Larbi Ben M'hidi University – Oum El Bouaghi

Faculty of Exact Sciences and Nature and Life Sciences

Department of Mathematics and Computer Science

Module: Operating Systems II

Lecturer: Dr. Moustafa Sadek KAHIL

2025 – 2026

Chapter VI: Deadlock

4.1 Introduction

Imagine two people (P1 and P2) sharing a single pen and a single notebook:

- P1 holds the pen and needs the notebook
- P2 holds the notebook and needs the pen
- Neither can proceed → **Deadlock**

In modern operating systems, multiple processes compete for limited system resources. When processes are not properly coordinated, they can enter a deadlock state, a situation where two or more processes are permanently blocked because each is waiting for resources held by another

A deadlock occurs when a set of processes is in a wait state where each process is waiting for a resource that another process in the set holds, creating a circular chain of dependencies. None of the processes can proceed, and the system becomes unresponsive.

Why Study Deadlock?

Understanding deadlock is critical because:

1. **System Reliability**: Deadlocks can cause complete system freezes
2. **Resource Utilization**: Deadlocked resources are wasted
3. **Performance**: Detection and recovery overhead affects system performance
4. **Design Decisions**: OS designers must choose appropriate handling strategies

Real-World Examples:

1 Database Systems

- Transaction T1 locks record A, needs record B
- Transaction T2 locks record B, needs record A
- Both transactions wait indefinitely

2 Traffic Gridlock

- Four cars arrive at a four-way intersection simultaneously
- Each car holds one quadrant and waits for the next
- No car can move → Traffic deadlock

3 Printer Spooling

- Process P1 holds printer, needs tape drive
- Process P2 holds tape drive, needs printer
- Both processes blocked

4.2 System Model and Resource Allocation

Resources in an operating system can be classified as:

1. Reusable Resources:

- Can be used by multiple processes over time
- Examples: CPU, memory, printers, disk drives
- Not consumed by use

2. Consumable Resources:

- Created and destroyed through use
- Examples: interrupts, signals, messages
- Produced by one process, consumed by another

A process typically follows this sequence when using resources:

```
// Typical resource usage pattern
void process_example() {
    // 1. Request the resource
    request_resource(R);
```

```
    // 2. Use the resource
    use_resource(R);

    // 3. Release the resource
    release_resource(R);
}
```

The operating system maintains tables to track resource allocation:

Process	Resources Held	Resources Requested
P1	R1	R2
P2	R2	R3
P3	R3	R1

4.3 Characterization of Deadlock

Necessary Conditions for Deadlock: For a deadlock to occur, all four of the following conditions must hold simultaneously:

1. Mutual Exclusion

- At least one resource must be held in a non-sharable mode
- Only one process can use the resource at a time
- Example: A printer can only print one job at a time

2. Hold and Wait

- A process must be holding at least one resource
- While waiting to acquire additional resources held by other processes
- Example: Process holds printer, requests scanner

3. No Preemption

- Resources cannot be forcibly taken away from processes
- Resources must be released voluntarily by the holding process
- Example: Cannot interrupt a print job mid-way

4. Circular Wait

- A circular chain of processes must exist
- Each process holds a resource that the next process in the chain needs
- Example: P1→R1→P2→R2→P3→R3→P1

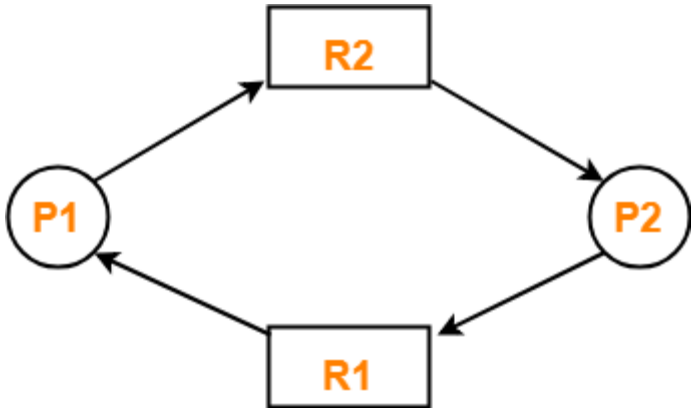

Figure 4.1: Deadlock Scenario

The figure above illustrates a classic deadlock scenario where:

- P1 holds R1 and waits for R2
- P2 holds R2 and waits for R1

This creates a circular dependency → **DEADLOCK**

4.4 Resource Allocation Graph (RAG)

A Resource Allocation Graph (RAG) is a directed graph used to model resource allocation and detect deadlocks.

Graph Components:

1 Vertices:

- **$P = \{P_1, P_2, ..., P_n\}$**: Set of all processes
- **$R = \{R_1, R_2, ..., R_m\}$**: Set of all resource types

2 Edges:

- **Request Edge ($P_i \rightarrow R_j$)**: Process P_i is requesting resource R_j
- **Assignment Edge ($R_j \rightarrow P_i$)**: Resource R_j is allocated to process P_i

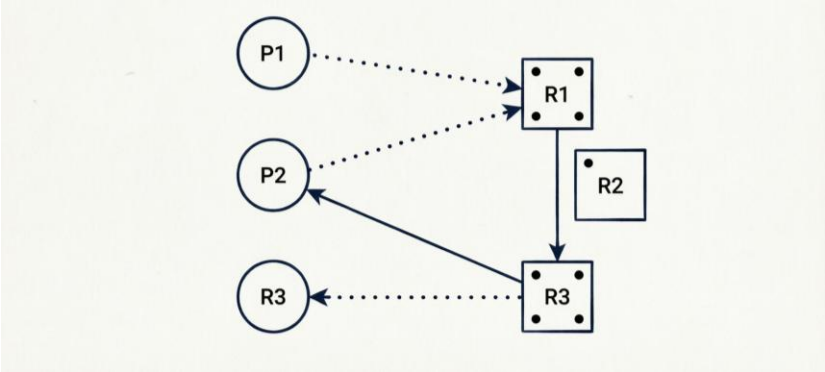

Figure 4.2: Resource Allocation Graph (RAG) Diagram

Deadlock Detection Using RAG

Rule 1: No Cycle → No Deadlock

- If the RAG contains no cycles, the system is not in deadlock

Rule 2: Cycle with Single Instance Resources → Deadlock

- If each resource type has only one instance and a cycle exists → **Deadlock**

Rule 3: Cycle with Multiple Instances → Possible Deadlock

- If resources have multiple instances, a cycle indicates potential deadlock

Example: RAG Analysis

System State:

Processes: P1, P2, P3
Resources: R1 (2 instances), R2 (1 instance), R3 (3 instances)

Current Allocation:
- P1 holds R1 (1 instance)
- P2 holds R3 (1 instance)
- P3 holds R1 (1 instance)

Current Requests:
- P1 requests R2
- P2 requests R3
- P3 requests R2

Analysis:

- Check for cycles in the graph
- If cycle exists and resources are single-instance → Deadlock detected

4.5 Methods for Handling Deadlock

Operating systems use three main approaches to handle deadlocks:

1. **Deadlock Prevention**: Prevent at least one necessary condition
2. **Deadlock Avoidance**: Dynamically avoid unsafe states
3. **Deadlock Detection & Recovery**: Allow deadlocks, detect and recover

4.5.1 Deadlock Prevention

Deadlock prevention works by ensuring that at least one of the four necessary conditions cannot occur.

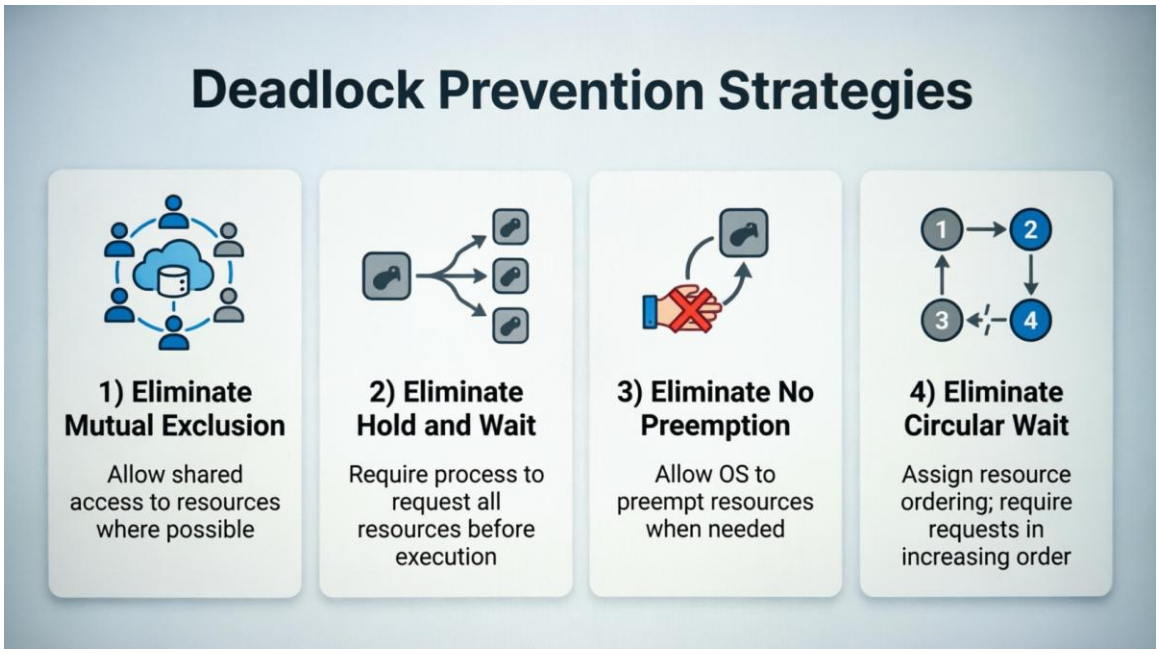

Figure 4.4: Deadlock Prevention Strategies

Strategy 1: Eliminate Mutual Exclusion

Approach:

- Make resources sharable where possible
- Not applicable to inherently non-sharable resources (printers, tape drives)

Example:

```
// Instead of exclusive access
// Use read-only shared access where applicable

// Shared memory segment (readable by multiple processes)
int *shared_data = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0);
```

```
// Multiple processes can READ simultaneously
```

Limitations:

- Some resources are inherently non-sharable
- Cannot eliminate mutual exclusion for all resources

Strategy 2: Eliminate Hold and Wait

Approach:

- Require processes to request all resources before execution begins
- Or: Require processes to release all resources before requesting new ones

Implementation:

```
// Method 1: Request all resources upfront
void process_all_at_once() {
    // Request ALL needed resources at start
    request_resources(R1, R2, R3);

    // Execute only if all resources granted
    if (all_resources_granted) {
        execute_process();
        release_all_resources();
    }
}

// Method 2: Release before requesting
void release_before_request() {
    request_resource(R1);
    use_resource(R1);

    // Release R1 before requesting R2
    release_resource(R1);
    request_resource(R2);
    use_resource(R2);
}
```

Advantages:

- Simple to implement
- Prevents hold and wait condition

Disadvantages:

- Low resource utilization (resources held but not used)
- Process starvation (may wait indefinitely for all resources)
- Processes may not know all needed resources in advance

Strategy 3: Eliminate No Preemption

Approach:

- Allow the OS to forcibly take resources from processes
- If a process cannot get needed resources, it must release held resources

Implementation:

```
// Preemption algorithm
void handle_resource_request(Process P, Resource R) {
    if (R is available) {
        allocate(R, P);
    } else {
        // Check if R is held by another process
        Process holder = get_holder(R);

        if (holder is waiting for other resources) {
            // Preempt resource from holder
            preempt_resource(R, holder);
            allocate(R, P);
            // Restart holder when resources available
        } else {
            // P must wait or release its resources
            P.wait_for(R);
        }
    }
}
```

Advantages:

- Better resource utilization
- Prevents indefinite waiting

Disadvantages:

- Complex implementation
- May cause process rollback

- Not applicable to all resource types (e.g., printers)

Strategy 4: Eliminate Circular Wait

Approach:

- Impose a total ordering on all resource types
- Require processes to request resources in increasing order

Implementation:

```c
// Define resource ordering
#define PRINTER    1
#define SCANNER    2
#define TAPE_DRIVE 3
#define DISK       4

// Process must request in increasing order
void process_ordered_request() {
    // CORRECT: Increasing order
    request_resource(PRINTER);    // 1
    request_resource(SCANNER);    // 2
    request_resource(TAPE_DRIVE); // 3

    // WRONG: Would cause circular wait
    // request_resource(TAPE_DRIVE);
    // request_resource(PRINTER);  // Violates ordering!
}

// Verification function
int can_request(Resource current, Resource requested) {
    return (requested > current);  // Must be in increasing order
}
```

Advantages:

- Simple and effective
- Low overhead
- Guarantees no circular wait

Disadvantages:

- May not match natural resource usage order
- Can lead to inefficient resource utilization

4.5.2 Deadlock Avoidance

Deadlock avoidance allows all four necessary conditions but dynamically examines the resource allocation state to ensure the system never enters an unsafe state.

Concept of Safe State

Safe State:

- A system is in a safe state if there exists a safe sequence of all processes
- In a safe sequence, each process can obtain its needed resources and complete

Safe Sequence:

- A sequence <P_1, P_2, ..., P_n> is safe if:
    - For each P_i, its resource needs can be satisfied by:
        - Currently available resources +
        - Resources held by all P_j where j < i

Unsafe State:

- No safe sequence exists
- Does not mean deadlock, but deadlock is possible

Banker's Algorithm

The Banker's Algorithm, developed by Edsger Dijkstra, is the most famous deadlock avoidance algorithm.

Analogy: Like a bank that never loans out cash in a way that prevents it from satisfying all customers' future needs.

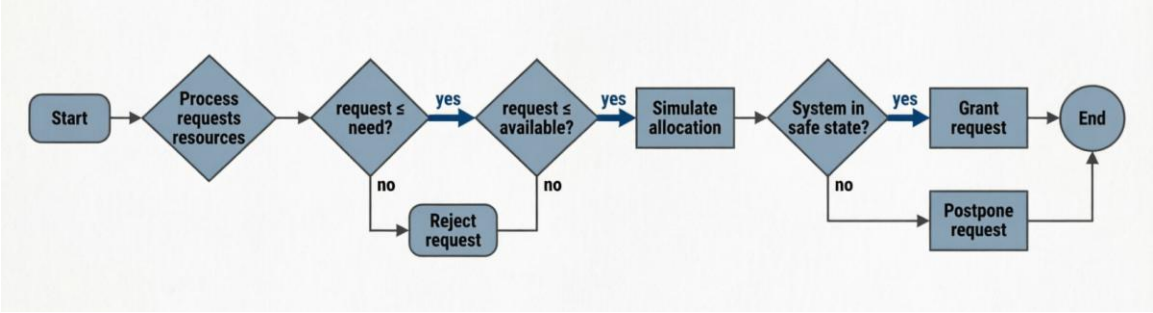

Figure 4.3: Banker's Algorithm Flowchart

Data Structures:

```
#define N_PROCESSES 5
```

```c
#define N_RESOURCES 3

// Available resources
int Available[N_RESOURCES];

// Maximum demand of each process
int Max[N_PROCESSES][N_RESOURCES];

// Currently allocated resources
int Allocation[N_PROCESSES][N_RESOURCES];

// Remaining need of each process
int Need[N_PROCESSES][N_RESOURCES];

// Work and Finish arrays for safety algorithm
int Work[N_RESOURCES];
bool Finish[N_PROCESSES];
```

Safety Algorithm:

```c
// Check if system is in safe state
bool is_safe_state() {
    int i, j;

    // Step 1: Initialize
    for (j = 0; j < N_RESOURCES; j++)
        Work[j] = Available[j];

    for (i = 0; i < N_PROCESSES; i++)
        Finish[i] = false;

    // Step 2: Find a process that can complete
    bool found;
    do {
        found = false;

        for (i = 0; i < N_PROCESSES; i++) {
            if (!Finish[i]) {
                // Check if Need <= Work
                bool can_finish = true;
                for (j = 0; j < N_RESOURCES; j++) {
                    if (Need[i][j] > Work[j]) {
                        can_finish = false;
                        break;
```

```
                    }
                }

                if (can_finish) {
                    // Process Pi can finish
                    for (j = 0; j < N_RESOURCES; j++) {
                        Work[j] += Allocation[i][j];
                    }
                    Finish[i] = true;
                    found = true;

                    printf("Process P%d can complete\n", i);
                }
            }
        }
    } while (found);

    // Step 3: Check if all processes finished
    for (i = 0; i < N_PROCESSES; i++) {
        if (!Finish[i])
            return false;  // Unsafe state
    }

    return true;  // Safe state
}
```

Resource Request Algorithm:

```
// Handle resource request from process Pi
int request_resources(int Pi, int Request[]) {
    int j;

    // Step 1: Check if Request <= Need
    for (j = 0; j < N_RESOURCES; j++) {
        if (Request[j] > Need[Pi][j]) {
            printf("Error: Request exceeds maximum claim\n");
            return -1;
        }
    }

    // Step 2: Check if Request <= Available
    for (j = 0; j < N_RESOURCES; j++) {
        if (Request[j] > Available[j]) {
            printf("Process P%d must wait: resources not available\n",
```

```
Pi);
            return 0;  // Must wait
        }
    }

    // Step 3: Pretend to allocate resources
    for (j = 0; j < N_RESOURCES; j++) {
        Available[j] -= Request[j];
        Allocation[Pi][j] += Request[j];
        Need[Pi][j] -= Request[j];
    }

    // Step 4: Check if resulting state is safe
    if (is_safe_state()) {
        printf("Request granted: System in safe state\n");
        return 1;  // Request granted
    } else {
        // Rollback: restore previous state
        for (j = 0; j < N_RESOURCES; j++) {
            Available[j] += Request[j];
            Allocation[Pi][j] -= Request[j];
            Need[Pi][j] += Request[j];
        }
        printf("Request denied: Would result in unsafe state\n");
        return 0;  // Must wait
    }
}
```

Example Scenario:

Consider a system with:

- **5 processes**: P_0, P_1, P_2, P_3, P_4
- **3 resource types**: A (10 instances), B (5 instances), C (7 instances)

Initial State:

```
Available Resources:
A  B  C
3  3  2

Allocation Matrix:
     A  B  C
P0   0  1  0
```

```
P1   2  0  0
P2   3  0  2
P3   2  1  1
P4   0  0  2

Max Matrix:
     A  B  C
P0   7  5  3
P1   3  2  2
P2   9  0  2
P3   2  2  2
P4   4  3  3

Need Matrix (Max - Allocation):
     A  B  C
P0   7  4  3
P1   1  2  2
P2   6  0  0
P3   0  1  1
P4   4  3  1
```

Safety Check Execution:

```c
// Step-by-step safety check
void demonstrate_safety_check() {
    printf("=== Safety Algorithm Execution ===\n\n");

    int Work[] = {3, 3, 2};  // Available
    bool Finish[5] = {false, false, false, false, false};
    int safe_sequence[5];
    int seq_index = 0;

    printf("Initial Work: [%d, %d, %d]\n\n", Work[0], Work[1], Work[2]);

    // Iteration 1: P1 or P3 can finish
    printf("Iteration 1:\n");
    printf("P1 Need [1,2,2] <= Work [3,3,2]? YES\n");
    printf("P1 executes and releases resources\n");
    Work[0] += 2; Work[1] += 0; Work[2] += 0;  // P1's allocation
    printf("New Work: [%d, %d, %d]\n\n", Work[0], Work[1], Work[2]);
    Finish[1] = true;
    safe_sequence[seq_index++] = 1;

    // Continue iterations...
```

```
    // Final safe sequence: <P1, P3, P4, P0, P2>

    printf("Safe Sequence: <P1, P3, P4, P0, P2>\n");
    printf("System is in SAFE state\n");
}
```

Resource Request Example:

```
// Example: P1 requests resources
void example_request() {
    int Request[] = {1, 0, 2};  // P1 requests 1A, 0B, 2C

    printf("Process P1 requests: [%d, %d, %d]\n",
           Request[0], Request[1], Request[2]);

    // Check: Request <= Need?
    // [1,0,2] <= [1,2,2]? YES

    // Check: Request <= Available?
    // [1,0,2] <= [3,3,2]? YES

    // Pretend to allocate
    int new_available[] = {2, 3, 0};

    // Check if new state is safe
    if (is_safe_state_with_new_allocation()) {
        printf("Request GRANTED\n");
    } else {
        printf("Request DENIED - would cause unsafe state\n");
        printf("Process P1 must wait\n");
    }
}
```

Complete Implementation:

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

#define P 5  // Number of processes
#define R 3  // Number of resource types

// Global data structures
int Available[R];
int Max[P][R];
```

```
int Allocation[P][R];
int Need[P][R];

// Function prototypes
void calculate_need();
bool is_safe();
bool request_resources(int process, int request[]);
void print_matrices();

int main() {
    // Initialize data
    // ... initialization code ...

    calculate_need();
    print_matrices();

    if (is_safe()) {
        printf("System is in SAFE state\n");
    } else {
        printf("System is in UNSAFE state\n");
    }

    return 0;
}

void calculate_need() {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            Need[i][j] = Max[i][j] - Allocation[i][j];
}

bool is_safe() {
    int Work[R], Finish[P];
    int safe_seq[P];
    int count = 0;

    // Initialize Work and Finish
    for (int j = 0; j < R; j++)
        Work[j] = Available[j];

    for (int i = 0; i < P; i++)
        Finish[i] = false;

    // Find safe sequence
    while (count < P) {
```

```
        bool found = false;

        for (int i = 0; i < P; i++) {
            if (!Finish[i]) {
                int j;
                for (j = 0; j < R; j++)
                    if (Need[i][j] > Work[j])
                        break;

                if (j == R) {  // Process can finish
                    for (int k = 0; k < R; k++)
                        Work[k] += Allocation[i][k];

                    Finish[i] = true;
                    safe_seq[count++] = i;
                    found = true;
                }
            }
        }

        if (!found) {
            printf("System is in UNSAFE state\n");
            return false;
        }
    }

    printf("Safe Sequence: < ");
    for (int i = 0; i < P; i++)
        printf("P%d ", safe_seq[i]);
    printf(">\n");

    return true;
}
```

Advantages of Banker's Algorithm:

- Guarantees system never enters deadlock state
- Allows more concurrency than prevention
- Dynamic resource allocation

Disadvantages:

- High computational overhead ($O(n^2 \times m)$)
- Requires knowing maximum resource needs in advance
- Processes may wait longer than necessary

- Not practical for systems with many processes/resources

4.5.3 Deadlock Detection and Recovery

When prevention and avoidance are not used, the system can allow deadlocks to occur, then detect and recover from them.

Deadlock Detection

a Single Instance Resources: Wait-For Graph

For systems with single-instance resources, use a Wait-For Graph:

```
// Wait-For Graph representation
// Processes: P0, P1, P2, ..., Pn
// Edge Pi -> Pj means Pi is waiting for Pj to release a resource

// Detection algorithm: Look for cycles
bool detect_deadlock_wait_for_graph() {
    // Build wait-for graph from resource allocation
    bool graph[P][P] = {false};

    for (int i = 0; i < P; i++) {
        for (int j = 0; j < P; j++) {
            if (i != j && is_waiting_for(i, j)) {
                graph[i][j] = true;
            }
        }
    }

    // Detect cycle using DFS
    bool visited[P] = {false};
    bool rec_stack[P] = {false};

    for (int i = 0; i < P; i++) {
        if (has_cycle_dfs(i, graph, visited, rec_stack)) {
            printf("DEADLOCK DETECTED!\n");
            return true;
        }
    }

    return false;
}

bool has_cycle_dfs(int node, bool graph[P][P], bool visited[], bool
rec_stack[]) {
    if (!visited[node]) {
```

```
        visited[node] = true;
        rec_stack[node] = true;

        for (int i = 0; i < P; i++) {
            if (graph[node][i]) {
                if (!visited[i] && has_cycle_dfs(i, graph, visited,
rec_stack))
                    return true;
                else if (rec_stack[i])
                    return true;
            }
        }
    }

    rec_stack[node] = false;
    return false;
}
```

b Multiple Instance Resources: Detection Algorithm

For systems with multiple instances of resources:

```
// Data structures
int Available[R];
int Allocation[P][R];
int Request[P][R];  // Current request of each process

// Deadlock detection algorithm
bool detect_deadlock_multiple_resources() {
    int Work[R];
    bool Finish[P];

    // Step 1: Initialize
    for (int j = 0; j < R; j++)
        Work[j] = Available[j];

    for (int i = 0; i < P; i++)
        Finish[i] = (Allocation[i][0] == 0 && Allocation[i][1] == 0);

    // Step 2: Find processes that can complete
    bool changed;
    do {
        changed = false;

        for (int i = 0; i < P; i++) {
```

```
            if (!Finish[i]) {
                // Check if Request[i] <= Work
                bool can_satisfy = true;
                for (int j = 0; j < R; j++) {
                    if (Request[i][j] > Work[j]) {
                        can_satisfy = false;
                        break;
                    }
                }

                if (can_satisfy) {
                    // Process can complete
                    for (int j = 0; j < R; j++)
                        Work[j] += Allocation[i][j];

                    Finish[i] = true;
                    changed = true;
                }
            }
        }
    } while (changed);

    // Step 3: Check for deadlock
    for (int i = 0; i < P; i++) {
        if (!Finish[i]) {
            printf("Process P%d is deadlocked\n", i);
            return true;
        }
    }

    return false;
}
```

When to Invoke Detection?

1. **After every resource request**: High overhead, immediate detection
2. **Periodically (e.g., every hour)**: Moderate overhead
3. **When CPU utilization drops**: Low overhead, reactive approach

Deadlock Recovery

Once deadlock is detected, the system must recover using one of these methods:

Method 1: Process Termination

Option A: Terminate All Deadlocked Processes

```c
// Abort all deadlocked processes
void recover_by_terminating_all(int deadlocked_processes[]) {
    printf("Terminating all deadlocked processes...\n");

    for (int i = 0; deadlocked_processes[i] != -1; i++) {
        int pid = deadlocked_processes[i];

        // Forcefully terminate process
        kill_process(pid);

        // Release all resources held by process
        release_all_resources(pid);

        printf("Process P%d terminated\n", pid);
    }

    printf("Deadlock resolved\n");
}
```

Option B: Terminate One Process at a Time

```c
// Select victim process based on cost factors
int select_victim_process(int deadlocked_processes[]) {
    int victim = -1;
    int min_cost = INT_MAX;

    for (int i = 0; deadlocked_processes[i] != -1; i++) {
        int pid = deadlocked_processes[i];

        // Calculate cost of terminating this process
        int cost = calculate_termination_cost(pid);

        if (cost < min_cost) {
            min_cost = cost;
            victim = pid;
        }
    }

    return victim;
}
```

```c
int calculate_termination_cost(int pid) {
    int cost = 0;

    // Factors to consider:
    cost += process_priority(pid);        // Lower priority = lower cost
    cost += execution_time_so_far(pid);   // Less time = lower cost
    cost += resources_held(pid);          // Fewer resources = lower cost
    cost += resources_needed(pid);        // More needed = higher cost
    cost += is_interactive(pid) ? 100 : 0; // Interactive = higher cost

    return cost;
}

// Incremental termination
void recover_by_incremental_termination() {
    while (detect_deadlock()) {
        int victim = select_victim_process(get_deadlocked_processes());

        if (victim == -1) {
            printf("No victim found!\n");
            break;
        }

        printf("Terminating process P%d to break deadlock\n", victim);
        kill_process(victim);
        release_all_resources(victim);
    }
}
```

Method 2: Resource Preemption

```c
// Preempt resources from processes
void recover_by_preemption() {
    int deadlocked_processes[] = get_deadlocked_processes();

    for (int i = 0; deadlocked_processes[i] != -1; i++) {
        int pid = deadlocked_processes[i];

        // Select resources to preempt
        int resources_to_preempt[] = select_resources_for_preemption(pid);

        // Preempt resources
        for (int r = 0; resources_to_preempt[r] != -1; r++) {
            int resource = resources_to_preempt[r];
```

```
            // Save process state
            save_process_state(pid, resource);

            // Preempt resource
            preempt_resource(pid, resource);

            printf("Preempted resource R%d from process P%d\n", resource,
pid);
        }

        // Rollback process to safe state
        rollback_process(pid);
    }
}

// Select which resources to preempt
int* select_resources_for_preemption(int pid) {
    static int resources[10];
    int count = 0;

    // Preempt resources based on:
    // - Cost of rollback
    // - Number of resources held
    // - Process priority

    int held_resources[] = get_held_resources(pid);

    for (int i = 0; held_resources[i] != -1 && count < 10; i++) {
        if (should_preempt(pid, held_resources[i])) {
            resources[count++] = held_resources[i];
        }
    }

    resources[count] = -1;  // End marker
    return resources;
}

// Rollback process to previous safe state
void rollback_process(int pid) {
    // Restore process state from checkpoint
    restore_from_checkpoint(pid);

    // Restart process from safe point
    restart_process(pid);
```

```
    printf("Process P%d rolled back to safe state\n", pid);
}
```

Recovery Considerations:

Process Termination:

- Simple to implement
- Fast recovery
- Loss of computation
- May require user intervention

Resource Preemption:

- Less wasteful than termination
- Can preserve some work
- Complex implementation
- May cause starvation
- Rollback overhead

Starvation Prevention:

```
// Prevent same process from always being selected as victim
int select_victim_with_aging(int deadlocked_processes[]) {
    int victim = -1;
    int min_cost = INT_MAX;

    for (int i = 0; deadlocked_processes[i] != -1; i++) {
        int pid = deadlocked_processes[i];

        int cost = calculate_termination_cost(pid);

        // Add penalty for processes that were victims before
        int victim_count = get_previous_victim_count(pid);
        cost += victim_count * 1000;  // Increase cost

        if (cost < min_cost) {
            min_cost = cost;
            victim = pid;
        }
```

```
    }

    return victim;
}
```

4.6 Comparison of Deadlock Handling Methods

Method	Advantages	Disadvantages	Use Cases
Prevention	Simple, guaranteed no deadlock	Low resource utilization, restrictive	Real-time systems, embedded systems
Avoidance (Banker's)	More concurrency, no deadlock	High overhead, needs max claim info	Batch systems, database systems
Detection & Recovery	High utilization, flexible overhead	Deadlock can occur, recovery cost	General-purpose OS, interactive systems
Ignore (Ostrich)	No overhead, simple	System can hang	Personal computers, non-critical systems

4.7 Real-World Implementations

Operating Systems:

1. **Windows**: Uses detection and recovery for some resources
2. **Linux**: Mostly ignores deadlock (Ostrich algorithm) for performance
3. **Database Systems**: Use deadlock detection with timeout and victim selection

Database Deadlock Handling:

```
-- Database systems use timeout and victim selection
-- Example: SQL Server deadlock detection

-- Set deadlock priority
SET DEADLOCK_PRIORITY LOW;

-- Transaction with deadlock handling
BEGIN TRY
    BEGIN TRANSACTION;

    -- Operations that might deadlock
    UPDATE Accounts SET Balance = Balance - 100 WHERE Id = 1;
```

```
    UPDATE Accounts SET Balance = Balance + 100 WHERE Id = 2;

    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF (XACT_STATE()) <> 0
        ROLLBACK TRANSACTION;

    -- Check for deadlock
    IF (ERROR_NUMBER() = 1205)  -- Deadlock error
        PRINT 'Deadlock detected. Transaction rolled back.';
END CATCH
```