Operating Systems 2023 Spring Term
Week 8

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Deadlock

April 27, 2023

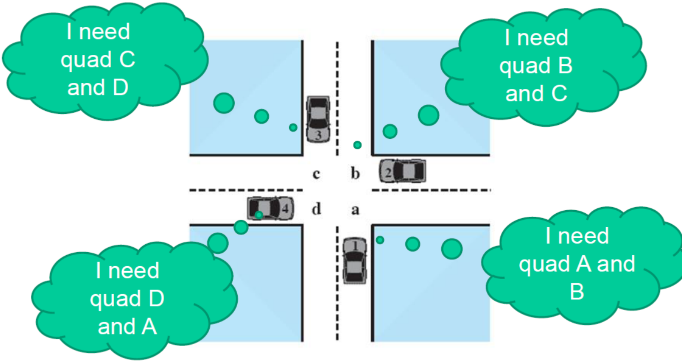
Week 6: Sample Glossary

- **deadlock:** The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads. (on Page 1245)
- **dining-philosophers problem:** A classic synchronisation problem in which multiple operators (philosophers) try to access multiple items (chopsticks) simultaneously. (on Page 1246)
- **readers-writers problem:** A synchronisation problem in which one or more processes or threads write data while others only read data (on Page 1265)

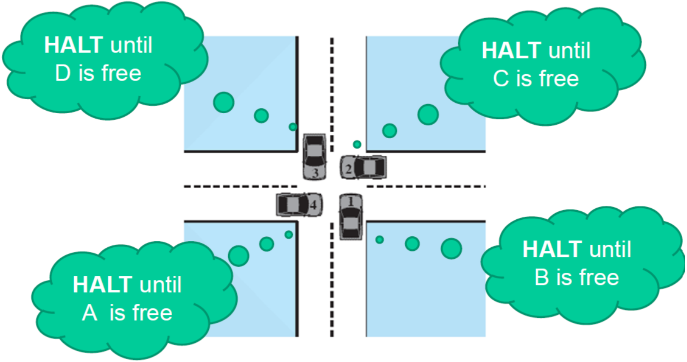
Race Conditions and how to prevent them:
https://youtu.be/MqnpIwN7dz0 (thanks to Muhammed Efe İncir)

Potential Deadlock

The typical rule of the road in the United States is that a car at a four way stop should defer to a car immediately to its right. This rule works if there are only two or three cars at the intersection (similar to the Dining-Philosophers Problem?).

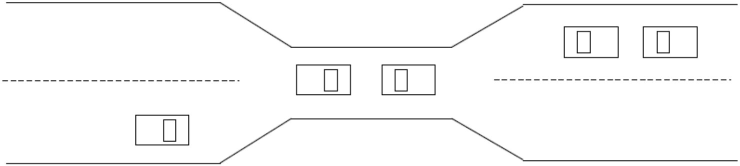


Actual Deadlock



Bridge Crossing Example

Traffic only in one direction.
Each section of a bridge can be viewed as a resource.
If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
Several cars may have to be backed up if a deadlock occurs.
if many times the same side (or processes) preemts resources and rollback; in that case the starvation is possible.



Resource Categories

Two general categories of resources:

- Reusable -> can be safely used by only one process at a time and is not depleted by that use.
- Consumable -> one that can be created (produced) and destroyed (consumed).

Reusable Resources

Such as:

- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

"In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads." -> deadlock -> a form of liveness failure

System Model Definitions

- System consists of resources
- Resource types R_1, R_2, . . ., R_m
- CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  request
  use
  release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

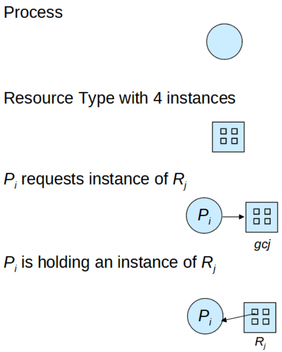
- Mutual exclusion: only one process at a time can use a resource
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has 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 I

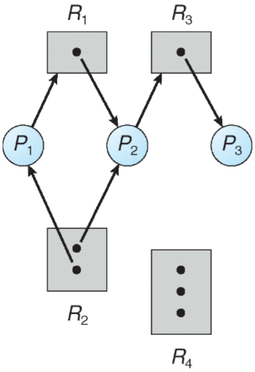
A set of vertices V and a set of edges E.

- 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 -> R_j
- assignment edge – directed edge R_j -> P_i

Resource-Allocation Graph II

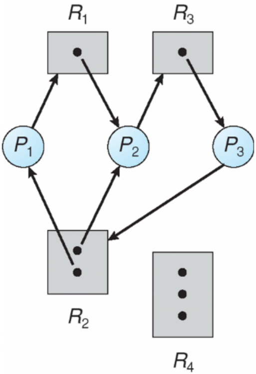


Resource-Allocation Graph: Example



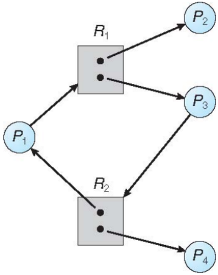
Thread T1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
Thread T2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
Thread T3 is holding an instance of R3.

Resource-Allocation Graph: Deadlock



Threads T1, T2, and T3 are deadlocked. Thread T2 is waiting for the resource R3, which is held by thread T3. Thread T3 is waiting for either thread T1 or thread T2 to release resource R2. In addition, thread T1 is waiting for thread T2 to release resource R1.

Resource-Allocation Graph: Cycle But No Deadlock



However, there is no deadlock. Observe that thread T4 may release its instance of resource type R2. That resource can then be allocated to T3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

Basic Facts

- If graph contains no cycles -> no deadlock
- If graph contains a cycle ->
  if only one instance per resource type, then deadlock
  if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  Deadlock prevention
  Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention I

Restrain the ways request can be made

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- 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, or allow process to request resources only when the process has none allocated to it. Low resource utilisation; starvation possible

Deadlock Prevention II

- 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
  Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- Circular Wait -> impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Example

Thread one attempts to acquire the mutex locks in the order (1) first mutex, (2) second mutex. At the same time, thread two attempts to acquire the mutex locks in the order (1) second mutex, (2) first mutex. Deadlock is possible if thread one acquires first mutex while thread two acquires second mutex.

```
/* 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);
}
```

Deadlock Example with Lock Ordering

Transactions 1 and 2 execute concurrently. Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Deadlock Avoidance

A method for handling deadlocks
Requires that the system has some additional a **priori** information available

- Simplest and most useful model requires that each process declare the **maximum number of resources** of each type that it may need
- 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

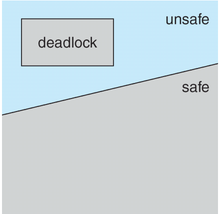
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <P_1, P_2, ..., P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j < I
- That is:
  If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State: Basic Facts

Concurrency arises in:

- If a system is in safe state -> no deadlocks
- If a system is in unsafe state -> possibility of deadlock
- **Avoidance -> ensure that a system will never enter an unsafe state.**

Safe, Unsafe, Deadlock State



A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent threads from requesting resources in such away that a deadlock occurs. The behavior of the threads controls unsafe states

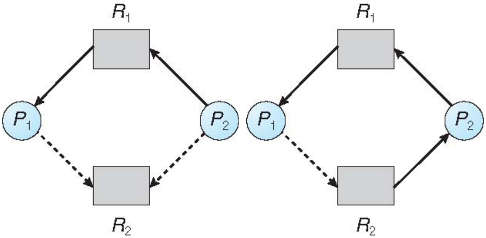
Avoidance Algorithms

- Single instance of a resource type
  Use a resource-allocation graph
- Multiple instances of a resource type
  Use the banker's algorithm

Resource-Allocation Graph Scheme

- Claim edge Pi -> Rj indicated that process Pj may request resource Rj; represented by a dashed line
- 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
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Resource Allocation Graph For Deadlock Avoidance - Unsafe State



Suppose that process Pi requests a resource Rj
The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph (This algorithm is not applicable to a resource allocation system with multiple instances of each resource type.)

Banker's Algorithm

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

- Multiple instances
- Each process must a priori claim maximum number of instances of each resource type that it may need
- This number may not exceed the total number of resources in the system.
- Otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm: Data Structures

Let n = number of processes, and m = number of resources types.

- Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available
- Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
- Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
- Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task
  Need [i,j] = Max[i,j] – Allocation [i,j]

Banker's Algorithm: Example I

5 processes P0 through P4;
3 resource types:
A (10 instances), B (5instances), and C (7 instances)
Snapshot at time T0:

	Allocation	*Max*	*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	

Banker's Algorithm: Example II

The content of the matrix Need is defined to be Max – Allocation
The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

	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

Banker's Algorithm: Illustration I

	Allocation			Max.			Need			Available
	A	B	C	A	B	C	A	B	C	
P0	0	1	0	7	5	3	7	4	3	3 3 2
P1	2	0	0	3	2	2	1	2	2	
P2	3	0	2	9	0	2	6	0	0	
P3	2	1	1	2	2	2	0	1	1	
P4	0	0	2	4	3	3	4	3	1	

	Allocation			Max.			Need			Available
	A	B	C	A	B	C	A	B	C	
P0	0	1	0	7	5	3	7	4	3	2 1 0
P1	3	2	2	3	2	2	0	0	0	
P2	3	0	2	9	0	2	6	0	0	
P3	2	1	1	2	2	2	0	1	1	
P4	0	0	2	4	3	3	4	3	1	

5 3 2

	Allocation			Max.			Need			Available
	A	B	C	A	B	C	A	B	C	
P0	0	1	0	7	5	3	7	4	3	5 2 1
P1	0	0	0	3	2	2	0	0	0	
P2	3	0	2	9	0	2	6	0	0	
P3	2	2	2	2	2	2	0	0	0	
P4	0	0	2	4	3	3	4	3	1	

7 4 3

	Allocation			Max.			Need			Availab le
	A	B	C	A	B	C	A	B	C	
P0	0	1	0	7	5	3	7	4	3	3 1 2
P1	0	0	0	3	2	2	0	0	0	
P2	3	0	2	9	0	2	6	0	0	
P3	0	0	0	2	2	2	0	0	0	
P4	4	3	3	4	3	3	0	0	0	

7 4 5

Banker's Algorithm: Illustration II

	Allocation			Max.			Need			Available
	A	B	C	A	B	C	A	B	C	
P0	7	5	3	7	5	3	0	0	0	0 0 2
P1	0	0	0	3	2	2	0	0	0	
P2	3	0	2	9	0	2	6	0	0	
P3	0	0	0	2	2	2	0	0	0	
P4	4	3	3	4	3	3	0	0	0	

7 5 5

	Allocation			Max.			Need			Availab le
	A	B	C	A	B	C	A	B	C	
P0	0	0	0	7	5	3	0	0	0	1 5 5
P1	0	0	0	3	2	2	0	0	0	
P2	9	0	2	9	0	2	0	0	0	
P3	0	0	0	2	2	2	0	0	0	
P4	4	3	3	4	3	3	0	0	0	

10 5 7

The system is in a safe state since the sequence <P1, P3, P4, P0, P2> satisfies safety criteria.

Banker's Algorithm: Example P1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ *true*.

	Allocation	*Need*	*Max*	*Available*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 1 0	7 4 3	7 5 3	2 3 1
P_1	3 0 2	0 2 0	3 2 2	
P_2	3 0 1	6 0 0	9 0 1	
P_3	2 1 1	0 1 1	2 2 2	
P_4	0 0 2	4 3 1	4 3 3	

$Need_i$ <= $Available_i$

Available

2 3 1

P1-> (2 3 1)-(0 2 0) = (2 1 1)
(2 1 1) + (3 2 2) = (5 3 3)
P3 -> (5 3 3)-(0 1 1)=(5 2 2)
(5 2 2)+(2 2 2)=(7 4 4)
P4 -> (7 4 4)-(4 3 1)=(3 1 3)
(3 1 3)+(4 3 3)=(7 4 6)
P0-> (7 4 6)-(7 4 3)=(0 0 3)
(0 0 3)+(7 5 3)=(7 5 6)
P2 -> (7 5 6)-(6 0 0)=(1 5 6)
(1 5 6)+(9 0 2)=(10 5 8)

- 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?
- Can request for (0,2,0) by P_0 be granted?

Banker's Algorithm: Example P0 Request (0,2,0)

- Check that Request $\leq$ Available (that is, $(0,2,0) \leq (2,3,1) \Rightarrow$ *true*.

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 2 0	7 5 3	7 3 3	2 2 1
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 1	9 0 1	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- Can request for (0,2,0) by P_0 be granted?

3 resource types ;
***A* (10 instances), B (5 instances), and *C* (7 instances).**

$Need_i \leq Available_i$

Available

2 2 1

P1 -> 5 2 3

P3 -> 7 3 4

P4 -> 7 3 6

P0 -> 7 5 6

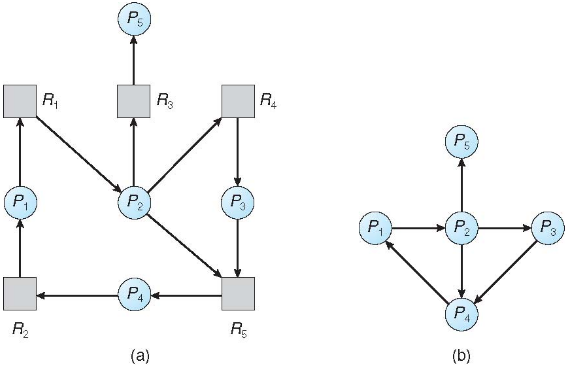
P2 -> 10 5 7

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single instance of each resource type -> maintain wait for graph
Nodes are processes.
Pi -> Pj if Pi is waiting for Pj
Periodically invoke an algorithm that searches for a cycle in the graph.

Resource-Allocation Graph and Wait-for Graph



(a) Resource-Allocation Graph (b) Corresponding wait-for graph

Several Instances of a Resource Type

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

Example of Detection Algorithm I

Five processes P_0 through P_4; three resource types
A (7 instances), B (2 instances), and C (6 instances).
Snapshot at time T_0:

Allocation	*Request*	*Available*
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	

$Need_i$ <= $Available_i$

Available

0 0 0
P0 -> 0 1 0
P2 -> 3 1 3
P3 -> 5 2 4
P4 -> 5 2 6
P1 -> 7 2 6

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

Example of Detection Algorithm II

P2 requests an additional instance of type C

- P_2 requests an additional instance of type C.

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 1	2 0 1	
P_2	3 0 2	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

State of system?

$Need_i$ <= $Available_i$

Available

0 0 0

P0 -> 0 1 0

Deadlock!

- Can reclaim resources held by process *P*0, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

If detection algorithm is invoked arbitrarily, 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
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  Priority of the process
  How long process has computed, and how much longer to completion
  Resources the process has used
  Resources process needs to complete
  How many processes will need to be terminated
  Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim -> minimise 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.