Assignment 2 (Report)

Name: Gabriel Martinez

Course: WES 237A

GitHub: https://github.com/Math140Instructor/wes237a/Assignment2

Video: https://drive.google.com/file/d/14PA2w2cQb2Nz_hgjWunpU0YCFmucd--d/view?usp=sharing

1. Objective

The goal of this assignment is to become familiar with Python's threading library by launching and managing multiple threads and coordinating shared resources using locks to solve the five dining philosophers problem by implementing LED blinking behavior to visualize concurrent thread execution and current state of each theard. Additionally, button interrupts are used to safely terminate running threads, reinforcing proper thread control and synchronization techniques.

2. Design Methodology

A top down design methodology was used to structure the implementation. The overall problem was decomposed into smaller, testable components:

1. I started off by controlling the four onboard leds and solo tri color led
2. Then I utilzed the previous assignment to create a blinking function using the pulse width modulation to control the frequency and duty cycle of the led.
3. Then I used one thread to call this function.
4. Then I used the threading.Events obbject to control the thread so I can signal a way to stop the threads while loop.
5. I then created methods to start, pause, unpause, and stop an led for any thread.
6. Then tested that I had full control of an led on a thread.
7. Created a dictionary of threads for easy access to a thread.
8. I created a button listener that runs on its own thread listening for two things:
   - The 4th button to start the assignment
   - Any other button to stop the threads as requested
9. I had the data structures and control of the threads at this point so I created a generic wait function for an led to blink.
10. From that generic wait function i created a nap,eat, and starve function with predefined wait values and blink rate per assignment request.

11. The initial functions all instantiated each thread to run these nap,eat, and starve functions all with the same values on each thread.
12. That was fine for the first part of the assignment but I had to modify my thread function to allow random wait values for each action for each thread per assignment part 2.

TL;DR. Each component was verified independently and manually tested before integrating into the final workflow.

3. Workflow and Implementation

The strategy for this assignment was to create a single worker function that could be easily instantiated and run on its own thread. The worker function, called `blinkLedWithLockControl`, operates by attempting to access an array of shared locks. In this design, each thread represents a philosopher, and the five locks represent the philosophers' left forks.

Part A2.1

The workflow begins by checking whether the current thread can acquire its own (left) fork. If successful, it then attempts to acquire the right fork using a non blocking approach.

Starve

If either fork is unavailable, any acquired lock is released and a `starve` method is called, which turns the LED off to visually indicate the starving state with a predefined `duration`. This process repeats until the thread is able to acquire both forks.

Eat

Once both forks are acquired, the `eat` method is executed. This method turns the LED on for a predefined `duration` to visually represent the eating state.

Nap

After eating, the locks are released and a `nap` method is called, which blinks the LED at a slower rate to indicate the resting state with a predefined `duration`.

This cycle continues until a button interrupt is triggered, which safely terminates the application.

Part A2.2

The same workflow was used however I needed to modify the predefinded `duration` values for the starve, eat, and nap methods so that each thread uses a random integer value per assignment ask.

4. Difficulties and Troubleshooting

Several challenges were encountered during development. First challenges was determining how to properly control and terminate a thread. After reviewing the Python threading API, I used the Event object to manage the thread's lifecycle. During early testing, many threads were spawned and continued running in the background, consuming system resources and requiring frequent kernel restarts. Another major challenge involved tuning the timing of the non blocking thread behavior, which initially led to excessive starvation. Adjusting the duration of each philosopher's actions required many iterations to achieve stable and balanced execution. Despite these efforts, no systematic or analytical approach was identified that could guarantee the absence of deadlocks.

5. Results and Analysis

The objectives of this assignment were successfully met. Individual LEDs were controlled by separate threads, demonstrating proper thread creation and independent execution. Non blocking lock based synchronization was used to safely access shared resources and produce the correct LED behavior, providing clear visual feedback of each thread's state. Button interrupts were also implemented to terminate threads as intended, confirming correct thread control and coordination.

Assignment 2 (Code)

The following section presents the code results for the assignment, followed by the report at the end.

Name: Gabriel Martinez

Course: WES 237A

GitHub: https://github.com/Math140Instructor/wes237a/Assignment2

In [1]:
```
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
```

In [2]:
```
import pynq.lib.rgbled as rgbled
sololed = rgbled.RGBLED(0) # seperate Tricolor LED
```

In [3]:
```
# Green
sololed.on(rgbled.RGB_GREEN)
```

In [4]:
```
# test off
sololed.off()
```

In [5]:
```
# Onboard LEDs - 0,1,2,3
leds = base.leds
#leds.write(0b1101,0b1111) #. works but easier to access is by index
```

In [6]:
```
import time
# TEST: toggle the 4 leds by index with a delay of .5secs on one thread
# goal is to programmatically control the onboard leds
i=0
numTestIter=4
print("testing toggle the 4 leds by index with a delay of .5s for %.2fs..."%
while i<numTestIter:
    leds[i%4].on()
    time.sleep(.5)
    leds[i%4].off()
    i=i+1
print("test done.")
```

```
testing toggle the 4 leds by index with a delay of .5s for 2.00s...
test done.
```

In [7]:
```
# setup thread events for LED control on a single thread: kill,pause
import threading
numPhilosophers=5
ledKillBlinkEvent = [threading.Event() for i in range(numPhilosophers)] # co
ledPauseBlinkEvent = [threading.Event() for i in range(numPhilosophers)] # c
```

In [8]:
```
# needs to run on its own thread to register the thread event
# modularized the action to blink for better control of an led.
def pwm(i,freq=1,duty=50):
    '''
    Dependencies are the global leds and solo led variables.
    The index i going from 0 to 3 controls the onboard leds anything beyond
    '''
    global leds, sololed
    T = 1./freq if freq>0 else 1
    on = T * (duty/100.)
    off = T - on
    if i<4:
        leds[i].on()
        time.sleep(on)
        leds[i].off()
        time.sleep(off)
    elif i>=4:
        sololed.on(rgbled.RGB_GREEN)
        time.sleep(on)
        sololed.off()
        time.sleep(off)
def testPerfomance(i,freq=1,duty=50): # used to debug the pwm function
    i=0
```

```
    while i<10:
        start=time.perf_counter()
        pwm(i,freq,duty)
        end = time.perf_counter() - start
        i+=1
        print("%.6fs %d%%"%(end,duty))
def blinkLed(i,freq=1, duty=50):
    ''' Description: main api to expose the pwm function
    '''
    global ledKillBlinkEvent,ledPauseBlinkEvent
    print("blink %d led, %dHz, %d%% duty"%(i,freq,duty))
    while not ledKillBlinkEvent[i].is_set():
        if not ledPauseBlinkEvent[i].is_set():
            pwm(i,freq,duty)
        else:
            time.sleep(.01) # dont starve the thread when not blinking
    print("blink %d done."%i)
```

In [9]:

```
# Test the blinkLed function to verify thread control of an led
# blink on led index 3 at 2Hz on a thread
indxLed=3
blinkHz=2
duty=1
t1 = threading.Thread(target=blinkLed, args=(indxLed,blinkHz,duty))
t1.start()
```

```
blink 3 led, 2Hz, 1% duty
```

In [10]:

```
# Enables the event and stops the blink for the specified led
ledPauseBlinkEvent[indxLed].set() # broadcast a thread event for the 3rd inc
```

In [11]:

```
# Reenables the event and continues the blink for the specified led
ledPauseBlinkEvent[indxLed].clear() # reset the thread event
```

In [12]:

```
# Kills the 3rd thread
ledKillBlinkEvent[indxLed].set() # broadcast an event to kill the thread.
```

In [13]:

```
# remember all threads created for the dining philosophers
threads={}
```

In [14]:

```
# modularize the thread controls for any led: start, pause, unpause, and sto
# stores the threads in a dictionary with the index as their key for quick t
global threads, ledKillBlinkEvent, ledPauseBlinkEvent
def unpauseLED(i):
    ledPauseBlinkEvent[i].clear()
    print("led %d unpaused event."%i)
def pauseLED(i):
    ledPauseBlinkEvent[i].set()
    print("led %d paused event."%i)
def stopLED(i):
    ledKillBlinkEvent[i].set();
    if i in threads:
        del threads[i]
    print("led %d stopped event."%i)
def startLED(i,freq=1, duty=50, _blink=None, *_blinkArgs):
```

```
    if i in threads:
        print("led is already active.")
        return threads[i]
    print("led %d start event."%i)
    ledKillBlinkEvent[i].clear()
    ledPauseBlinkEvent[i].clear()
    if _blink == None:
        _blink = blinkLed

    if _blinkArgs:
        t = threading.Thread(target=_blink, args=(i,freq,duty,*_blinkArgs),
    else:
        t = threading.Thread(target=_blink, args=(i,freq,duty), name="blinkl
    threads[i]=t
    t.start()
    return t
def resetEvents():
    for i in range(numPhilosophers):
        ledPauseBlinkEvent[i].clear()
        ledKillBlinkEvent[i].clear()
    print("events reset.")
def stopAll():
    for i in range(numPhilosophers):
        stopLED(i)
def startAll(freq=1,duty=50, _blink=None, _blinkArgs=None):
    for i in range(numPhilosophers):
        if _blinkArgs:
            startLED(i,freq,duty,_blink,*_blinkArgs)
        else:
            startLED(i,freq,duty,_blink)
def viewThreads():
    print("["+", ".join(str(t) for t in threads.keys())+"]")
    print("active threads: %d"%(threading.active_count()))
```

In [15]:

```
resetEvents() # start from scratch clear all thread events
stopAll() # force kill any previous threads in the threads dictionary
viewThreads() # verifies dictionary is empty
print(threading.active_count()) # used to verify no additional threads are l
```

```
events reset.
led 0 stopped event.
led 1 stopped event.
led 2 stopped event.
led 3 stopped event.
led 4 stopped event.
[]
active threads: 9
9
```

In [16]:

```
# test all 5 leds blink at different rates on their own seperate thread to v
for i in range(numPhilosophers):
    startLED(i,i+1)
print(threading.active_count()) # used to show that 5 additional threads are
```

```
led 0 start event.
blink 0 led, 1Hz, 50% duty
led 1 start event.
blink 1 led, 2Hz, 50% duty
led 2 start event.
blink 2 led, 3Hz, 50% duty
led 3 start event.
blink 3 led, 4Hz, 50% duty
led 4 start event.
blink 4 led, 5Hz, 50% duty
14
```

In [17]:
```
# test that it shouldn't create any threads because they already exist
for i in range(numPhilosophers):
    startLED(i,i+1)
print(threading.active_count()) # used to show that no additional threads we
```

```
led is already active.
led is already active.
led is already active.
led is already active.
led is already active.
14
```

In [18]:
```
viewThreads() # used to debug the threads in the stored dictionary
```

```
[0, 1, 2, 3, 4]
active threads: 14
```

In [19]:
```
# test pause event for each thread
for i in range(numPhilosophers):
    pauseLED(i)
```

```
led 0 paused event.
led 1 paused event.
led 2 paused event.
led 3 paused event.
led 4 paused event.
```

In [20]:
```
# test unpause event for each thread
for i in range(numPhilosophers):
    unpauseLED(i)
```

```
led 0 unpaused event.
led 1 unpaused event.
led 2 unpaused event.
led 3 unpaused event.
led 4 unpaused event.
```

In [21]:
```
#test kill threads for each thread
for i in range(numPhilosophers):
    stopLED(i)
```

```
led 0 stopped event.
led 1 stopped event.
led 2 stopped event.
led 3 stopped event.
led 4 stopped event.
```

In [22]:
```
# test threading.Lock() behavior to make sure I understand the behavior of a
lf = threading.Lock()
rf = threading.Lock()
cf = threading.Lock()
print(lf.locked(),cf.locked(),rf.locked())
print(lf.acquire(blocking=False),rf.acquire(blocking=False))
print(lf.locked(),rf.locked())
time.sleep(1)
lf.release()
print(lf.acquire(blocking=False),rf.acquire(blocking=False))
time.sleep(1)
rf.release()
lf.release()
time.sleep(1)
print(lf.locked(),rf.locked())
```

```
blink 2 done.
False False False
True True
True True
blink 4 done.
blink 1 done.
blink 3 done.
blink 3 done.
blink 0 done.
True False
False False
```

Part A2.1:

- Write code for the dining philosophers' problem. Use five LEDs, one for each philosopher, and five locks for forks. The five LEDs will be the four on-board green LEDs above the buttons and one of the on-board RGB LEDs that we saw in Lab1 (make it green to match the other LEDs).
- Find appropriate durations for the philosophers to be eating and napping. Consider choices such that your threads do not go into a constant starvation. (i.e., should napping time be greater than or less than eating time?)
- When one of the philosophers is eating, both forks are used by that philosopher, and the LED should blink at a higher rate to indicate "eating".
- When a philosopher is napping, the LED should blink at a lower rate to indicate "napping".
- When a philosopher is waiting for forks, its LED should be off to indicate "starving".
- The code must run forever. To terminate the program, you have to use push buttons.

In [23]:
```
# Created a button listener that will be attached on its own thread to start
# assignment
import threading
import time

def buttonListener(_stopAll=None, _philosophizeEvent=None, _btnListenerEvent
```

```
    global base
    btns = base.btns_gpio
    btnListenerKillEvent = threading.Event() if not _btnListenerEvent else _

    print("button listener listening...")
    while not btnListenerKillEvent.is_set():
        time.sleep(.005)
        if btns.read() & 0b1000:
            print("start philosophizing event.")
            if _philosophizeEvent and not _philosophizeEvent.is_set():
                _philosophizeEvent.set()
            time.sleep(.3)
        elif btns.read():
            btnListenerKillEvent.set()
            if _stopAll:
                _stopAll()
            if _philosophizeEvent:
                _philosophizeEvent.clear()
                print("philosophizing cleared.")
            print(threading.active_count())
    print("button listener done.")
```

In [24]:

```
'''
    Description: modularized wait function for the pwm methods. This method
    given by the frequency multiplied by the number of seconds to wait.
    Ex, Freq=60 and we want to wait 5 seconds then 60*5 = 300 iterations is
'''
def wait(i,freq,duty,waitDur,_pwm=None):
    '''generic wait method that will blink led at position i with the provic
    global ledKillBlinkEvent
    numItr=0
    stopItr=freq*waitDur
    while(not ledKillBlinkEvent[i].is_set() and numItr<stopItr): # blink as
        if _pwm:
            _pwm(i,freq,duty)
        else:
            print("iter: %d"%numItr) # only used for debugging
        numItr += 1
    return
def eat(i, freq=60, duty=100, waitDur=5, _pwm=pwm): # predefine the eating l
    wait(i,freq,duty,waitDur,_pwm)
def nap(i, freq=5, duty=50, waitDur=5, _pwm=pwm): # predefine the napping le
    wait(i,freq,duty,waitDur,_pwm)
def starve(i, freq=1, duty=0, waitDur=5, _pwm=pwm): # predefine the starving
    wait(i,freq,duty,waitDur,_pwm)
```

In [25]:

```
# Create the philosophers forks. Assume each philospher at index i has a lef
forks = [threading.Lock() for _ in range(numPhilosophers)] # represents each
startPhilosophizingEvent = threading.Event() # signal to start the assignmer

# TODO: make thread see its neighbors: [0 1 2 3 4]
# edge cases:
# i=4 right fork is index 0's fork
# global numPhilosophers, ledKillBlinkEvent, ledPauseBlinkEvent
```

```
def blinkLedWithLockControl(i,freq=1, duty=50, starveDur=None,eatDur=None,na
    if starveDur and eatDur and napDur:
        print("starveDur=%ds, eatDur=%ds, napDur=%ds"%(starveDur,eatDur,napD

    print("philosphers waiting for event to start...")
    global startPhilosophizingEvent

    while not ledKillBlinkEvent[i].is_set() and not startPhilosophizingEvent
        pwm(i,freq,duty)

    print("philosophizing event started %d."%i)
    while not ledKillBlinkEvent[i].is_set():

        leftFork = forks[i%numPhilosophers] # his fork (current) is the left
        rightFork = forks[(i+1)%numPhilosophers] # neigbors fork

        if leftFork.locked(): #if current fork is being used then starve for
            starve(i,freq=1,duty=0,waitDur=starveDur)
            continue

        leftFork.acquire(blocking=False) #tell everyone im trying to eat, lo

        if rightFork.locked():
            leftFork.release()
            continue

        rightFork.acquire(blocking=False) # grab right fork. Ready to eat

        if (not ledKillBlinkEvent[i].is_set()
        and leftFork.locked()
        and rightFork.locked()):
            # i am eating
            # eat for 5sec, visually stay on to tell everyone im eating
            eat(i,freq=60,duty=100,waitDur=eatDur)
            leftFork.release()
            rightFork.release()
        else: # shouldnt reach here but just in case
            print("ERROR")
            stopAll()

        #take NAP
        nap(i,freq=10,duty=25,waitDur=napDur)

    print("blink %d done."%i)
```

In [26]:

```
btnListenerThread = threading.Thread(target=buttonListener, args=(stopAll,st
btnListenerThread.start()
starveDur,eatDur,napDur = 1,1,1
startAll(freq=1,_blink=blinkLedWithLockControl, _blinkArgs=(starveDur,eatDur
```

```
button listener listening...
led 0 start event.
starveDur=1s, eatDur=1s, napDur=1s
philosphers waiting for event to start...
led 1 start event.
starveDur=1s, eatDur=1s, napDur=1s
philosphers waiting for event to start...
led 2 start event.
starveDur=1s, eatDur=1s, napDur=1s
philosphers waiting for event to start...
led 3 start event.
starveDur=1s, eatDur=1s, napDur=1s
philosphers waiting for event to start...
led 4 start event.
starveDur=1s, eatDur=1s, napDur=1s
philosphers waiting for event to start...
start philosophizing event.
philosophizing event started 0.
philosophizing event started 1.
philosophizing event started 2.
philosophizing event started 3.
philosophizing event started 4.
led 0 stopped event.
led 1 stopped event.
led 2 stopped event.
led 3 stopped event.
led 4 stopped event.
philosophizing cleared.
14
button listener done.
blink 0 done.
blink 2 done.
blink 3 done.
blink 1 done.
blink 4 done.
```

Part A2.2:

- In this part, you use the random library to generate random numbers for the eating and napping states. By using random.randint(a, b) you can get a random number between a and b.
- You have to set the boundaries for your random number (a, b) such that napping is not longer than eating, and therefore your threads do not go to a constant starvation.

In [27]:
```
import random
def blinkLedWithLockControl(i,freq=1, duty=50, starveDur=None,eatDur=None,na
    if starveDur and eatDur and napDur:
        print("starveDur=%ds, eatDur=%ds, napDur=%ds"%(starveDur,eatDur,napD
    else:
        print("Will use random wait times for each action.")

    print("philosphers waiting for event to start...")
```

```
    global startPhilosophizingEvent

    while not ledKillBlinkEvent[i].is_set() and not startPhilosophizingEvent
        pwm(i,freq,duty)

    print("philosophizing event started %d."%i)
    while not ledKillBlinkEvent[i].is_set():

        starveDur = random.randint(1,10)
        eatDur = random.randint(1,10)
        napDur = random.randint(0,eatDur-1)

        leftFork = forks[i%numPhilosophers] # his fork (current) is the left
        rightFork = forks[(i+1)%numPhilosophers] # neigbors fork

        if leftFork.locked(): #if current fork is being used then starve for
            starve(i,freq=1,duty=0,waitDur=starveDur)
            continue

        hasLeft = leftFork.acquire(blocking=False) #tell everyone im trying

        if rightFork.locked():
            leftFork.release()
            continue # go back and starve

        hasRight = rightFork.acquire(blocking=False) # grab right fork. Reac

        if (not ledKillBlinkEvent[i].is_set()
        and hasLeft
        and hasRight):
            # i am eating
            # eat for 5sec, visually stay on to tell everyone im eating
            eat(i,freq=60,duty=100,waitDur=eatDur)
            if hasLeft:
                leftFork.release()
            if hasRight:
                rightFork.release()
        else: # shouldnt reach here but just in case
            print("ERROR")
            stopAll()

        #take NAP
        nap(i,freq=10,duty=25,waitDur=napDur)

    print("blink %d done."%i)
```

In [28]:

```
stopAll()
resetEvents()
btnListenerThread = threading.Thread(target=buttonListener, args=(stopAll,st
btnListenerThread.start()

startAll(freq=1,_blink=blinkLedWithLockControl, _blinkArgs=None)
```

```
led 0 stopped event.
led 1 stopped event.
led 2 stopped event.
led 3 stopped event.
led 4 stopped event.
events reset.
button listener listening...
led 0 start event.
Will use random wait times for each action.
philosphers waiting for event to start...
led 1 start event.
Will use random wait times for each action.
philosphers waiting for event to start...
led 2 start event.
Will use random wait times for each action.
philosphers waiting for event to start...
led 3 start event.
Will use random wait times for each action.
philosphers waiting for event to start...
led 4 start event.
Will use random wait times for each action.
philosphers waiting for event to start...
led 0 stopped event.
led 1 stopped event.
led 2 stopped event.
led 3 stopped event.
led 4 stopped event.
philosophizing cleared.
14
button listener done.
philosophizing event started 0.
blink 0 done.
philosophizing event started 1.
blink 1 done.
philosophizing event started 2.
blink 2 done.
philosophizing event started 3.
blink 3 done.
philosophizing event started 4.
blink 4 done.
```

In [29]:

```
from IPython.display import Video, display
display(Video("Fixed.mp4", embed=True))
```

In [ ]:

```
from IPython.display import Video, display
display(Video("Random.mp4", embed=True))
```

In [ ]: