OPERATING SYSTEMS & CONCURRENT PROGRAMMING IN ACTION

Part 5 of 11: The Sleeping Barber Problem

Welcome to Part 5 of our seminar series, *"Operating Systems & Concurrent Programming in Action."* Today, we explore the **Sleeping Barber Problem**, a charming yet powerful illustration of concurrency and resource management. As always, we provide examples of real Bash scripts from our repository at https://github.com/antonioclim/OS5. We aim to explain these principles "like for newbies," ensuring everyone can follow along and appreciate the underlying concepts.

---

1. A QUICK OVERVIEW

The Setting

Envision a small barbershop with exactly one barber, one barber's chair, and a limited number of waiting seats. Customers (like "processes" requesting service) enter sporadically. The rules are:

1. If the barber is free (asleep), a new customer wakes him up and takes a seat in the barber's chair.
2. If the barber is busy, arriving customers either sit in an empty waiting seat (if available) or leave (if the waiting area is full).
3. Once the barber finishes cutting hair, if no one is waiting, he goes back to sleep. Otherwise, he beckons the next customer from the waiting area.

Why It Matters

This scenario perfectly captures **resource allocation** (the barber is the resource) and **waiting conditions** (the chairs are the waiting line). It highlights:

- How to avoid starving resources (the barber should not remain idle if there is work to do).
- How to handle tasks arriving unexpectedly and manage them in an orderly queue.
- How concurrency can be handled efficiently without chaos or indefinite waiting times.

---

2. RELATION TO OPERATING SYSTEM SCHEDULING

1. **Resource Starvation**
   The system ensures that no process (customer) waits forever. It must manage queues effectively—just like OS schedulers that manage I/O queues, CPU usage, and other shared resources.
2. **Waiting Conditions**
   If the resource (barber) is busy, tasks queue up. If the queue is too long, new tasks are turned away. Similar rules apply in OS kernels for limiting network connections or bounding CPU load.
3. **Optimal Resource Utilisation**
   We want to avoid scenarios where the "barber" is idle while requests exist. OS schedulers do the same, aiming to keep CPU usage high but also avoiding over-scheduling that can cause system thrashing.

---

3. CLASSIC APPROACHES AND SCRIPTS

3.1 Semaphore-Based Pseudocode

In many concurrency textbooks, the Sleeping Barber solution uses semaphores:

```
int waiting = 0;          // number of waiting customers
```

```
semaphore mutex = 1;       // for mutual exclusion in updating waiting count
semaphore customers = 0;   // tracks waiting customers
semaphore barber = 0;      // barber's availability

// Barber Process
while(true) {
   wait(customers);       // barber sleeps if no customers
   wait(mutex);
   waiting = waiting - 1;
   signal(barber);        // barber is now busy
   signal(mutex);
   // ... cut hair ...
}

// Customer Process
while(true) {
   wait(mutex);
   if(waiting < CHAIRS) {
      waiting = waiting + 1;
      signal(customers);  // notify the barber
      signal(mutex);
      wait(barber);      // wait to be served
      // ... get haircut ...
   } else {
      signal(mutex);
      // ... leave (too many in queue) ...
   }
}
```

3.2 Bash Scripting Example

While Bash does not provide sophisticated concurrency primitives like semaphores, we can simulate the scenario with **file locks** and conditional logic:

```
#!/bin/bash
# 5sleeping_barberZ.sh - Simplified Sleeping Barber simulation in Bash

BARBER_LOCK="/tmp/barber.lock"
CHAIRS=3
waiting=0

# This function tries to acquire an exclusive lock on the barber
function barber_sleep() {
   echo "Barber is sleeping..."
   exec {fd}>"$BARBER_LOCK"
   flock -x $fd    # Will block until lock is acquired or immediate if none
   # Once locked, barber is effectively "busy"
}

# This function releases the barber lock
function barber_release() {
   flock -u $fd
   exec {fd}>&-
   echo "Barber is done and released the lock."
}

# The "barber" loop
barber_sleep
while :  # infinite loop
```

```
do
  if (( waiting == 0 )); then
    echo "No customers waiting; barber continues sleeping..."
    sleep 2
  else
    # "Cut hair"
    echo "Barber is cutting hair for one customer..."
    sleep 3
    # once the haircut is done, reduce waiting
    (( waiting-- ))
    echo "One customer served, waiting left: $waiting"
  fi
done
```

Customer simulation:

```
#!/bin/bash
# 5sleeping_barber_customer.sh - A script that simulates a customer trying to get a haircut

BARBER_LOCK="/tmp/barber.lock"
CHAIRS=3

echo "Customer arrives..."
# If too many are waiting, leave
if (( waiting >= CHAIRS )); then
    echo "No available chairs. Customer leaves."
    exit 0
fi

echo "Customer sits and waits."
(( waiting++ ))
```

The scripts are conceptual. In reality, we'd incorporate synchronization around waiting. A more robust approach uses background processes and advanced logic with ephemeral locks or semaphores (possible with named pipes or advanced shell features).

---

4. REAL-WORLD ANALOGUES

1. **Web Server Connection Handling**
   A web server (barber) can handle only so many concurrent connections before turning away new requests (customers). If no requests arrive, the server is idle (sleeping).
2. **Thread Pool Management**
   A thread pool with a limited queue mimics chairs; tasks come in, wait if threads are busy, or get rejected if the queue is full.
3. **Database Connection Pooling**
   With limited DB connections, tasks queue. If the DB is free, tasks proceed. Otherwise, tasks wait or fail if the pool is at capacity.

---

5. KEY INSIGHTS FOR OPERATING SYSTEM DESIGN

- **Priority-Based Scheduling**: Some tasks can be given priority in the queue (analogy: letting an urgent customer cut in line).
- **Round-Robin**: In a naive approach, the barber might rotate among waiting customers in a cyclic manner.
- **Resource Allocation**: If multiple barbers exist, the system must coordinate them (like multiple CPU cores).

---

6. WHY "NEWBIES" SHOULD LEARN THE SLEEPING BARBER

It's an intuitive introduction to concurrency:

- **Covers All Major Synchronisation Topics**: waiting queues, resource constraints, and blocked states.
- **Maps Easily to Real OS Constructs**: The problem's queue is akin to a scheduler's wait queue; the barber is akin to a CPU or limited peripheral.

Studying this problem helps you grasp how concurrency issues appear in everyday computing tasks, from systemd services in Linux to background tasks in Windows or macOS.

CONCLUSION

The **Sleeping Barber Problem** offers a clear lens for viewing critical concurrency issues: resource usage, waiting queues, and strategies to prevent wasted potential or indefinite waits. Its fundamental insights—balancing supply (the barber's time) with sporadic demand (incoming customers)—apply directly to OS schedulers, server resource allocation, and countless other modern computing scenarios.

By mastering the ideas behind the Sleeping Barber, you gain a robust mental model for how operating systems manage processes, how thread pools function, and how fair scheduling is enforced. Stay tuned for the rest of our seminar, where we expand on **deadlock handling**, **system resources**, and more advanced scheduling.

REFERENCES

- Dijkstra, E. W. (1965). *Cooperating Sequential Processes*. In F. Genuys (Ed.), *Programming Languages* (pp. 43–112). Academic Press.
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley. https://doi.org/10.1002/9781119320913
- Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson. https://doi.org/10.1007/978-1-4471-5625-3
