OPERATING SYSTEMS & CONCURRENT PROGRAMMING IN ACTION

Part 6 of 11: The Banker's Algorithm

Welcome to **Part 6** of our seminar, *"Operating Systems & Concurrent Programming in Action."* Today, we bring together the theoretical framework **and** a simplified Bash script implementation of the **Banker's Algorithm**. You'll see how the conceptual logic of this deadlock-avoidance approach (pseudocode) connects seamlessly to a small example in a shell script—offering a beginner-friendly demonstration of the core ideas.

All relevant scripts, including the one shown here, are publicly accessible in our GitHub repository at https://github.com/antonioclim/OS5. Explanations are crafted "for newbies," ensuring the fundamentals are clear while preserving the formal structure needed in academic and real-world scenarios.

---

1. REVISITING THE BANKER'S ALGORITHM

1.1 Conceptual Overview

The **Banker's Algorithm**, created by Edsger Dijkstra, is a deadlock-avoidance strategy designed to keep resource allocations in a **safe** state. Think of an operating system as a bank with limited capital (resources), processes as customers seeking loans, and each process's request as a resource demand. This analogy underpins the name "Banker's" Algorithm.

1. **Safe State**: The OS (bank) can allocate resources in such a way that all processes (customers) can eventually acquire what they need and complete, returning resources to the system.
2. **Unsafe State**: Resource demands could be allocated in a way that leads to potential deadlocks, meaning at least one process may never complete.

By checking each request for safety before finalizing allocation, the system avoids conditions where processes hold resources indefinitely and block further progress.

---

2. CLASSIC PSEUDOCODE

The Banker's Algorithm is often divided into two parts:

1. **Safety Algorithm** – confirms whether a current or hypothetical allocation leaves the system in a safe state.
2. **Resource-Request Algorithm** – decides if a particular request can be granted immediately or must be deferred/denied to maintain safety.

Below is the *textbook-style* pseudocode.

2.1 Safety Check

```
function SafetyCheck() returns bool
   Work[ m ] = Available[]      // temporary copy
   Finish[ n ] = false          // initially, no process can finish

   repeat
      found = false
      for i in 0..(n-1)
         if (Finish[i] == false) AND (Need[i] <= Work) then
            // pretend process i finishes, returns resources
            Work = Work + Allocation[i]
            Finish[i] = true
            found = true
      end for
   until (found == false)
```

```
// if all processes are marked finish, system is safe
for i in 0..(n-1)
    if (Finish[i] == false)
        return false
return true
```

2.2 Resource-Request Handling

```
function RequestResources(process i, Request[i]) returns bool
    // A. Validate request
    if (Request[i] > Need[i]) then
        error("Request exceeds the maximum declared Need of process i")

    // B. Check resource availability
    if (Request[i] > Available) then
        block(process i) // must wait, insufficient resources

    // C. Tentative allocation
    Available    = Available - Request[i]
    Allocation[i]= Allocation[i] + Request[i]
    Need[i]      = Need[i] - Request[i]

    // D. Check safety with new allocation
    if SafetyCheck() == true
        return true // request granted
    else
        // revert to previous state
        Available    = Available + Request[i]
        Allocation[i]= Allocation[i] - Request[i]
        Need[i]      = Need[i] + Request[i]
        return false // request denied
```

---

3. SIMPLIFIED BASH SCRIPT EXAMPLE

While real operating systems implement the Banker's Algorithm in kernel-level resource managers, we can *simulate* it in Bash to demonstrate the **core principle** of checking for safe states. Below is a single script that:

1. Loads resource data from text files,
2. Maintains arrays for Available, Allocation, Max (and thus Need),
3. Performs a minimal "safety check,"
4. Grants or denies resource requests in a manner reminiscent of the Banker's Algorithm.

Disclaimer: This script is purely educational. Bash lacks direct concurrency primitives, so you'll see manual arrays and file reads. Yet it effectively conveys the step-by-step logic.

3.1 Directory & Data

Suppose we have these files under ~/OS5/bankers/:

- **available.txt**: A single line with comma-separated values for each resource type's available count.
- **max.txt**: Multi-line CSV – each line is a process's maximum demand.
- **allocation.txt**: Multi-line CSV – each line is the current allocation for that process.

3.2 bankers.sh

```
#!/usr/bin/env bash
#
# bankers.sh - A simplified Banker's Algorithm example in Bash
#
```

```
DATA_DIR="$HOME/OS5/bankers"

# load_csv: read a single-line CSV into a Bash array
load_csv() {
   local file="$1"
   local -n arr_ref="$2"
   IFS=',' read -r -a arr_ref < "$file"
}

# load_matrix: read multi-line CSV into a 2D array (emulated)
load_matrix() {
   local file="$1"
   local -n matrix_ref="$2"
   local i=0

   while IFS=',' read -r a b c; do
      matrix_ref[$((i*3 + 0))]=$a
      matrix_ref[$((i*3 + 1))]=$b
      matrix_ref[$((i*3 + 2))]=$c
      ((i++))
   done < "$file"
}

# Arrays to store data
declare -a AVAILABLE
declare -a MAX_ARR
declare -a ALLOC_ARR
declare -a NEED_ARR
n=3   # number of processes
m=3   # resource types

# 1) Load data from CSV
load_csv    "$DATA_DIR/available.txt"    AVAILABLE
load_matrix "$DATA_DIR/max.txt"          MAX_ARR
load_matrix "$DATA_DIR/allocation.txt"   ALLOC_ARR

# 2) Compute NEED = MAX - ALLOCATION
compute_need() {
   for ((i=0; i<n; i++)); do
      for ((j=0; j<m; j++)); do
         idx=$((i*m + j))
         NEED_ARR[$idx]=$(( ${MAX_ARR[$idx]} - ${ALLOC_ARR[$idx]} ))
      done
   done
}
compute_need

# 3) Safety check function
is_system_safe() {
   # local copies
   local WORK=("${AVAILABLE[@]}")
   local FINISH=()
   for ((i=0; i<n; i++)); do
      FINISH[$i]=false
   done

   local progress=true
   while $progress; do
      progress=false
      for ((i=0; i<n; i++)); do
         if [ "${FINISH[$i]}" = false ]; then
            # check if NEED <= WORK
            local can_finish=true
            for ((j=0; j<m; j++)); do
               idx=$((i*m + j))
```

```
            if [ ${NEED_ARR[$idx]} -gt ${WORK[$j]} ]; then
              can_finish=false
              break
            fi
          done
          if $can_finish; then
            # pretend process i finishes => release resources
            for ((j=0; j<m; j++)); do
              idx=$((i*m + j))
              WORK[$j]=$(( ${WORK[$j]} + ${ALLOC_ARR[$idx]} ))
            done
            FINISH[$i]=true
            progress=true
          fi
        fi
      done
    done

    # confirm if all are FINISH=true
    for ((i=0; i<n; i++)); do
      if [ "${FINISH[$i]}" = false ]; then
        return 1 # not safe
      fi
    done
    return 0 # safe
}

# 4) Resource-Request function
request_resources() {
    local p="$1"
    local reqA="$2"
    local reqB="$3"
    local reqC="$4"
    echo "Process $p requests ($reqA, $reqB, $reqC)"

    # Basic indexing
    idxA=$((p*m + 0))
    idxB=$((p*m + 1))
    idxC=$((p*m + 2))

    # Check against Need
    if [ $reqA -gt ${NEED_ARR[$idxA]} ] || \
       [ $reqB -gt ${NEED_ARR[$idxB]} ] || \
       [ $reqC -gt ${NEED_ARR[$idxC]} ]; then
       echo "Error: Request exceeds process $p's declared maximum need!"
       return 1
    fi

    # Check if enough available
    if [ $reqA -gt ${AVAILABLE[0]} ] || \
       [ $reqB -gt ${AVAILABLE[1]} ] || \
       [ $reqC -gt ${AVAILABLE[2]} ]; then
       echo "Process $p must wait, not enough resources."
       return 1
    fi

    # Tentative allocation
    AVAILABLE[0]=$(( ${AVAILABLE[0]} - reqA ))
    AVAILABLE[1]=$(( ${AVAILABLE[1]} - reqB ))
    AVAILABLE[2]=$(( ${AVAILABLE[2]} - reqC ))

    ALLOC_ARR[$idxA]=$(( ${ALLOC_ARR[$idxA]} + reqA ))
    ALLOC_ARR[$idxB]=$(( ${ALLOC_ARR[$idxB]} + reqB ))
    ALLOC_ARR[$idxC]=$(( ${ALLOC_ARR[$idxC]} + reqC ))
```

```
    NEED_ARR[$idxA]=$(( ${NEED_ARR[$idxA]} - reqA ))
    NEED_ARR[$idxB]=$(( ${NEED_ARR[$idxB]} - reqB ))
    NEED_ARR[$idxC]=$(( ${NEED_ARR[$idxC]} - reqC ))

    # Check system safety now
    if is_system_safe; then
        echo "Request granted (system remains safe)."
        return 0
    else
        echo "Request denied (would leave system unsafe), rolling back."

        # Roll back
        AVAILABLE[0]=$(( ${AVAILABLE[0]} + reqA ))
        AVAILABLE[1]=$(( ${AVAILABLE[1]} + reqB ))
        AVAILABLE[2]=$(( ${AVAILABLE[2]} + reqC ))

        ALLOC_ARR[$idxA]=$(( ${ALLOC_ARR[$idxA]} - reqA ))
        ALLOC_ARR[$idxB]=$(( ${ALLOC_ARR[$idxB]} - reqB ))
        ALLOC_ARR[$idxC]=$(( ${ALLOC_ARR[$idxC]} - reqC ))

        NEED_ARR[$idxA]=$(( ${NEED_ARR[$idxA]} + reqA ))
        NEED_ARR[$idxB]=$(( ${NEED_ARR[$idxB]} + reqB ))
        NEED_ARR[$idxC]=$(( ${NEED_ARR[$idxC]} + reqC ))
        return 1
    fi
}

# Example usage
echo "Starting State: Available=(${AVAILABLE[@]})"
echo "Allocation=(${ALLOC_ARR[@]})"
echo "Need=(${NEED_ARR[@]})"

# Suppose process 0 requests (1,2,0)
request_resources 0 1 2 0

# Suppose process 1 requests (1,0,1)
request_resources 1 1 0 1

echo "Final: Available=(${AVAILABLE[@]})"
echo "Alloc=(${ALLOC_ARR[@]})"
echo "Need=(${NEED_ARR[@]})"
```

Running:

1. Ensure you have available.txt, max.txt, and allocation.txt under ~/OS5/bankers/.
2. Make the script executable: chmod +x bankers.sh.
3. Execute: ./bankers.sh. Watch the script indicate if resource requests are granted or denied.

---

4. STRENGTHS & LIMITATIONS

- **Advantages**:
    - Ensures no deadlocks by design, checking each request.
    - Works with multiple resource types.
    - Particularly effective when maximum demands are well-known.
- **Drawbacks**:
    - Requires overhead in checking safety after each request.
    - Might be *too* conservative, occasionally denying feasible requests.

- Not widely used in general-purpose OS for dynamic workloads but remains vital in safety-critical & real-time contexts.

---

5. RECAP & LOOKING FORWARD

The Banker's Algorithm is a **cornerstone** of deadlock avoidance theory. While it may seem "old," its logic is found in resource-limiting strategies used across modern systems, especially those requiring absolute reliability. The pseudocode clarifies each step—ensuring no process request results in an **unsafe** state—while our Bash script example offers a tangible glimpse of how one might code these concepts in practice.

REFERENCES

- Dijkstra, E. W. (1965). *Cooperating Sequential Processes.* In F. Genuys (Ed.), *Programming Languages*. Academic Press.
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts (10th ed.).* Wiley. https://doi.org/10.1002/9781119320913
- Courtois, P. J., Heymans, F., & Parnas, D. L. (1971). Concurrent Control with Readers and Writers. *Communications of the ACM, 14*(10), 667–668. https://doi.org/10.1145/362919.362945
