OPERATING SYSTEMS & CONCURRENT PROGRAMMING IN ACTION

Part 9 of 11: Resource Allocation & System Safety

In our ninth session, we address **resource allocation** and **system safety**—two vital concepts in ensuring robust, stable, and deadlock-free operation of computer systems. We'll explore how the **Banker's Algorithm** keeps resource distribution safe and learn the basics of **system monitoring** (CPU, memory, disk usage, and zombie process cleanup) to maintain overall system health. We'll also reference relevant **Bash scripts** from https://github.com/antonioclim/OS5.

---

1. RESOURCE ALLOCATION FUNDAMENTALS

1.1 Importance

An operating system manages a finite set of resources—CPU time, memory, storage, devices—that multiple processes compete to use. Efficient allocation ensures:

- **No Starvation**: Processes eventually receive the resources they need.
- **No Overcommitment**: The OS avoids promising more resources than it can safely provide.

1.2 Unsafe vs. Safe States

A **safe** state is one where the system can allocate resources to each process in some order without risking deadlock. Conversely, an **unsafe** state may not immediately be a deadlock, but the system can't guarantee a safe execution sequence if more allocations occur.

---

2. BANKER'S ALGORITHM

2.1 Overview

Originally devised by Edsger Dijkstra, the **Banker's Algorithm** treats processes like "customers" with maximum demands, and system resources like the "bank's capital." Before fulfilling each request, the system checks if doing so will keep it in a safe state. If not, the request is temporarily denied.

Key data structures:

- Available[]: How many instances of each resource type are unallocated.
- Max[process][resource]: Maximum demand a process can ever request.
- Allocation[process][resource]: Current allocation to each process.
- Need[process][resource]: The remainder of a process's required resources (Max - Allocation).

2.2 Classic Pseudocode

```
function RequestResources(process p, Request[p]):
  # Step A: Validate
  if Request[p] > Need[p]:
    error("Request exceeds declared maximum")

  # Step B: Check availability
  if Request[p] > Available:
    block process p   # insufficient resources

  # Step C: Tentative allocation
  Available = Available - Request[p]
  Allocation[p] = Allocation[p] + Request[p]
```

```
  Need[p] = Need[p] - Request[p]

  # Step D: Safety check
  if SafetyAlgorithm() == true:
     # remain allocated
     return true
  else:
     # revert
     Available = Available + Request[p]
     Allocation[p] = Allocation[p] - Request[p]
     Need[p] = Need[p] + Request[p]
     return false
```

2.3 Bash Script Demo: 12BANKER_algorithmV2real.sh

What It Does:

1. **Loads** arrays for Available, Max, and Allocation from text files (similar to previous examples).
2. **Computes** each process's Need.
3. **Implements** the request logic—tentatively assigning resources and then checking if the system stays safe.
4. **Grants or denies** the request accordingly.

Snippet (conceptual):

```
# Hypothetical resource request function
request_resources() {
   local process_id=$1
   local reqA=$2
   local reqB=$3
   # 1) Check Request vs Need
   # 2) Check vs Available
   # 3) Tentatively allocate
   # 4) Safety check -> if unsafe, revert
}
```

When you run the script, you'll see whether each request is granted ("System remains safe") or denied ("Would leave system unsafe").

---

3. SYSTEM MONITORING & SAFETY

While the Banker's Algorithm prevents unsafe allocations, **system monitoring** ensures that even correct allocations aren't overshadowed by resource mismanagement or rogue processes.

3.1 Monitoring Key Metrics

1. **CPU Usage** – Identify if processes hog CPU or if load is extremely high.
2. **Memory Usage** – Spot potential leaks or memory saturation.
3. **Disk Usage** – Prevent out-of-space conditions that degrade performance or cause failures.
4. **Zombie Cleanup** – Zombie processes, having finished execution but not removed from process tables, occupy resource slots unnecessarily.

3.2 Bash Script Demo: 9OSresource_allocationV1REAL.sh

What It Does:

1. **CPU Load**: Reads from **/proc/loadavg** or uses commands like **uptime**.
2. **Memory**: Uses **free -h** or other references to show usage.

3. **Disk**: Uses **df -h**.
4. **Zombie Detection**: Scans processes with ps axo stat,ppid,pid,comm | grep -w defunct and attempts to clean them up.

Snippet:

```
monitor_resources() {
  echo "===== Resource Report ====="
  echo "CPU Load Average:"
  cat /proc/loadavg
  echo "Memory:"
  free -h
  echo "Disk:"
  df -h | grep -v tmpfs

  # Checking zombies
  zombies=$(ps axo stat,ppid,pid,comm | grep -w defunct)
  if [ -n "$zombies" ]; then
    echo "Found zombie processes:"
    # Attempt to kill or signal
  fi
}
```

Regularly running such a script or scheduling it via cron ensures the system remains stable, clearing out any resource-consuming anomalies.

---

4. CONCLUDING INSIGHTS

- **Resource allocation** is central to OS design: must be done carefully to avoid blocking or unsafe states.
- **Banker's Algorithm** is a direct solution for ensuring the system can handle maximum demands without deadlock.
- **System monitoring** addresses real-time usage, highlighting possible resource misallocations, memory leaks, or rogue processes.

Learning to combine these strategies—safe resource allocation plus thorough system monitoring—builds the backbone of a robust, deadlock-free, and efficiently managed operating system environment.

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.
- Scripts on GitHub: https://github.com/antonioclim/OS5
