OPERATING SYSTEMS & CONCURRENT PROGRAMMING IN ACTION

Part 8 of 11: Deadlocks—Occurrence, Detection, and Prevention

---

In this eighth part of our seminar, we focus on **deadlocks**, a notorious pitfall in concurrent computing. We'll explore what deadlocks are, how they form, and practical steps to detect and avoid them. By referencing **Bash scripts** from https://github.com/antonioclim/OS5, we'll see real-world examples that illustrate these principles "like for newbies." Where helpful, we'll include small pseudocodes and Bash snippets.

---

1. WHAT IS A DEADLOCK?

A **deadlock** is a condition in which two or more processes become permanently blocked, each waiting for the other to release resources. No progress can be made, leading to a system impasse. An everyday analogy is two people crossing a narrow hallway from opposite ends. Each politely waits for the other to move first, so neither proceeds—indefinitely.

---

2. FOUR CONDITIONS FOR DEADLOCK

Deadlocks occur if and only if these four conditions hold simultaneously:

1. **Mutual Exclusion**
   Some resources are non-shareable—only one process can use them at a time.
2. **Hold and Wait**
   A process holding at least one resource can request additional resources without releasing its current holdings.
3. **No Preemption**
   Resources cannot be forcibly taken from processes; they release resources voluntarily.
4. **Circular Wait**
   A chain of processes exists where each process holds a resource the next one needs, forming a cycle of waiting.

Breaking any one condition prevents deadlocks from forming.

---

3. DEMONSTRATIONS & SCRIPTS

We illustrate three main angles of deadlock handling:

1. **Occurrence** – showing how two processes can get stuck.
2. **Detection** – scanning for locked resources.
3. **Prevention** – using strategies like timed locking to avoid indefinite waits.

3.1 Deadlock Occurrence: 10deadlock_occurenceV1.sh

Pseudocode for a minimal deadlock scenario with two processes (P1, P2) and two resources (R1, R2) might look like this:

```
Process P1:
  acquire_lock(R1)
  sleep(1)
  acquire_lock(R2)
  # stuck if P2 has R2
```

```
Process P2:
   acquire_lock(R2)
   sleep(1)
   acquire_lock(R1)
   # stuck if P1 has R1
```

Why it deadlocks: Each process holds one resource and waits for the other. Because neither releases what it already has, they block each other forever.

Bash snippet (conceptually similar):

```
# P1
exec 200>"/tmp/R1.lock"  # lock for resource R1
flock -n 200 || exit 1
echo "P1 acquired R1"
sleep 1

exec 201>"/tmp/R2.lock"
flock -n 201 || exit 1
echo "P1 acquired R2"
# ... never reached if P2 holds R2

# P2 similarly does R2 first, then R1
```

When run in parallel, the scripts can get stuck, thus **demonstrating** a real deadlock.

3.2 Deadlock Detection: 11deadlocks_conditionsV1real.sh

Once a system is suspected of being in deadlock, **detection** is crucial. A typical approach is to list which processes hold which locks and see if a **circular wait** arises.

Pseudo-steps for detection:

```
for each resource R:
  find which process holds R
  find which process wants R
look for a cycle in the process-resource graph
```

In **Bash**, a practical method includes:

- Using **lsof** to see open files and the processes that hold them.
- Checking for cyclical dependencies in the output.

Snippet:

# A script might run:

```
lsof /tmp/*.lock > locks_info.txt
```

# parse locks_info.txt to see if processes block each other

If we find a cycle (P1 → R1 → P2 → R2 → P1), we have a deadlock.

3.3 Deadlock Prevention: *10deadlock_occurenceV2resolved.sh*

Prevention is typically safer than detection + recovery. One strategy is **timed lock acquisition**: if a process cannot acquire a lock within a certain time, it releases any locks it holds, waits, and retries. This avoids indefinite waiting.

Pseudo-steps for timed lock approach:

```
function timed_lock(resource R, timeout T):
   attempt to flock(R) with a timeout of T
   if success:
    return LOCK_ACQUIRED
  else:
    # if timed out
    release any other locks
    wait random interval
    retry
```

Bash snippet (conceptual example):

```
acquire_lock() {
 local lockfile="$1"
 exec {fd}>$lockfile
 if ! flock -w 5 $fd; then
    echo "Timed out. Releasing resources and retrying..."
    release_all_locks
    sleep $((RANDOM % 3 + 1))
    # possibly retry here
    return 1
 fi
 echo "Lock on $lockfile acquired."
}
```

By avoiding indefinite blocking, we break the **hold and wait** or **circular wait** conditions, drastically reducing the chance of deadlock.

OPTIONAL MENTION: *7deadlockV2firsttorun.sh*

This **initialization script** can set up lock files or environment variables. While not complex, it highlights the importance of correct resource initialization—if the system incorrectly configures locks or metadata, subsequent scripts might fail or lead to unintentional concurrency issues.

---

4. SUMMARY

- **Deadlocks** halt system progress when processes hold resources each other needs.
- **Conditions** for deadlock: mutual exclusion, hold and wait, no preemption, and circular wait.
- **Occurrences** can be replicated in Bash scripts, e.g., 10deadlock_occurenceV1.sh.
- **Detection** can be automated with lsof or resource-graph analysis (11deadlocks_conditionsV1real.sh).
- **Prevention** might use timed locks or other strategies to break one of the deadlock conditions (10deadlock_occurenceV2resolved.sh).

Armed with these concepts and example scripts, you can experiment and see how real concurrency issues arise and how to address them effectively.

REFERENCES

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
- Coffman, E. G., Elphick, M. J., & Shoshani, A. (1971). System Deadlocks. *ACM Computing Surveys, 3*(2), 67–78.
- Scripts on GitHub at https://github.com/antonioclim/OS5.

End of Part 8 of 11