Lecture.7.Deadlock.txt

- System Model
    - A system consists of a finite number of resources of different types R_1, R_2, ..., R_m
        - Resource types can be: CPU cycles, memory space, I/O devices, etc.
            - There can be a number of instances of the same resource
                - i.e. 5 I/O devices, 2 CPUs, etc.
        - Each resource type R_i has W_i identical instances
            - i.e. If a system has 2 CPUs, then resource type CPU has two instances
    - When the system allocates a resource, it allocates one instance of the resource
        - The instance (W_i) of the resource is not relevant
            - The output should be the same, regardless of the instance used
            - i.e. It doesn't matter which CPU does the work, because the calculation is the same
    - When a thread requests an instance of a resource type, the allocation of any instance of the type should satisfy the request
        - This means that each instance of particular type should be the same
    - Each process utilizes a resource as follows:
        1. Request the resource
        2. Use the resource
        3. Release the resource
            - This is important because other threads may need the resource
    - One of the major tasks of an operating system is to manage resources

- Kernel Managed Resource
    - The operating system checks to make sure that the thread has requested, and has been allocated the correct resource
        - Allocating resources to threads/processes is (one of) the job of the operating system
    - A system table is used to keep track of resources and where they are allocated
        - For each resource that is allocated, the table records the thread to which it is allocated
    - If multiple threads need the same resource, then a waiting queue is used
        - This is called queue of waiting threads
        - Managing this is (one of) the job of the operating system
    - A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused by only another thread in the set
        - Another state the system may end up in is Livelock

        - Livelock occurs when a thread continuously attempts an action that fails. Thus, the lock is never released
            - i.e. Infinite loop
            - This is less common than deadlock
    - In a deadlocked system, there is no progress or execution of instructions
        - In a livelock, the system executes instructions, but makes no meaningful progress
            - i.e. Flow remains in the same block of code

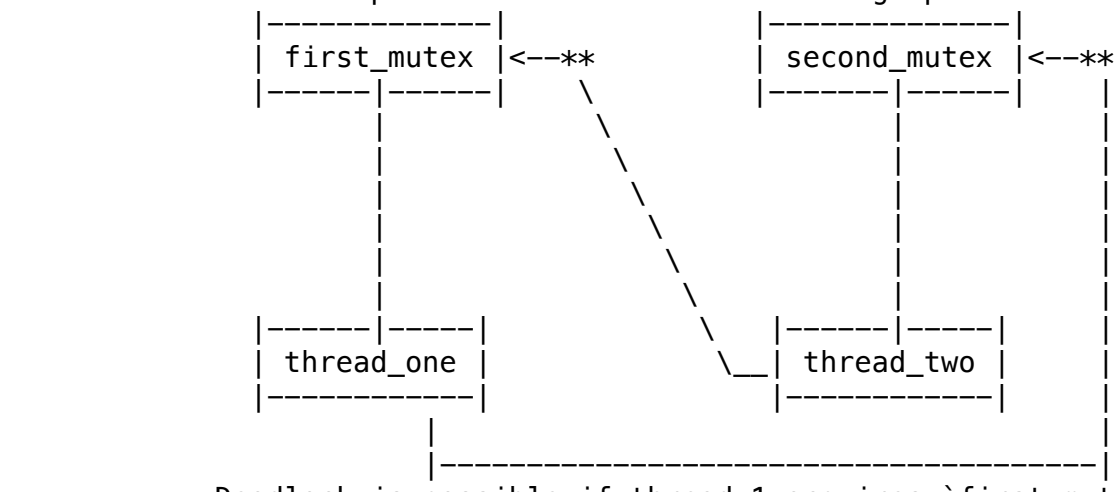
- Deadlock In Multithreaded Application
    - i.e. Example code

```
// Two mutex locks are created and 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);
}
```

    - If thread one acquires `first_mutex`, and thread two acquires `second_mutex`, then the system is deadlocked
        - Because thread one wants the `second_mutex`, but thread two has it. And, thread two wants the `first_mutex`, but thread one has it. Both threads are waiting on each other to release the lock, but none will release it
        - Note: This may happen, but it depends on the order of execution which is not guaranteed. So, sometimes the

program will work fine, and other times it may dead-
lock. You may not notice this error until (much) later

- A Resource Allocation Graph
  - i.e. Example of a resource allocation graph



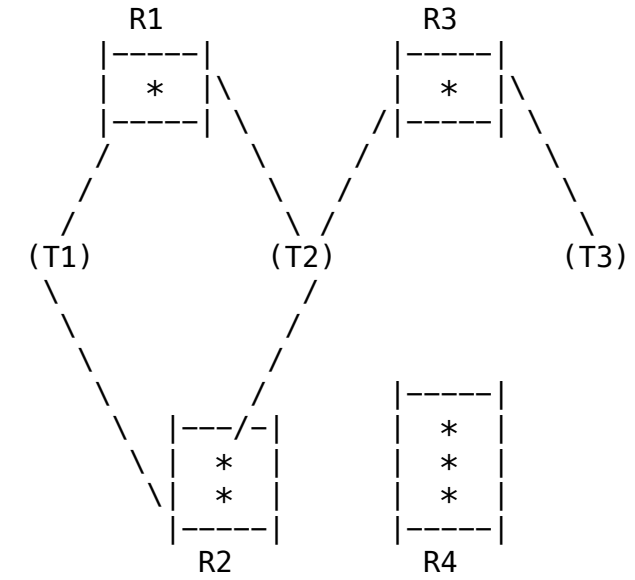
  - Deadlock is possible if thread 1 acquires `first_mutex`, and thread 2 acquires `second_mutex`
    - Then, thread 1 waits for `second_mutex`, and thread 2 waits for the `first_mutex`
      - The system is now deadlocked
  - It is difficult to identify and test deadlocks that may occur only under certain scheduling circumstances

- Deadlock Characterization
  - Deadlock can arise if four conditions hold simultaneously:
    - Mutual exclusion
      - Only one process at a time can use a resource
        - However, if the memory segment is read only, then multiple threads can read it at the same time
    - Hold and wait
      - A process holding at least once 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
        - If there is preemption, and the resource can be released, then deadlock will not happen
    - Circulat wait
      - There exists a set {P0, P1, ..., Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0
        - If one of these conditions are broken, then deadlock will not happen
  - This is a general approach

    - If one of these four conditions does not hold, then deadlock will not happen
    - If none of the four conditions are satisfied, then there will be no deadlock

- Resource Allocation Graph
    - Deadlocks can be more precisely described by a system resource allocation graph
        - Contains a set of vertices, 'V', and a set of edges, 'E'
            - This visual representation helps us intuitively understand where the problem may occur
            - 'V' is partitioned into two types; we can have two types of vertices:
                - P = {P1, P2, ..., P_n}
                    - The set consisting of all the processes in the system
                - R = {R1, R2, ..., R_m}
                    - The set consisting of all the resource types in the system
            - There are two types of edges:
                - Request edge
                    - Directed edge: P_i -> R_j, i IN {1, ..., n}, j IN {1, ..., m}
                - Assignment edge
                    - Directed edge: R_j -> P_i

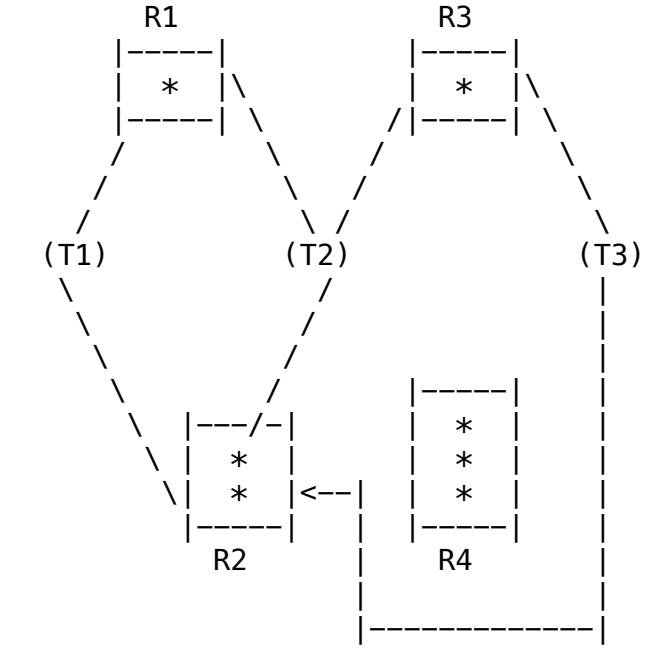
- Resource Allocation Graph Example
    - Threads (T_i) are represented as a circle
    - Resources (R_j) are represented as a rectangle
    - i.e. Graph example



        - The graph has:
            - 1 instance of R1

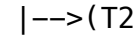
    - 2 instances of R2
    - 1 instance of R3
    - 3 instances of R4
  - T1 holds 1 instance of R2, and is waiting for an instance of R1
    - In other to progress, T1 needs an instance of R1, but it is assigned to T2
  - T2 holds 1 instance of R1, and 1 instance of R2
  - T3 holds 1 instance of R3
  - E = {T1 -> R1, T2 -> R3, R1 -> T2, ...}
  - The system is not deadlocked because T3 has R3, and eventually it will release R3. T2 will be able to acquire R3 and finish its operation(s). Upon completion, T2 will release R1, R2, and R3. Finally, T1 will be able to acquire R1, and finish its job. Thus, the system is not deadlocked

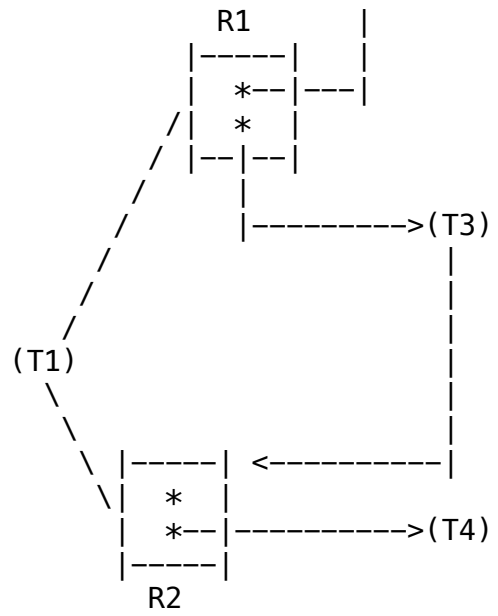
- Graph With A Deadlock
  - i.e. Previous example with deadlock



  - In this example, T3 is waiting for both resources from R2, which are assigned to T1 and T2. Since T1 and T2 are waiting for other resources that are assigned to other threads, the system is deadlocked
    - T1, T2, and T3 are deadlocked
  - There are 3 cycles in the example above
    1. T1 -> T2 -> T1
    2. T2 -> T3 -> T2
    3. T1 -> T2 -> T3 -> T1

- Graph With A Cycle But No Deadlock
  - i.e. Example graph





        - In this graph:
            - T1 has 1 instance of R2
            - T1 wants an instance of R1
            - T2 has 1 instance of R1
            - T3 has 1 instance of R1
            - T3 wants an instance of R2
            - T4 has 1 instance of R2
        - Even though there is a cycle in this example, there is no deadlock
            - The cycle is between T1 and T3
            - T2 and T4 do not require additional resources
                - Upon completion, they will release the resources that they hold; R1 and R2, respectively
            - When T2 and T4 release their resources, T1 and T3 will be able to acquire them, and finish their task
        - If there is a cycle, then the system may or may not be in a deadlocked state

- Methods For Handling Deadlocks
    - To ensure that the system will never enter a deadlock state, we can use:
        - Deadlock prevention
        - Deadlock avoidance
    - Depending on how many resources are used in a system, a different algorithm may be used
    - Deadlock prevention algorithms prevent deadlocks by limiting how requests can be made
        - i.e. Low device utilization
        - i.e. Reduced system throughput
    - Deadlock avoidance requires additional information about how resources are to be requested
        - Information is required in-advance to be able to design an

      algorithm that will help us avoid deadlocks
    - Algorithm Definition
      - An algorithm is a predetermined set of instructions for solving a specific problem in a limited number of steps

- Deadlock Prevention
    - There are 4 conditions that must hold in order for the system to be deadlocked
      - In order to avoid deadlock, 1 of the 4 conditions needs to be dealt with
      - If one of the conditions are broken, then deadlock will not happen
    - Mutual Exclusion
      - Must hold for non-shareable resources
      - Not required for shareable resources
        - i.e. Read-only files
      - If there is no mutual exclusion, then there is no deadlock
    - Hold and Wait
      - Must guarantee that whenever a process requests a resource, it does not hold any other resource(s)
        - 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 indefinitely for a resource
    - 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
        - i.e. If P1 holds R1 and R2, and needs R3 to continue, but does not get R3, then P1 must release R1 and R2
    - Circular Wait
      - Impose a total ordering of all resource types, and require that each process request resources in an increasing order of enumeration
        - i.e. Mutexes need to be acquired in the same order for all threads

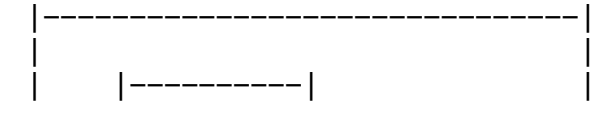
- Circular Wait
    - Invalidating the circular wait condition is the most common way to impose a total ordering of all resource types
      - i.e. Simply assign each resource, 'R = {R1, R2,..., Rn} a unique number, 'F'
        - F : R -> |N, injective (one-to-one)
      - Resources must be acquired in order
        - If multiple threads want to use both `first_mutex` and `second_mutex` at the same time, then it must request/acquire `first_mutex` and then `second_mutex`
          - This is the solution to the example code, "Deadlock

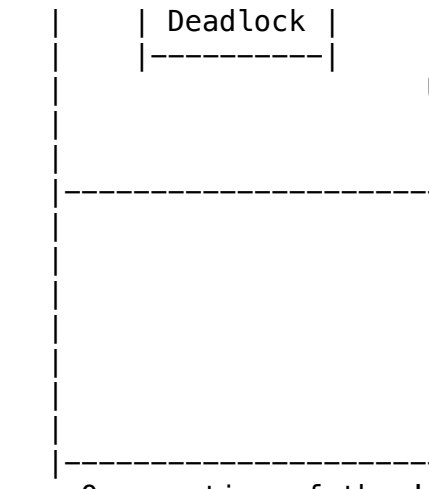
In Multithreaded Application"
  - The lock ordering for this would be:
    - F(first_mutex) = 1
    - F(second_mutex) = 5

- Deadlock Avoidance
  - One of the ways to avoid deadlock is to impose a total ordering on all resource types
    - This solves the circular wait issue
  - Requires that the system has some additional 'a priori' information available
    - We need information in-advance in order to avoid deadlock before it happens
      - i.e. In a system with resources 'R1' and 'R2', the system might need to know that thread 'P' will first request 'R1' and then 'R2' before releasing both resources, whereas thread 'Q' will request 'R2' and then 'R1'
    - If resource allocation requirements are known in advance, then 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 on the processes

- Safe State
  - Safe states are deadlock free
  - A state is safe if the system can allocate resources to each thread and still avoid deadlock
  - The system is in a safe state if there exists a sequence < P1, P2, ..., Pn > of all processes in the system 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', where 'j' < 'i'
    - The process is in a safe state if the required resources for a process is the sum of currently available resources and resources that it already holds

- Basic Facts
  - If a system is in a safe state   -> No deadlock
  - If a system is in an unsafe state -> Possibility of deadlock
    - Being in an unsafe state does not guarantee deadlock
  - Deadlock avoidance ensures that a system will never enter an unsafe state
  - i.e. Diagram of states





- One portion of the 'unsafe' state can cause deadlock
- A system should never, or avoid, entering an 'unsafe' state
  - 'Unsafe' states can lead to deadlock
- If the system is in a 'safe' state, then there's no deadlock
- Deadlock avoidance algorithms are designed in a way to ensure that the system will never enter an 'unsafe' state

- Unsafe code can be converted into safe code
  - This is the job of the algorithm

- Example
  - Synopsis
    - Assume that a system has 12 resources and 3 threads
      - Thread #0 (T0) requires 10 resources
      - Thread #1 (T1) requires 4 resources
      - Thread #2 (T2) requires 9 resources
    - Suppose that T0 is holding 5 resources, T1 is holding 2 resources, and T2 is holding 2 resources. The distribution of resources is summed below:

	Max. Needs	Holding	Current Need
T0	10	5	5
T1	4	2	2
T2	9	2	7

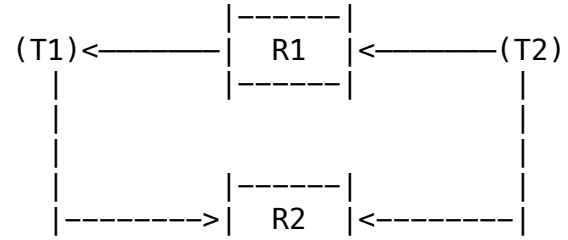
    - The sequence < T1, T0, T2 > satisfies the safety condition
      - The system is in a safe state
      - In order to derive this sequence, 'a priori' knowledge is required
      - Only T1 can be executed first, because there are only 3 resources available, and T1 needs 2; the other threads need more resources than what is available, thus they cannot be executed before T1
        - After T1 finishes, 5 resources are available, which

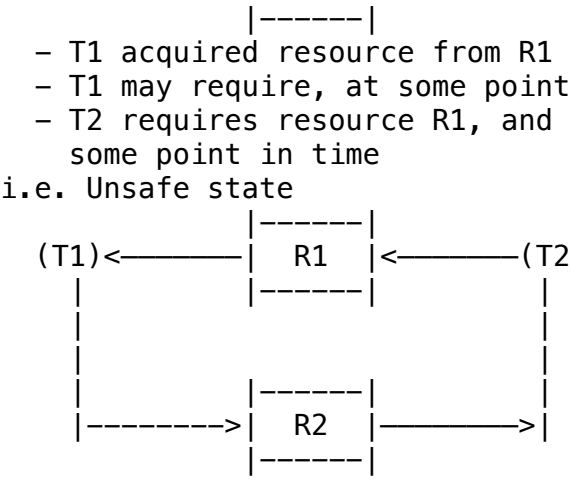
is exactly what T0 requires. After T0 finishes, it will free up enough resources for T2 to proceed
    - Question
        - Suppose that, at time t1, thread T2 requests and is allocated one more resource. Is the system in a safe state?
    - Answer
        - No, the system is not in a safe state. This is because the sequence < T1, T0, T2 > won't be satisfied due to insufficient resources. T1 will be able to acquire 2 resources and finish, but T0 will not be able to acquire 5 resources, because the system will only have 4 resources to offer

- 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, we use:
            - Banker's Algorithm
    - It is easier to design an avoidance algorithm for one instance per resource
        - Designing an avoidance algorithm for multiple instances of a resource type is much more complex

- Resource Allocation Graph Scheme
    - In the resource allocation graph, another edge is added
        - In addition to request and assignment edge, a claim edge is added
            - The claim edge indicates that process 'P_i' may request resource 'R_j'
                - i.e. 'P_i' -> 'R_j'
            - Claim edge converts to request edge when a process requests a resource
                - The request edge is converted to an assignment edge when the resource is allocated to the process
    - Resources mst be claimed 'a priori' in the system; before thread 'T_i' starts executing
        - All of its claim edges must already appear in the resource allocation graph

- Resource Allocation Graph
    - A claim edge is represented by a dashed line
        - Solid lines represent acquired resource
    - i.e. Claim edge graph example





- T1 acquired resource from R1
- T1 may require, at some point in time, an instance from R2
- T2 requires resource R1, and may require resource R2 at some point in time

- i.e. Unsafe state

- Although R2 is currently free it cannot be allocated to T2, since this action will create a cycle in the graph
  - This is unsafe (state)

- Relatively simple and only works if there is one instance per resource

- Banker's Algorithm
  - Used if there are multiple instances of resources
  - Each process must 'a priori' claim maximum use
    - Before a system starts, we must know the maximum number of resources a process will need
  - When a process requests a resource it may have to wait
  - Is there an execution sequence of processes, where all processes will eventually get the resources that they need to finish
    - When a process gets all its resources it must return them in a finite amount of time
      - If a process does not return a resource, then the system won't progress
  - This safety algorithm is relatively simple

- Data Structures For The Banker's Algorithm
  - In the Banker's algorithm, there are:
    - 2 variables, 'n' and 'm':
      - n = number of processes
      - m = number of resource types
        - Example of resource type: CPU, Memory, I/O, Etc.
    - 1 Vector:
      - Available
        - Vector of length 'n'
        - If `available[j] == k`, there are 'k' instances of resource type 'R_j' available
    - 3 Matrices:
      - Max
        - Is an `n x m` matrix
        - If `Max[i,j] = k`, then process 'P_i' may request at most 'k' instance of resource type 'R_j'

        - Allocation
            - Is an `n x m` matrix
            - If `Allocation[i,j] = k`, then 'P_i' is currently allocated 'k' instances of 'R_j'
        - Need
            - Is an `n x m` matrix
            - If `Need[i,j] = k`, then 'P_i' may need 'k' more instances of 'R_j' to complete its task
    - The correlation between the 3 matrices is:
      Need[i,j] = Max[i,j] - Allocation[i,j]

- Safety Algorithm
    - Let X, Y be vectors of length 'n'
        - X <= Y <--> X[i] <= Y[i], i = 1, 2, ..., n
            - The 'i' element in vector 'X' is less than or equal to the 'i' element in vector 'Y'
            - Vector 'X' is less than or equal to Vector 'Y'
        - The rows in the 'Allocation[i]' and 'Need[i]' matrices are treated as vectors
    - Steps to determine if the system is in safe state:
        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 AND 'Need[i]' <= 'Work'
              If no such 'i' exists, go to step 4
        3. Perform the following calculations and go to step 2:
            Work = Work + Allocation
            Finish[i] = true
        4. If `finish[i] == true` for all 'i', the system is in a safe state

- Resource Request Algorithm For Process P_i
    - If there is a request at sometime, we need to determine whether that request can be safely granted
        - Will the system continue to be in a safe state or go to an unsafe state?
    - Assume we have:
      `Request_i[j] = k`, 'P_i' wants 'k' instances of 'R_j'
    - Steps to determine whether the request can be safely granted:
        1. If `Request_i` <= `Need_i`, go to step 2. Otherwise, raise error condition since process has exceeded its maximum claim
        2. If `Request_i` <= `Available`, go to step 3. Otherwise, 'P_i' must wait since resources are not available
            - The number of available resource is changing overtime. As other processes finish, they release their resources, allowing other processes to acquire them and finish
                - The sooner a process releases its resources, the

better
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;
- The system is 'safe' if the resources are allocated to 'P_i'
- Otherwise, the system is 'unsafe' and 'P_i' must wait, and the old resource-allocation state is restored

- Example Of Banker's Algorithm
  - Assume there is a system with 5 processes; P0 to P4. There are 3 resource types: A, B, and C. Resource A has 10 instances, resource B has 5 instances, and resource C has 7 instances. At some time, t0, the state of the system is summarized in the table below:

	Allocation			Maximum			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

    - P0 has been allocated 0 instances of Resource 'A', 1 instance of Resource 'B', and 0 instances of Resource 'C'
      - It needs 7 instances of 'A', 5 instances of 'B', and 3 instances of 'C'
    - P1 has 2 instances of 'A', but no instances of 'B' and 'C'
      - It needs 3 instances of 'A' and 2 instances of 'B' and 'C'
  - The Need matrix is calculated by: `Need = Max - Allocation`. The results are summed in the table below:

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1

|----|-------|
P4	4 3 1

  - The system is in a safe state since the sequence < P1, P3, P4, P2, P0 > satisfies the safety criteria
    - This sequence is the order of execution for the processes. P1 executes first, then P3, and finally P0
      - The needs of P1 < 1, 2, 2 > can be satisfied with the available resources < 3, 2, 2 >
      - Once a process completes, it releases the resources it is holding onto
  - A thread is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock

- Example: P1 Requests (1,0,2)
  - Question:
    - Based on the example above, if P1 requests (1,0,2), can this request be satisfied? Let `Request_1 = (1,0,2)` represent P1 requesting (1,0,2)
  - Answer:
    - Check 'Request_1' <= 'Available'
      - 'Request_1' = (1,0,2) AND 'Available' = (3,3,2)
        - (1,0,2) <= (3,3,2)
        - This request can be satisfied
    - Therefore, the sequence < P1, P3, P4, P0, P2 > can still be safely satisfied, based on the safety algorithm
  - Question:
    - Can request for (3,3,0) by P4 be granted?
  - Answer:
    - The request for P4 cannot be granted if P4 is the first process to execute in the sequence. However, if P4 is the 3rd process to execute, followed by P1 and P3, then the request can be satisfied and the system will continue to be in a safe state
  - Note: The sequence specified above does not factor in priority; all processes are assumed to have the same priority

- Deadlock Detection
  - If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur
  - A detection algorithm determines whether deadlock has occurred or not
    - i.e. If the system is behaving in an unexpected manner, then the algoritm may kick in
      OR
      The algorithm is executed in specific intervals
    - There are two types of detection algorithms
      1. Resources that only have 1 instance

        2. Resources with multiple instances
    - A recovery scheme is an algorithm used to recover from deadlock
        - i.e. If system is deadlocked, then slowly release locks for processes until the system makes progress

- Single Instance Of Each Resource Type
    - This is called a 'wait-for' graph
        - In this graph, nodes are processes
            - 'P_i' -> 'P_j' IF 'P_i' is waiting for 'P_j'
        - An algorithm periodically searches for a cycle in the graph
            - If there is a cycle, then a deadlock exists
        - 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
            - This algorithm is computationally expensive, because it is exponentially increasing
                - Adding more processes to the graph requires exponential more time to search for
            - Note: This is for a single instance of resource(s)

- Resource Allocation Graph & Wait-For Graph
    - A 'wait-for' graph from the resource allocation graph is obtained by removing the resource nodes and collapsing the appropriate edges
        - If there is a cycle in the 'wait-for' graph, then there is a deadlock state
    - The 'wait-for' graph does not work if there are multiple resources and multiple instances for the resources

- 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
        - Another data structure, and an algorithm to iterate it, is needed to handle this
            - The data structure contains:
                - Available
                    - A vector of length 'm' that indicates the number of available resources of each type
                - Allocation
                    - An 'n x m' matrix that defines the number of resources of each type currently allocated to each process
                - Request
                    - An 'n x m' matrix that indicates the current request of each process
                        - If `Request[i][j] = k`, then the process 'P_i' is requesting 'k' more instances of resource type 'R_j'

- Detection Algorithm

- There are four steps to the detection algorithm:
    1. Let 'Work' and 'Finish' be vectors of length 'm' and 'n', respectively. Compute:
        (a) Work = Available
        (b) for i = 1, 2, ..., n IF allocation_i =/= 0, then
            Finish[i] = false, otherwise Finish[i] = true
    2. Find an index 'i' such that both:
        (a) Finish[i] == false
        (b) Request_i <= Work
       If no such 'i' exists, go to step 4
    3. Compute:
        Work = Work + Allocation_i;
        Finish[i] = true
       Go to step 2
    4. If 'Finish[i] == false', for some 'i', where 'i <= n', then the system is in a deadlocked state.
       Moreover, if Finish[i] == false, then 'P_i' is deadlocked
- Requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state
    - Where:
        - m = number of resources
            - Increases at a linear rate
        - n = number of processes
            - Increases exponentially, and makes the algorithm expensive
    - Note: This is for multiple instances of resources
- There are some similarities between the detection algorithm and Banker's algorithm

- Example Of Detection Algorithm
    - Assume there are 5 processes; P0 through P4, and 3 resource types; A (10 instances), B (2 instances), and C (7 instances). The snapshot at time T0 is summarized by the table:

	Allocated			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

    - P0 is the first process to be executed, because it does not need any additional resources, and it releases 1 instance of 'B' when it completes

        - P2 is the second process to execute, because it does not need any additional resources, and it releases 3 instances of resource 'A' and 'C'
            - At this point, we have (3,1,3) available resources
        - Finally, P3, P1, and P4 can be executed because there are now enough available resources to satisfy their requirements
    - The sequence < P0, P2, P3, P1, P4 > will result in `Finish[i] = true` for all 'i'
        - 'i' presents the processes
        - The system is not in a deadlocked state

- Example Of Detection Algorithm Continued
    - Suppose that P2 requests an additional instance of type C
        - The new request table is:

	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

        - The sequence < P0, P2, P3, P1, P4 > is NO longer valid, because the system has insufficient resources to fulfill other processes
            - The only process that can be executed is P0, but every other process cannot be executed because their resource requirements are not met
        - The system is deadlocked
            - It consists of the processes P1, P2, P3, and P4
        - Requesting an additional instance of a resource can be the difference between a deadlocked system and a system in a safe state

- Detection Algorithm Usage
    - The biggest question about deadlock detection algorithm is: When, and how often, should the detection algorithm be invoked?
        - This depends on:
            - How often a deadlock is likely to occur?
            - How many processes will be affected by deadlock when it happens?
        - i.e.
            - Run detection algorithm every X minutes/hours?
    - Approaches to when the deadlock algorithm should run:

        - Invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately
            - Even though a deadlock will be immediately caught, the tradeoff is considerable overhead due to computation time
                - However, we can easily identify the process that "caused" the deadlock
            - Note: If the request cannot be immediately granted, then the system may or may not be in deadlock
        - Invoke the deadlock detection algorithm arbitrarily
            - i.e. Once per hour
            - However, the issue with this is that 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
        - Have the user invoke a deadlock check
            - i.e. Press the 'Refresh' button if the system stops responding. Upon clicking the button, the deadlock detection algorithm will be executed

- Recovery From Deadlock: Process Termination
    - If the algorithm detects deadlock, then what is the best way to get out of the unsafe state the system is in?
        - Should the system abort all deadlocked processes OR abort one process at a time until the deadlock cycle is eliminated
            - The easiest solution is to abort all processes, and then re-run everything
                - However, this is computationally expensive
        - Also, in which order should the processes be aborted?
            - Priority of the process
                - i.e. Abort all low priority processes and keep the high priority processes running
            - How long process has computed, and how much longer to completion
                - i.e. If a process is nearly complete, then it is better to keep it alive, and let it finish. Discarding its work is inefficient
            - Resources the process has used
            - Resources the process needs to complete
                - i.e. If a process requires a lot of resources to complete, then it should be aborted
            - How many processes will need to be terminated
            - Is the process interactive or batch?
    - The best strategy is one that requires the fewest computation time

- Recovery From Deadlock: Resource Preemption
    - There are a few different strategies for recovering from dead-lock. For example:
        - Reset/Reboot

        - Terminates all processes and starts everything from the beginning
        - Not the best solution
    - Rollback
        - Return the system to some (previous) safe state that was not deadlocked
        - Restart process for that state
    - Selecting a victim
        - Based on which resources and processes are to be pre-empted to minimize cost
        - Some processes may always be picked as a victim
            - This is not good practice, because the (victim) process may never be completed

- End
    - Operating systems are among the most complex pieces of software ever developed

- WatchDog Timer
    - Usually used in embedded systems
    - How it works:
        - Assume a simple while loop is performing a number of operations. This operation needs to be completed in 100ms. If a deadlock or livelock occurs, and the time exceeds 100 milliseconds, then WatchDog will reset the process. It can detect that something has gone wrong because WatchDog has its own timer. If WatchDog's timer reaches 0 before the timer of the process, then it will reset the process/system.
    - Not a foolproof solution to deadlock
        - It is more like a band-aid
            - i.e. Assume a system has 1,000,000 states it can be in. And out of those states, 10 states are privy to deadlock. In the rare event that the system reaches one of those unsafe states, WatchDog will reset the system/process, and allow it to try again with the hope of not entering an unsafe state.
        - If the system is poorly designed (i.e. Infinite loop in a process), then WatchDog cannot help because the system will end up in an unsafe state and become dead-locked anyways.