Lab 08
Deadlocks

Section 1: Summary – Self-study before entering the lab

1. Deadlocks in Operating Systems

Deadlocks occur when a set of processes are permanently blocked because each process is waiting for a resource held by another process in the set.

2. System Model and Resource Utilization

- The system consists of **resources**.
- Resource types are designated as R_1, R_2, ..., R_m (e.g., CPU cycles, memory space, I/O devices).
- Each resource type R_i has W_i instances.
- A process utilizes a resource in three steps:
  - **Request**
  - **Use**
  - **Release**

3. Deadlock Characterization

Deadlock can arise if and only if four necessary conditions hold simultaneously:

- **Mutual Exclusion**: Only one process at a time can use a resource (must hold for non-sharable resources).
- **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 of waiting processes $\{P_0, P_1, ..., P_n\}$ such that P_0 is waiting for a resource held by P_1, P_1 for a resource held by P_2, ..., P_n for a resource held by P_0.

4. Resource-Allocation Graph

The state of resource allocation can be illustrated with a **Resource-Allocation Graph**:

- The set of vertices, V, is partitioned into **processes** (P) and **resource types** (R).
- **Request Edge**: A directed edge $P_i \rightarrow R_j$ indicates that process P_i is waiting for resource R_j.
- **Assignment Edge**: A directed edge $R_j \rightarrow P_i$ indicates that resource R_j has been allocated to process P_i.
- **Basic Facts**:
  - If the graph contains **no cycles**, there is **no deadlock**.
  - If the graph contains a **cycle**:
    - If there is **only one instance per resource type**, then a **deadlock exists**.
    - If there are **several instances per resource type**, there is a **possibility of deadlock**.

Methods for Handling Deadlocks

There are three main ways to handle deadlocks:

I. Deadlock Prevention
II. Deadlock Avoidance
III. Deadlock Detection and Recovery
IV. Ignoring the Problem (pretending deadlocks never occur).

I. Deadlock Prevention

This approach invalidates one of the four necessary conditions for deadlock:

- **Mutual Exclusion**: Cannot be invalidated for non-sharable resources.
- **Hold and Wait**: Guarantee that a process requesting a resource does not hold any other resources. This can be achieved by:
    - Requiring a process to request all its resources before starting execution.
    - Allowing a process to request resources only when it has none allocated to it.
    - *Drawbacks*: Low resource utilization and possible starvation.
- **No Preemption**: If a process holding resources requests a resource that is unavailable, all currently held resources are released and added to the waiting list for the process.
- **Circular Wait**: Impose a **total ordering** of all resource types, and require processes to request resources in an **increasing order** of enumeration (this is the most common approach).

II. Deadlock Avoidance

This requires a priori information about the maximum resources a process may need, and dynamically examines the resource-allocation state to ensure a circular-wait condition can never occur.

- **Safe State**: A system is in a safe state if there is a sequence of all processes such that each process P_i can have its needs satisfied by currently available resources plus resources held by processes that finished before P_i.
    - A safe state guarantees **no deadlocks**.
    - An **unsafe state** means a **possibility of deadlock**.
    - Avoidance ensures the system **never enters an unsafe state**.
- **Avoidance Algorithms**:
    - **Single instance per resource type**: Use a Resource-Allocation Graph scheme, where a request is only granted if it doesn't form a cycle. Claim edges ($P_i \rightarrow R_j$) are introduced to represent potential future requests.
    - **Multiple instances per resource type**: Use the **Banker's Algorithm**.

III. Deadlock Detection and Recovery

This allows the system to enter a deadlock state, then uses an algorithm to detect it and a scheme to recover.

- **Detection (Single Instance)**: Maintain a **wait-for graph** (nodes are processes; an edge $P_i \rightarrow P_j$ means P_i is waiting for P_j). A cycle in the wait-for graph indicates a deadlock.
- **Detection (Several Instances)**: Uses a resource-allocation state model with *Available*, *Allocation*, and *Request* matrices/vectors. The detection algorithm checks if there is any safe sequence; if Finish[i] = false for any process P_i after running the algorithm, P_i is deadlocked.

- **Recovery: Process Termination**:
  - Abort all deadlocked processes.
  - Abort one process at a time until the deadlock is eliminated.
  - Selection criteria for abortion include priority, resources used/needed, and how many processes will be terminated.
- **Recovery: Resource Preemption**:
  - **Selecting a victim** to minimize cost.
  - **Rollback** the victim process to a safe state and restart it.
  - Must prevent **starvation** (where the same process is always chosen as the victim) by including the number of rollbacks in the cost factor.

Banker's Algorithm

The **Banker's Algorithm** is a deadlock-avoidance algorithm that is applicable when a system has **multiple instances of each resource type**. It dynamically examines the resource-allocation state to ensure that there can never be a **circular-wait** condition 2, thereby preventing the system from entering an **unsafe state**3.

For the algorithm to work, each process must **a priori declare the maximum number of resources of each type that it may need**.

1. Data Structures for the Banker's Algorithm

The algorithm uses the following data structures, where ***n*** is the number of processes and ***m*** is the number of resource types:

Structure	**Description**	**Type**	**Calculation**
Available	A vector of length m. If ***Available[j]=k***, there are k instances of resource type R_j available.	Vector (m)	
Max	An ***n x m*** matrix. ***Max[i, j]=k*** means process P_i may request at most k instances of resource type R_j.	Matrix (n x m)	
Allocation	An ***n x m*** matrix. ***Allocation[i, j]=k*** means P_i is currently allocated ***k*** instances of R_j.	Matrix (n x m)	
Need	An ***n x m*** matrix. Need[i, j]=k means P_i may need k more instances of R_j to complete its task.	Matrix (n x m)	Need[i,j]=Max[i, j] - Allocation[i, j]

2. Safety Algorithm

The safety algorithm is used to determine if the current resource-allocation state is **safe**, meaning a **safe sequence** of all processes can be found.

- **Initialization**:
  - Initialize a vector Work = Available.
  - Initialize a boolean vector Finish[] = false for all processes i = 0, 1, ..., n-1.
- **Find a Process**: Find an index i such that **both** conditions are true:
  - Finish[i] = false
  - $Need_i <= Work$ (The process's remaining needs can be satisfied by the current available resources)
  - If no such i exists, go to Step 4.
- **Allocate and Terminate (Pretend)**: If a process P_i is found, assume it executes to completion and releases its resources:
  - $Work = Work + Allocation_i$ (Work increases by the resources released by P_i).
  - Finish[i]=true.
  - Go back to Step 2.
- **Conclusion**: If Finish[i]= true for **all** i, then the system is in a **safe state**.

Example of Banker's Algorithm

Consider a system with 5 processes (P0 through P4) and 3 resource types: **A (10 instances total)**, **B (5 instances total)**, and **C (7 instances total)**.

Snapshot at Time T0

The initial state is given by the following Allocation and Max matrices, and the Available vector:

Process	Allocation (A B C)	Max (A B C)
P_0	0 1 0	7 5 3
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

Available (A B C) = 3 3 2

Need Matrix Calculation

The Need matrix is calculated as Max – Allocation:

Process	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

Safety Check

We run the Safety Algorithm with Work = (3, 3, 2).

1. **P1 is selected**: Need_1 = (1, 2, 2) <= Work = (3, 3, 2) is **true**
   - Work = Work} + \text{Allocation}_1 = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)$.
   - Finish_1 = true.
2. **P3 is selected**: Need_3 = (0, 1, 1) <= Work= (5, 3, 2) is **true**.
   - Work = Work + Allocation_3=(5, 3, 2) + (2, 1, 1) = (7, 4, 3).
   - Finish_3 = true.
3. **P4 is selected**: Need_4=(4, 3, 1) <= Work=(7, 4, 3) is **true**.
   - Work = Work+Allocation_4 =(7, 4, 3) + (0, 0, 2) = (7, 4, 5).
   - Finish_4 = true.
4. **P2 is selected**: Need_2 = (6, 0, 0) <= Work=(7, 4, 5) is **true**.
   - Work} = Work + Allocation_2=(7, 4, 5) + (3, 0, 2) =(10, 4, 7).
   - Finish_2= true.
5. **P0 is selected**: Need_0=(7, 4, 3) <= Work= (10, 4, 7) is **true**.
   - Work = Work + Allocation_0 = (10, 4, 7) + (0, 1, 0) = (10, 5, 7).
   - Finish_0 = true.

Since all processes finished, the system is in a **safe state**. The resulting safe sequence is <P_1, P_3, P_4, P_2, P_0>.

Resource-Request Algorithm (Briefly)

When a process, say P_i, requests resources, the Banker's Algorithm uses the **Resource-Request Algorithm** to decide whether to grant the request immediately.

For example, if process P_1 requests Request_1 = (1, 0, 2):

- **Check 1**: Is Request_1 <= Need_1?
  - (1, 0, 2) <= (1, 2, 2) is true. (If false, it means P_1 exceeded its initial max claim).
- **Check 2**: Is $\text{Request}_{1 <=}$ Available?
  - (1, 0, 2) <= (3, 3, 2) is true. (If false, P_1 must wait).

- **Pretend Allocation**: The state is modified as if the request was granted:
  - Available = (3, 3, 2) - (1, 0, 2) = (2, 3, 0).
  - Allocation_1 = (2, 0, 0) + (1, 0, 2) = (3, 0, 2).
  - Need_1 = (1, 2, 2) - (1, 0, 2) = (0, 2, 0).
- **Final Safety Check**: Run the **Safety Algorithm** on the new state. If the new state is found to be safe (which it is, with sequence <P_1, P_3, P_4, P_0, P_2>, the request is granted. Otherwise, the allocation is rolled back, and P_1 must wait.

Section 2: Discussion – selected questions only (1 Hour)

1. Explain the four conditions that must be held for deadlock to occur.

2. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

3. Consider the following snapshot of a system:

Process	Allocation ABCD	Max ABCD	Available ABCD
P0	0012	0012	1520
P1	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

4. Answer the following questions using the banker's algorithm:
   a) What does the content of the matrix *Need*?
   b) Is the system in a safe state?
   c) If a higher priority request is made by process P1 for (0,4,2,0) resources, can the request be granted immediately?
   d) Is the new system in a safe state?

5. What is the optimistic assumption made in the deadlock-detection algorithm? How could this assumption be violated?

Section 3: Practical exercises

Exercise 1: [Book example] Figure 8.1 page 320.

This is a complete example that uses the segment defined in the figure above to demonstrate the occurrence of a deadlock.

Both threads are permanently blocked, fulfilling the definition of a deadlock.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
```

```
// 1. Initialization of the two mutex locks (Resources)
pthread_mutex_t first_mutex;  // Resource 1
pthread_mutex_t second_mutex; // Resource 2

// Function for the first thread (Process 1)
void *do_work_one(void *param)
{
    printf("Thread 1: Trying to acquire first_mutex...\n");
    // Acquire first_mutex (Resource 1)
    pthread_mutex_lock(&first_mutex);
    printf("Thread 1: Acquired first_mutex. Now trying to acquire
second_mutex...\n");

    // Pause briefly to ensure Thread 2 can acquire its first lock
(second_mutex)
    // This timing is what creates the deadlock condition
    sleep(1);

    // Attempt to acquire second_mutex (Resource 2). This is where it will
block if Thread 2 holds it.
    pthread_mutex_lock(&second_mutex);

    // CRITICAL SECTION (Will only be reached if deadlock is avoided)
    printf("Thread 1: Acquired both locks. Doing work...\n");

    // Release locks in reverse order of acquisition
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    printf("Thread 1: Finished and released both locks.\n");

    pthread_exit(0);
}

// Function for the second thread (Process 2)
void *do_work_two(void *param)
{
    printf("Thread 2: Trying to acquire second_mutex...\n");
    // Acquire second_mutex (Resource 2) in the REVERSE ORDER
    pthread_mutex_lock(&second_mutex);
    printf("Thread 2: Acquired second_mutex. Now trying to acquire
first_mutex...\n");

    // Pause briefly (not strictly necessary but aids in demonstrating the
deadlock)
    sleep(1);

    // Attempt to acquire first_mutex (Resource 1). This is where it will
block if Thread 1 holds it.
    pthread_mutex_lock(&first_mutex);

    // CRITICAL SECTION (Will only be reached if deadlock is avoided)
    printf("Thread 2: Acquired both locks. Doing work...\n");

    // Release locks
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    printf("Thread 2: Finished and released both locks.\n");

    pthread_exit(0);
}

int main()
{
    pthread_t tid1, tid2;

    // Initialize the mutexes (Resources)
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);

    printf("System initialized. Creating threads...\n");

    // Create the two threads (Processes)
    pthread_create(&tid1, NULL, do_work_one, NULL);
    pthread_create(&tid2, NULL, do_work_two, NULL);

    // Wait for the threads to finish (which they won't if a deadlock occurs)
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("Main thread: All threads completed (This line is often not
reached when deadlocked).\n");

    // Destroy mutexes
    pthread_mutex_destroy(&first_mutex);
    pthread_mutex_destroy(&second_mutex);

    return 0;
}
```

1. Compile and run the program and observe the output.
2. Try to understand why the system hangs and no more messages are printed out.

Exercise 2: [Book example] Figure 8.7 page 330.
This is a complete example that uses the segment defined in the figure above to demonstrate the prevention of a deadlock. This code prevents the deadlock by invalidating the Circular Wait condition. Both do_work_one_SAFE and do_work_two_SAFE acquire the mutexes in the ascending order of enumeration (1 → 5).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Global Mutex Locks (Resources)
pthread_mutex_t first_mutex;  // Assume Order = 1
pthread_mutex_t second_mutex; // Assume Order = 5

// Function for the first thread (Process 1) - SAFE
void *do_work_one_SAFE(void *param)
{
    printf("Thread 1: Trying to acquire locks in ORDER 1 → 5...\n");

    // 1. Acquire first_mutex (Order 1)
    pthread_mutex_lock(&first_mutex);
    printf("Thread 1: Acquired first_mutex. Now acquiring second_mutex...
\n");

    // 2. Acquire second_mutex (Order 5)
    pthread_mutex_lock(&second_mutex);

    // CRITICAL SECTION
    printf("Thread 1: Acquired both locks and is doing work.\n");

    // Release locks in reverse order of acquisition
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    printf("Thread 1: Finished and released both locks.\n");

    pthread_exit(0);
}

// Function for the second thread (Process 2) - SAFE
void *do_work_two_SAFE(void *param)
{
    printf("Thread 2: Trying to acquire locks in ORDER 1 → 5...\n");

    // 1. Acquire first_mutex (Order 1) - ***CHANGE FROM DEADLOCKED
VERSION***
    pthread_mutex_lock(&first_mutex);
    printf("Thread 2: Acquired first_mutex. Now acquiring second_mutex...
\n");

    // 2. Acquire second_mutex (Order 5)
    pthread_mutex_lock(&second_mutex);

    // CRITICAL SECTION
    printf("Thread 2: Acquired both locks and is doing work.\n");
    // Release locks
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    printf("Thread 2: Finished and released both locks.\n");

    pthread_exit(0);
}

int main()
{
    pthread_t tid1, tid2;

    // Initialize the mutexes
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);

    printf("System initialized. Creating SAFE threads...\n");

    // Create the two threads, both running the SAFE functions
    pthread_create(&tid1, NULL, do_work_one_SAFE, NULL);
    pthread_create(&tid2, NULL, do_work_two_SAFE, NULL);

    // Wait for the threads to finish
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("Main thread: All threads completed successfully (Deadlock
Avoided).\n");

    // Destroy mutexes
    pthread_mutex_destroy(&first_mutex);
    pthread_mutex_destroy(&second_mutex);

    return 0;
}
```

1. Compile and run the code
2. Compare the output of the program in exercise 1.
3. Discover the differences between this program and the program in Ex1.

Exercise 3: The following C program can be used to solve the example in section 8.6.3.3.

This is a complete example that uses the segment defined in the figure above to demonstrate the prevention of a deadlock. This code prevents the deadlock by invalidating the Circular Wait condition. Both do_work_one_SAFE and do_work_two_SAFE acquire the mutexes in the ascending order of enumeration (1 → 5).

```
#include <stdio.h>
#include <stdbool.h>

// Define system parameters
#define P 5 // Number of processes (P0 to P4)
#define R 3 // Number of resource types (A, B, C)

// Global data structures for the Banker's Algorithm (Initial T0 State)
// Note: This uses the initial state *before* P1's request is provisionally
granted.
int max[P][R] = {
    {7, 5, 3}, // P0 Max
    {3, 2, 2}, // P1 Max
    {9, 0, 2}, // P2 Max
    {2, 2, 2}, // P3 Max
    {4, 3, 3}  // P4 Max
};

int allocation[P][R] = {
    {0, 1, 0}, // P0 Allocation
    {2, 0, 0}, // P1 Allocation
    {3, 0, 2}, // P2 Allocation
    {2, 1, 1}, // P3 Allocation
    {0, 0, 2}  // P4 Allocation
};

int available[R] = {3, 3, 2}; // Available resources (A, B, C)

// Need Matrix calculation
int need[P][R];

// Global temporary variables for safety check
int work[R];
bool finish[P];
int safe_sequence[P];

// Function to calculate the Need Matrix
void calculateNeed()
{
    for (int i = 0; i < P; i++)
    {
        for (int j = 0; j < R; j++)
        {
            // Need[i,j] = Max[i,j] - Allocation[i,j]
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
```

```c
// -------------------------------------------------------------------------
// 2. Safety Algorithm (Page 335)
// -------------------------------------------------------------------------
bool isSafe()
{
    int i, j, count = 0;

    // 1. Initialize Work and Finish
    for (j = 0; j < R; j++)
    {
        work[j] = available[j]; // Work = Available
    }
    for (i = 0; i < P; i++)
    {
        finish[i] = false; // Finish[] = false
    }

    // 2. Find an i such that Finish[i] == false AND Need[i] <= Work
    while (count < P)
    {
        bool found = false;
        for (i = 0; i < P; i++)
        {
            if (finish[i] == false)
            {
                // Check if Need[i] <= Work
                int k;
                for (k = 0; k < R; k++)
                {
                    if (need[i][k] > work[k])
                    {
                        break;
                    }
                }

                // If Need[i] <= Work is true
                if (k == R)
                {
                    // 3. Work = Work + Allocation[i] and Finish[i] = true
                    for (j = 0; j < R; j++)
                    {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = true;
                    safe_sequence[count++] = i;
                    found = true;
                }
            }
        }

        // If no such i is found in an iteration, break the loop
        if (found == false)
        {
            break;
        }
    }
    // 4. Check if Finish[i] == true for all i
    if (count == P)
    {
        return true; // System is in a safe state
    }
    else
    {
        return false; // System is in an unsafe state
    }
}

// ---------------------------------------------------------------------
// 3. Resource-Request Algorithm (Page 29)
// ---------------------------------------------------------------------
void requestResources(int pid, int request[])
{
    int i, j;

    printf("\nProcess P%d requests resources: (%d, %d, %d)\n",
           pid, request[0], request[1], request[2]);

    // Step 1: Check Request <= Need
    for (i = 0; i < R; i++)
    {
        if (request[i] > need[pid][i])
        {
            printf("--- ERROR: P%d exceeded its maximum claim (Request >
Need) ---\n", pid);
            return;
        }
    }

    // Step 2: Check Request <= Available
    for (i = 0; i < R; i++)
    {
        if (request[i] > available[i])
        {
            printf("--- P%d must wait. Resources are not available (Request >
Available) ---\n", pid);
            return;
        }
    }

    // Step 3: Pretend to allocate requested resources

    // Save current state for restoration if necessary
    int alloc_temp[R], avail_temp[R], need_temp[R];
    for (i = 0; i < R; i++)
    {
        alloc_temp[i] = allocation[pid][i];
        avail_temp[i] = available[i];
        need_temp[i] = need[pid][i];
    }

    // Apply temporary allocation
    for (i = 0; i < R; i++)
    {
        available[i] -= request[i];
        allocation[pid][i] += request[i];
        need[pid][i] -= request[i];
    }

    // Run Safety Algorithm
    if (isSafe())
    {
        // If safe, the allocation is granted
        printf(">>> Request GRANTED. System remains in a safe state.\n");
        printf("    Safe Sequence found: <");
        for (i = 0; i < P; i++)
        {
            printf("P%d%s", safe_sequence[i], (i == P - 1) ? "" : ", ");
        }
        printf(">\n");
    }
    else
    {
        // If unsafe, restore the old state and P must wait
        printf(">>> Request DENIED. Granting request would lead to an unsafe 
state.\n");
        printf("    P%d must wait, and the old resource-allocation state is 
restored.\n", pid);

        for (i = 0; i < R; i++)
        {
            allocation[pid][i] = alloc_temp[i];
            available[i] = avail_temp[i];
            need[pid][i] = need_temp[i];
        }
    }
}

// ----------------------------------------------------------------------
// 4. Main Execution
// ----------------------------------------------------------------------
int main()
{
    calculateNeed(); // Calculate initial Need Matrix

    printf("--- Initial State at T0 ---\n");
    printf("Initial Available: (%d, %d, %d)\n", available[0], available[1], 
available[2]);

    // Check initial state safety (as per the example)
    if (isSafe())
    {
        printf("Initial state is SAFE.\n");
        printf("Safe Sequence found: <");
        for (int i = 0; i < P; i++)
        {
            printf("P%d%s", safe_sequence[i], (i == P - 1) ? "" : ", ");
        }
        printf(">\n");
    }
    else
    {
        printf("Initial state is UNSAFE.\n");
    }

    // Example 8.6.3.3: P1 requests (1, 0, 2)
    int request1[] = {1, 0, 2};
    requestResources(1, request1);

    return 0;
}
```

1. Compile and run the code.
2. Trace the solution generated by this program with the steps on pages 335, 336, and 337 of the textbook.
3. Modify the program to solve the exercise 8.3, 8.21 and 8.9 of the textbook.