. If *available*$[j] = k$, there are k instances of resource type R_j available
  - **Max**: $n \times m$ matrix. If *Max*$[i,j] = k$, then process P_i may request at most k instances of resource type R_j
  - **Allocation**: $n \times m$ matrix. If *Allocation*$[i,j]$ = k then P_i is currently allocated k instances of R_j
  - **Need**: $n \times m$ matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

Let X, Y be vectors of length n.
$X \leq Y \;\Leftrightarrow\; X[i] \leq Y[i], \quad i = 1, 2, ..., n$

We treat row in the matrices $Allocation_i$ and $Need_i$ as vectors.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:
   $Work = Available$
   $Finish[i] =$ *false* for $i = 0, 1, ..., n - 1$
2. Find an i such that:
   $Finish[i] ==$ *false* $\wedge\ Need_i \leq Work$
   if no such i exists, go to step 4
3. $Work = Work + Allocation_i$
   $Finish[i] =$ *true*
   go to step 2
4. If $Finish[i] ==$ *true* for all i, the system is in a safe state

Resource-Request Algorithm for Process P_i

Determine whether request can be safely granted.

$Request_i[j] = k$, P_i wants k instances of R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   $Available = Available - Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$
   - If safe $\Rightarrow$ the resources are allocated to P_i
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

5 processes P_0 through P_4;
3 resource types: A (10 instances), B (5), and C (7)

Snapshot at time t_0:

	Allocation	Max	Available
	ABC	ABC	ABC
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	

Example of Banker's Algorithm

▷ $Need = Max - Allocation$

	Need
	ABC
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

The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

A state is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.

Example: P_1 Request (1,0,2)

$Request_1 = (1, 0, 2)$

Check $Request_1 \leq Available$ that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ true

	Allocation	Need	Available
	ABC	ABC	ABC
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	

Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

- Detection algorithm - determine whether a deadlock has occurred.
- Recovery scheme - an algorithm to recover from the deadlock

Single Instance of Each Resource Type

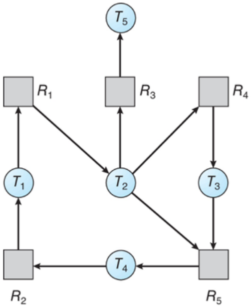
Maintain wait-for graph

- Nodes are processes
- $P_i \rightarrow P_j$ if P_i is waiting for P_j

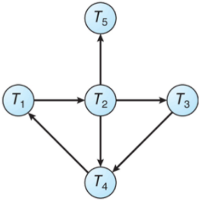
Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

An algorithm to detect a cycle in a graph requires an order of $O(n^2)$ operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-allocation graph



Corresponding wait-for graph

We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

Data structures:

- **Available**: A vector of length m indicates the number of available resources of each type.
- **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An $n \times m$ matrix indicates the current request of each process. If *Request* $[i][j] = k$, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively.
   (a) $Work = Available$
   (b) *for* $i = 1, 2, ..., n$ if $Allocation_i \neq 0$, then
   $Finish[i] = false$ otherwise $Finish[i] = true$
2. Find an index i such that both
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$

If no such i exists, go to step 4 (next page)

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2
4. If $Finish[i] == false$, for some i, $i \leq n$, then the system is in deadlock state.
   Moreover, if $Finish[i] == false$, then P_i is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

5 processes: P_0 through P_4;
3 resource types: A(10 instances), B(2), and C(7)

Snapshot at time T_0:

	Allocation	Request	Available
	ABC	ABC	ABC
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	

Sequence $< P_0, P_2, P_3, P_1, P_4 >$ will result in $Finish[i] =$ *true* for all i.

The system is not in a deadlocked state.

Example (Cont.)

P_2 requests an additional instance of type C

	Request
	ABC
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

System can reclaim resources held by process P_0, but insufficient resources to fulfill other processes.

Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

When, and how often, to invoke depends on:

- How often a deadlock is likely to occur?
- How many processes will be affected by deadlock when it happens?

We can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
▷ Considerable overhead in computation time, but we can identify process that "caused" the deadlock

Invoked arbitrarily (i.e. once per hour)
▷ There may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

Abort all deadlocked processes or abort one process at a time until the deadlock cycle is eliminated?

In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch

Recovery from Deadlock: Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Selecting a victim - which resources and which processes are to be preempted to minimize cost.

Rollback - return to some safe state, restart process for that state.

Starvation - same process may always be picked as victim, include number of rollback in cost factor.

Thank you !

Operating Systems are among the

most complex pieces of software ever developed !