Lab 7: Banker's Algorithm

CS363 • Operating System (Lab)

Deadlocks

Definition

A **deadlock** is a situation in a multiprogramming environment where several processes compete for a finite number of resources. A process requests resources. If unavailable, the process enters a waiting state. A waiting process might never change state because the resources it has requested are held by other waiting processes.

System Model and Resources

A system consists of a finite number of resources distributed among competing processes. Resources are partitioned into several types, each consisting of identical instances.

Examples: Memory space, CPU cycles, files, I/O devices, printers, DVD drives.

Under normal operation:

1. **Request:** The process requests the resource.
2. **Use:** The process operates on the resource.
3. **Release:** The process releases the resource.

Deadlock Characterization (Necessary Conditions)

A deadlock may arise if all four conditions hold simultaneously:

1. **Mutual Exclusion:** At least one resource is held in a non-sharable mode.
2. **Hold and Wait:** A process holds at least one resource while waiting for others.
3. **No Preemption:** Resources cannot be forcibly taken away.
4. **Circular Wait:** A circular chain of processes exists, each waiting for a resource held by the next.

Resource-Allocation Graph (RAG)

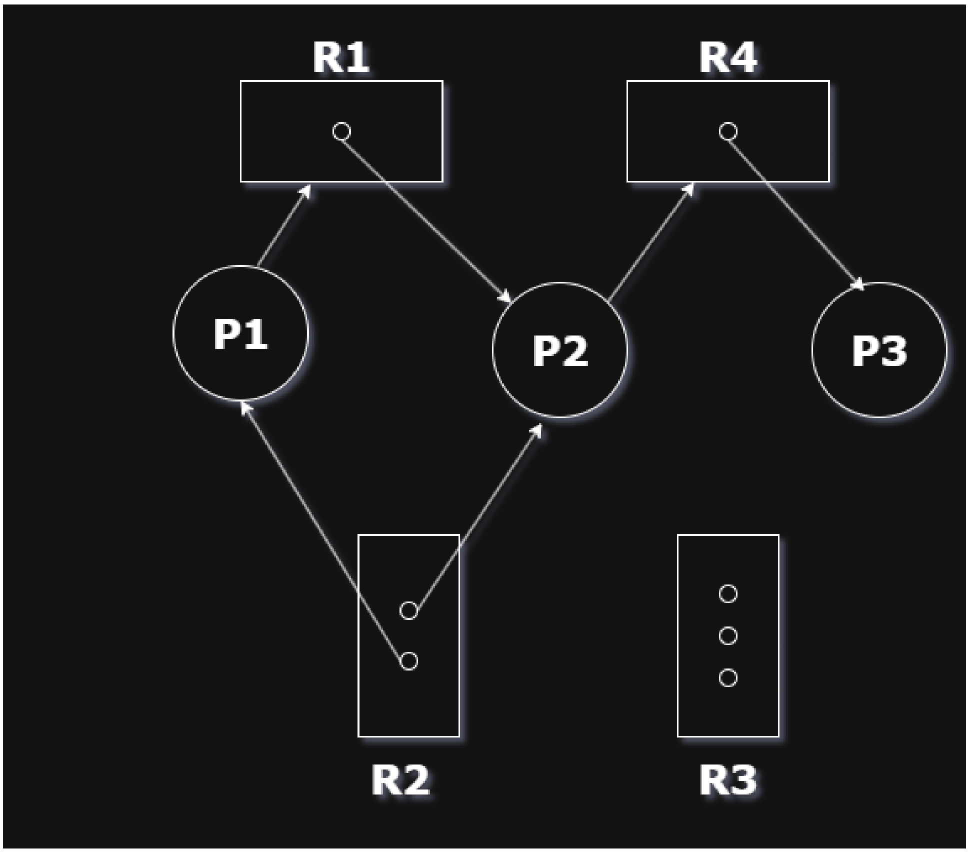

Figure 1: Example of a Resource Allocation Graph (RAG).

Vertices: Processes $P_1, P_2, \ldots$ and resources $R_1, R_2, \ldots$

Request edge: $P_i \to R_j$ means P_i requested R_j.

Assignment edge: $R_j \to P_i$ means R_j is allocated to P_i.

If the graph has a cycle, a deadlock *may* exist.

Methods for Handling Deadlocks

1. **Prevention:** Ensure at least one of the four conditions cannot hold.
2. **Avoidance:** Use algorithms like Banker's to avoid unsafe states.
3. **Detection and Recovery:** Allow deadlock, then detect and recover.

Deadlock Prevention Summary

- **Mutual Exclusion:** Needed for non-sharable resources.
- **Hold and Wait:** Request all resources at once or release before re-requesting.
- **No Preemption:** Preempt resources from waiting processes if possible.
- **Circular Wait:** Impose total ordering on resource types.

Deadlock Avoidance

Deadlock avoidance requires advance knowledge of resource requests. A system state is **safe** if resources can be allocated in some order (safe sequence) so that all processes finish without deadlock.

Banker's Algorithm

An avoidance algorithm that ensures the system stays in a safe state.

Data Structures

For n processes and m resource types:

- `Available[m]` — resources currently available.
- `Max[n][m]` — maximum resources each process may request.

- `Allocation[n][m]` — resources currently allocated.
- `Need[n][m]` = `Max - Allocation`

Safety Algorithm

1. Initialize: `Work = Available`, `Finish[i] = false`.
2. Find i such that `Finish[i] == false` and `Need[i]` `Work`.
3. If found, `Work = Work + Allocation[i]`, `Finish[i] = true`.
4. Repeat until all Finish[i] = true. Otherwise, unsafe.

Resource Request Algorithm

When process P_i makes a request `Request[i]`:

1. If `Request[i] > Need[i]`, raise an error.
2. If `Request[i] > Available`, process waits.
3. Otherwise, pretend to allocate and test for safety.

Example C Program for Banker's Algorithm

Listing 1: Banker's Algorithm in C

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

int main() {
    int n, m; // n = processes, m = resources
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &m);

```

```
    int alloc[n][m], max[n][m], avail[m];
    int need[n][m], finish[n], safeSeq[n];

    printf("\nEnter Allocation Matrix (%d x %d):\n", n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);

    printf("\nEnter Max Matrix (%d x %d):\n", n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &max[i][j]);

    printf("\nEnter Available Resources (%d values):\n", m);
    for (int i = 0; i < m; i++)
        scanf("%d", &avail[i]);

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];

    for (int i = 0; i < n; i++)
        finish[i] = 0;

    int count = 0;

    while (count < n) {
        bool found = false;
        for (int p = 0; p < n; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < m; j++)
                    if (need[p][j] > avail[j])
                        break;
```

```c
                    if (j == m) {
                        for (int k = 0; k < m; k++)
                            avail[k] += alloc[p][k];
                        safeSeq[count++] = p;
                        finish[p] = 1;
                        found = true;
                    }
                }
            }
            if (!found) {
                printf("\nSystem is in UNSAFE state!\n");
                return 0;
            }
        }

    printf("\nSystem is in SAFE state.\nSafe sequence is: ");
    for (int i = 0; i < n; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");
    return 0;
}
```

Output:

```
System is in SAFE state.
Safe sequence is: P0 P3 P4 P1 P2
```

Deadlock Detection and Recovery

Detection (Single Instance)

Use a **Wait-for Graph**. Deadlock exists iff the graph contains a cycle.

Detection (Multiple Instances)

Algorithm uses `Available`, `Allocation`, and `Request` matrices. A process is deadlocked if `Finish[i] = false` after the detection loop.

Recovery

Two methods:

- **Process Termination:** Abort all or selected processes.
- **Resource Preemption:** Reclaim resources from some processes.