

Assignment 2:

Concurrency and Resource Management

Name: Genaro Salazar Ruiz

Course: WES 237

Date: January 29, 2026

Github: [WES-237-Assignment-2](#)

1. Introduction & Methodology

The Goal: Implementing the Dining Philosophers problem to demonstrate thread synchronization and deadlock prevention on the PYNQ-Z2.

Top-Down Design:

1. **Resource Mapping:** Defined the 5 forks as `threading.Lock()` objects and map the 5 LEDs (4 green, 1 RGB).
2. **Thread Logic:** Created a philosopher_task that manages the three states: STARVING, EATING, and NAPPING.
3. **Deadlock Prevention:** Implemented an asymmetric lock acquisition strategy to break circular wait conditions.

2. Technical Implementation

2.1 Synchronization:

To ensure mutual exclusion, I used the `with lock:` context manager. This ensures that a resource is automatically released even if an error occurs within the critical section. Each philosopher must successfully enter two nested `with` blocks before transitioning to the EATING state.

2.2 Hardware Interfacing:

Using the `BaseOverlay` class allowed me to control the physical feedback of the system without GPIO mapping:

- **LED:** Used for **Philosophers # 0-3**. Accessed via `base.leds` and controlled via `.on()` and `.off()` methods.

- **RGB:** Used for **Philosopher # 4**. Controlled via `base.rgbleds[4]` to display Green. I used `.write(2)` to display Green, maintaining visual consistency with the other philosophers.

2.3 State Representation:

For different states, making discrete patterns for each one was key to representing functionality

- **EATING:** A high-frequency blink (0.1s interval).
- **NAPPING:** A low-frequency blink (0.5s interval).
- **STARVING:** LED remains off.

3. Results and Analysis

3.1 Timing and Deadlock Prevention

- **Problem:** If every philosopher picks up their left fork simultaneously, the system enters a deadlock where no one can move.
- **Solution:** I implemented an asymmetric strategy where Philosopher 4 picks up the **right** fork first, while Philosophers 0-3 pick up the **left** fork first. This prevents the circular wait, which is necessary for a deadlock to occur; this mathematically guarantees no deadlock in an infinite run of the simulation.

3.2 Randomization:

I utilized `random.randint(3, 6)` for EATING and `random.randint(1, 2)` for NAPPING. While randomized timing helps desynchronize threads and reduces the likelihood of resource contention, it is not a formal guarantee against deadlock.

In a symmetric system, the threads will eventually align in a way that induces a deadlock. My implementation provides a *deterministic guarantee* through asymmetry; by breaking the resource hierarchy for Philosopher 4, a deadlock becomes mathematically impossible regardless of the timing values used. Also, ensuring a mandatory nap, even one shorter than the eating duration, creates the necessary context switch window for the OS to redistribute forks to starving neighbors.

4. Troubleshooting & Difficulties

- **Initialization Errors:**
 - Initially, I attempted to instantiate LEDs using raw integer IDs, which caused a TypeError. I resolved this by accessing the pre-instantiated LED and RGBLED objects directly from the base overlay, ensuring the proper hardware methods were available.

- **Zombie Threads:**
 - To prevent threads from hanging after the cell is stopped, I used a global `keep_running` flag. This allows threads to finish their current blink cycle and exit cleanly.
 - **RGB Mapping:**
 - Initially, the RGB LED required a slightly different control logic than the standard LEDs. I solved this by nesting a `set_led(state)` helper function within the task to abstract the setup more.
 - **Hardware Interrupt:**
 - Utilizing `btns[0].read()` provided a reliable way to terminate the infinite loops. By calling `.join()` on all thread objects after the button press, I ensured all resources were released before the script ended.
-

5. Video Demonstration Link

[Link](#)

The video demonstrates: BTN0, LED Patterns, and Fairness implementing the Assymetry Strategy,

5. Appendix: Jupyter Notebook

```
In [18]: import threading
import time
import random
from pynq.lib import LED, RGBLED, Button
from pynq.overlays.base import BaseOverlay

from datetime import datetime
base = BaseOverlay("base.bit")
btms = base.btm_gpio

# global control to kill loops
keep_running = True

def philosopher_task(id, left_fork, right_fork, led):
    global keep_running

    #functions to handle the difference between LEDs and RGB
    def set_led(state):
        if id == 4: # RGB LED
            led.write(2 if state else 0)
        else: # Green LEDs
            led.on() if state else led.off()

    while keep_running:
        ### STATE: STARVING ###
        set_led(False)

        # DEADLOCK PREVENTION:
        # if everyone grabs their left fork at once, lock up.
        # philosopher 4 grabs Right then Left to break the cycle.
        first, second = (left_fork, right_fork) if id < 4 else (right_fork,
                                                               left_fork)

        with first:
            with second:
                # ## STATE: EATING ##
                # higher rate blink (0.1s)
                eat_duration = random.randint(3, 6)
                start_time = time.time()
                while time.time() - start_time < eat_duration and keep_running:
                    set_led(True); time.sleep(0.1)
                    set_led(False); time.sleep(0.1)

                ### STATE: NAPPING ###
                # lower rate blink (0.5s)
                # RULE Nap < eat to avoid constant starvation
                nap_duration = random.randint(1, 2)
                start_time = time.time()
                while time.time() - start_time < nap_duration and keep_running:
                    set_led(True); time.sleep(0.5)
                    set_led(False); time.sleep(0.5)
```

```
set_led(False) # ensure LED is off when thread closes
```

```
In [20]: from pyng import GPIO

# init 4 Green LEDs
philosopher_leds = [base.leds[i] for i in range(4)] + [base.rgbleds[4]]

# init buttons (button 0 will be the stop button)
btms = base.buttons

# init 5 forks (locks)
forks = [threading.Lock() for _ in range(5)]

print("Hardware and Locks init.")
```

Hardware and Locks initialized.

```
In [24]: keep_running = True
threads = []

# launch the 5 threads
for i in range(5):
    # FORK LOGIC: Phil 0 uses Fork 0 and 1; Phil 4 uses Fork 4 and 0
    t = threading.Thread(target=philosopher_task,
                         args=(i, forks[i], forks[(i+1)%5], philosopher_leds))
    threads.append(t)
    t.start()

print("Running. Press Button 0 to STOP.\n")

try:
    # main loop monitors the button
    while keep_running:
        if btms[0].read() == 1:
            print("\nButton 0 pressed. Shutting down...")
            keep_running = False
        time.sleep(0.1)
except KeyboardInterrupt:
    keep_running = False

# wait for all threads to finish their current loop and exit
for t in threads:
    t.join()

print("Threads stopped. Resources Released.\n")
```

Running. Press Button 0 to STOP.

Button 0 pressed. Shutting down...
Threads stopped. Resources Released.

In []:

In []: