OPERATING SYSTEMS & CONCURRENT PROGRAMMING IN ACTION

Part 7 of 11: Concurrent Programming & Synchronization

Welcome to **Part 7** of our seminar series, *"Operating Systems & Concurrent Programming in Action."* Today, we'll examine **concurrency** and **synchronization** via **three** of the most recognised concurrency scenarios: the **Dining Philosophers**, **Producer–Consumer**, and **Readers–Writers** problems. Alongside concise *pseudocode* demonstrating each scenario's core logic, we'll reference their **Bash scripts**—all found at https://github.com/antonioclim/OS5—which use flock to manage file-based locking. These script-based examples show how "newbies" can easily test out concurrency in a Bash environment.

---

1. CONCURRENCY & SYNCHRONIZATION

1.1 Concurrency

Modern systems run multiple processes **simultaneously** to better utilize CPU and I/O resources. While concurrency boosts **throughput** and **responsiveness**, it also poses the problem of **shared resource conflicts**.

1.2 Synchronization

To prevent race conditions and data corruption, concurrency demands **synchronization**—ensuring processes coordinate resource usage safely. In Bash, a common approach uses file locking (flock) to control exclusive or shared access to files that represent resources.

---

2. DINING PHILOSOPHERS

2.1 Pseudocode

```
# N philosophers, N forks. Each philosopher picks up two forks to eat.

repeat forever:
   think()
   pick_up_fork(i)      # philosopher i picks left fork
   pick_up_fork((i+1)%N) # philosopher i picks right fork
   eat()
   put_down_fork(i)
   put_down_fork((i+1)%N)
```

Key Issue: If all philosophers pick up the left fork first, a deadlock can occur when each is waiting for a second fork. Solutions typically impose an ordering or a lock strategy to avoid circular waits.

2.2 Bash Implementation

Filename: 1dining_Philosophers.sh (GitHub link)

- **Forks** are mapped to lock files (e.g., /tmp/fork0.lock, /tmp/fork1.lock, etc.).
- Each philosopher uses flock -x to obtain an exclusive lock for each "fork file."
- Once done, the philosopher releases the locks, preventing indefinite resource holding.

Snippet:

```
acquire_fork() {
   exec {fd}>"/tmp/fork$1.lock"
   flock -x "$fd"
   echo "Philosopher $2 has acquired fork $1"
}

release_fork() {
```

```
    flock -u "$fd"
    exec {fd}>&-
    echo "Philosopher $1 has released their forks"
}
```

By carefully ordering how forks are picked up, the script avoids deadlock.

---

3. PRODUCER–CONSUMER

3.1 Pseudocode

```
# Producer and Consumer share a finite buffer of capacity MAX.

Producer:
  loop forever:
    produce(item)
    while buffer is full:
      wait
    add item to buffer
    signal consumer

Consumer:
  loop forever:
    while buffer is empty:
      wait
    remove item from buffer
    signal producer
    consume(item)
```

Goal: Ensure producers don't overfill the buffer, and consumers don't read from an empty buffer. They must coordinate usage of the shared buffer.

3.2 Bash Implementation

Filename: 2Producer-ConsumerLIST.sh (GitHub link)

- **Buffer** is a directory in the file system.
- **Producer** creates files (items) if there's space.
- **Consumer** removes files if any exist.
- flock ensures only one process modifies the directory at a time, preventing concurrent writes or reads that lead to corruption.

Snippet:

```
# Producer-like function
while true; do
    count=$(ls "$buffer_dir" | wc -l)
    if [[ $count -lt $MAX_BUFFER_SIZE ]]; then
        touch "$buffer_dir/item_$RANDOM"
        echo "Produced an item"
    else
        echo "Buffer is full, waiting..."
    fi
    sleep 1
done

# Consumer-like function
while true; do
    items=( $(ls "$buffer_dir") )
    if [ ${#items[@]} -gt 0 ]; then
        rm "$buffer_dir/${items[0]}"
        echo "Consumed ${items[0]}"
```

```
    else
        echo "Buffer is empty, waiting..."
    fi
    sleep 2
done
```

---

4. READERS–WRITERS

4.1 Pseudocode

```
# Readers can share access; a writer needs exclusive use.

repeat:
  if Reader:
    # Acquire shared lock
    read data
    # Release shared lock

  if Writer:
    # Acquire exclusive lock
    write data
    # Release exclusive lock
```

Challenge: Let multiple readers read simultaneously, but block writers until readers finish, and block readers when a writer has the lock.

4.2 Bash Implementation

Filename: 3Readers-WritersSEMAPHOREiniSCRIPTS.sh (GitHub link)

- **Readers** use flock -s (shared) so multiple concurrent reads are allowed.
- **Writers** use flock -x (exclusive) so no one else can read or write during an update.

Snippet:

```
# Reader
{
    flock -s 200
    echo "Reader $id is reading."
    cat datafile
    sleep 1
    echo "Reader $id finished reading."
} 200>"/tmp/data.lock"

# Writer
{
    flock -x 200
    echo "Writer $id is writing..."
    echo "New line from writer $id" >> datafile
    sleep 2
    echo "Writer $id done."
} 200>"/tmp/data.lock"
```

---

5. BRINGING IT ALL TOGETHER

1. **Dining Philosophers**: Demonstrates deadlock avoidance in a scenario of multiple resources.
2. **Producer–Consumer**: Emphasizes controlling buffer capacity and ensuring correct order of production/consumption.
3. **Readers–Writers**: Balances concurrency for multiple readers with the exclusive needs of a writer.

File locking in Bash offers a simple but effective approach to ensure only one process modifies a shared resource at a time (or that multiple readers can share it without writes in progress). This usage of flock is quite educational—even though more advanced systems often employ semaphores, condition variables, and other concurrency primitives at a lower level (e.g., in C or within the OS kernel).

---

6. NEXT STEPS

- **Practice**: Clone or download the scripts from https://github.com/antonioclim/OS5.
- **Experiment**: Tweak parameters (like number of philosophers or buffer size) and see how concurrency management changes.
- **Link**: Connect these examples to higher-level concurrency or scheduling algorithms (like the Banker's Algorithm or advanced locking strategies).

REFERENCES

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
- **GitHub Scripts** for these demos: https://github.com/antonioclim/OS5

End of Part 7 of 11