WES 237A – Assignment 2, Dining Philosophers

Ricardo Lizarraga, rlizarraga0@gmail.com , *619.252.4157* | **PID**: **A69028483**
https://github.com/RiLizarraga/WES237A_Assign2

Demo videos files at: https://drive.google.com/drive/folders/1-4u9nCfFqKFPoN8tsQZnBI7ovx16lEZg?usp=sharing
"Assigm2_SolutionFor_A2_1.mp4",
"Assigm2_SolutionFor_A2_2.mp4",

The goals for this assignment are as follows

1.< > Familiarlize yourself with the python threading library.
- •< > Launching multiple threads
- • < > Sharing locks between threads

2. < > Implement LED blinking capabilities
3. < > Use button interrupts for killing threads

Problem

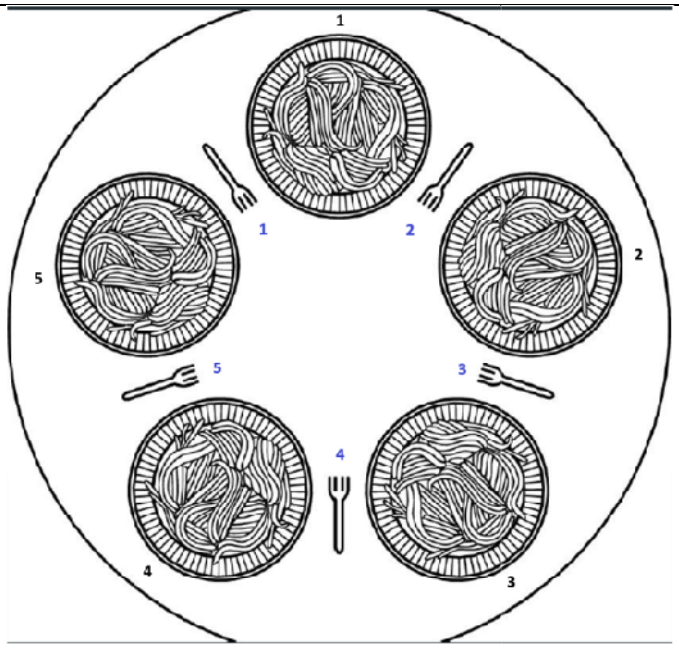
There are five philosophers dining together at table with five forks. Each philosopher shares their forks with neighboring philosophers and needs both forks (left and right) to eat. https://www.youtube.com/watch?v=NbwbQQB7xNQ
When a philosopher is done eating, it relinquishes the forks and takes a nap. Finally, when the philosopher is finished with the nap, it will wait, starving, for the two pairs of forks (left and right) to be freed in order to eat. Thus, there are 3 possible states for each philosopher

1. EATING: the philosopher has both forks (left and right)
2. NAPPING: the philosopher is finished eating
3. STARVING: the philosopher is waiting to have both forks (left and right)

Part A2.1:

Implementation/ design:

All input data is scalable, we just have to change the data arrays for a different size & names of filosophers, including Dinning States, which is very important to keep the right algorithm running with the correct state machine handling
The progress flow of the algorithm is driven by "Finite-State Machine"



Philosopher seating positions:
1.- Descartes
2.- Aristotle
3.- Socrates
4.- Thoreau
5.- Th. Paine

Optimized Starting sequence:
1.- Descartes
3.- Socrates
5.- Th. Paine
2.- Aristotle
4.- Thoreau

```
enum_dining_sequence = enumerate(dining_sequence)
for idx, e in enum_dining_sequence:
    threads[e].start()
```

Two Critical section were implemented (Locking resources or shared resources):

1. Locking for trying to lock both forks at the same time (, it requires a locking to be granted, else will have to try later on
2. After a Philosopher is allowed "to Query" for a "Request of both forks **OR** none", if both forks are available, then both forks will be locked by that particular Philosopher **ATOMIZATION** of Left & Right forks), **else** will have to try later on. Locked forks are release after Philosopher has finished Eating. lock Release will be executed by the user-thread, but locking is exclusively done just by the atomic function

Finite-State machine (FSM)

A **state machine** is a model of a system with discrete dynamics that at each reaction maps valuations of the inputs to valuations of the outputs, where the map may depend on its current state. A **finite-state machine** (**FSM**) is a state machine where the set *States* of possible states is finite.

If the number of states is reasonably small, then FSMs can be conveniently drawn using a graphical notation like that in Figure 3.3. Here, each state is represented by a bubble, so for this diagram, the set of states is given by

$$States = \{State1, State2, State3\}.$$

At the beginning of each sequence of reactions, there is an **initial state**, State1, indicated in the diagram by a dangling arrow into it.

State = ['STARVING','EATING','NAPPING','DONE']

3.3.1 Transitions

Transitions between states govern the discrete dynamics of the state machine and the mapping of input valuations to output valuations. A transition is represented as a curved arrow, as shown in Figure 3.3, going from one state to another. A transition may also start and end at the same state, as illustrated with State3 in the figure. In this case, the transition is called a **self transition**.

In Figure 3.3, the transition from State1 to State2 is labeled with "guard / action." The **guard** determines whether the transition may be taken on a reaction. The **action** specifies what outputs are produced on each reaction.

A guard is a **predicate** (a boolean-valued expression) that evaluates to *true* when the transition should be taken, changing the state from that at the beginning of the transition to that at the end. When a guard evaluates to *true* we say that the transition is **enabled**. An action is an assignment of values (or *absent*) to the output ports. Any output port not

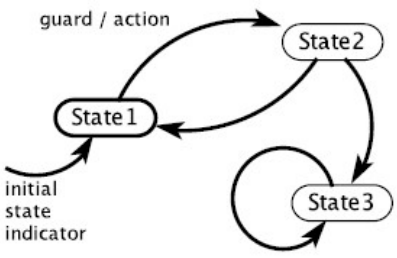


Figure 3.3: Visual notation for a finite state machine.

'STARVING' -> 'EATING' -> 'NAPPING' -> 'DONE'

Global variables and scalable input data

```
Names, dining_sequence = ['Descartes','Aristotle','Socrates ','Thoreau  ','Th. Paine'], [0, 2, 4, 1, 3]
State, ForkNumbers = ['STARVING','EATING','NAPPING','DONE'], [1, 2, 2, 3, 3, 4, 4, 5, 5, 1]
Philosophers, threads, forklocks = [], [], []
led4, led5 = rgbled.RGBLED(4), rgbled.RGBLED(5)
eating_time, napping_time, askagain_time = 10, 20, 0.3
eating_blinkingRate, napping_blinkingRate, askagain_blinkingRate = 0.03, 1, 0.03
```

Philosopher names, States, Forks available/ placing time, eating time, napping & askagain time can be changed on the header of the script for a custom simulation

In order to better architecture design skeleton, execution, I had to design 2 classes:

```
class forks:
    def __init__(self, fork1, fork2):
        self.fork1 = fork1
        self.fork2 = fork2
class Philosopher:
    def __init__(self, name, state, forks):
        self.name = name
        self.state = state
        self.forks = forks
```

There're a total of 6 threads:
One per each Philosopher actions and another one to take user input of the push buttons.

```
Press buttons 0-3
   0 - Restart simulation
   1 - No use
   2 - No use
   3 - Exit program
```

At the beginning of the execution, Jupyter Notebook prints the state of each Philosopher:

```
Descartes    has  Forks  1  and  2   State:  STARVING
Aristotle    has  Forks  2  and  3   State:  STARVING
Socrates     has  Forks  3  and  4   State:  STARVING
Thoreau      has  Forks  4  and  5   State:  STARVING
Th. Paine    has  Forks  5  and  1   State:  STARVING
```

When the program has exit, Jupyter Notebook also print the state of each Philosopher:

```
Descartes    has  Forks  1  and  2   State:  DONE
Aristotle    has  Forks  2  and  3   State:  DONE
Socrates     has  Forks  3  and  4   State:  DONE
Thoreau      has  Forks  4  and  5   State:  DONE
Th. Paine    has  Forks  5  and  1   State:  DONE
```

In addition to that, will also print the state for each of the Fork locks (this to make sure all Forks are release and available while all Philosophers are satisfied eaten.

```
Forks lock status: False False False False False
```

<X > Write code for dining philosophers problem. Use **five LEDs**, one for each philosopher and
<X > **five locks for forks**.
<X> **The five LEDs** will the 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** other **other LEDs**).
•<x> Find appropriate durations for the philosophers to be eating and napping. Consider choices such that your threads do not go to a constant starvation. (i.e. should napping time be greater than or less than eating time?)
<x> • When one of the philosophers is eating, both forks is used by that philosopher and its LED should blink with a higher rate to indicate "eating".
• <x> When a philosopher is napping, its LED should blink with a lower rate to indicate "napping".
• <x> When a philosopher is waiting for forks, its LED should be off to indicate "starving".
• <x> The code must run for ever. To terminate the program, you have to use push buttons.

Part A2.2:

• <X> In this part, you use 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.
• <X> 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.

```
EATING_MIN, EATING_MAX, DINING_MIN = 4, 8, 2
```

```
eating_time = random.randint(EATING_MIN, EATING_MAX)
print("Random eating_time: {}".format(eating_time))
napping_time = random.randint(DINING_MIN, eating_time)#napping is not longer than eating
print("Random napping_time: {}".format(napping_time))
```

Deliverables

Each student must submit the following **individually**
• A PDF report detailing your work flow and relative Jupyter notebook cells relating to your progress.
• Your report should detail your work flow throughout the assignment. Be sure to discuss **any** difficulties or troubles you encountered and your troubleshooting procedure. Also, detail your thought process which led to the design and implementation of the code. For 1 example, describe your top-down design methodology (i.e. how did you split the large task into smaller, more incremental jobs? How were you able to test each of these smaller parts?)
• Please also include a brief discussion on the timings for EATING, NAPPING, and STARVING you chose in order to avoid a deadlock.
2. Your complete Jupyter notebook, downloaded as a PDF **attached at the very end of your report**
• You can do this by selecting 'File -> Print Preview' then printing to PDF from the browser.
• Use a PDF stitching tool like pdfjoiner to join your Report and Jupyter Notebook into a single PDF file
3. Video demonstration of your code working on the PYNQ board. Please limit videos to 60 seconds and upload them to a video sharing site and include the link on your PDF report. Each team should submit the following **one per team**
1. All relevant code (.ipynb, .py, .cpp, .c, etc files) pushed to your team's git repo.

Difficulties:

Avoid dead locks

Testing tactic:

Testing takes time, so initially I used very very short time to quickly evaluate behavior

Comments/ Observations

The key part for a good design of the algorithm is to view the overall architecture, create makes sense structured objects. Like a data class for the forks needed then to be re-used in the Philosopher's class, then later create an array of instanced philosophers objects. The right locking design will warranty no dead locks

In [ ]:
```
from IPython.display import display, HTML; display(HTML("<style>.container { width:100
import threading;import time;import pynq.lib.rgbled as rgbled;from pynq.overlays.base
base = BaseOverlay("base.bit")

Names, dining_sequence = ['Descartes','Aristotle','Socrates ','Thoreau  ','Th. Paine']
State, ForkNumbers = ['STARVING','EATING','NAPPING','DONE'], [1, 2, 2, 3, 3, 4, 4, 5,
Philosophers, threads, forklocks = [], [], []
led4, led5 = rgbled.RGBLED(4), rgbled.RGBLED(5)
eating_time, napping_time, askagain_time = 10, 20, 0.3
eating_blinkingRate, napping_blinkingRate, askagain_blinkingRate = 0.03, 1, 0.03
sz = len(Names)
threads_flag = True    # "cond" will keep threads live or kill them all
class forks:
    def __init__(self, fork1, fork2):
        self.fork1 = fork1
        self.fork2 = fork2
class Philosopher:
    def __init__(self, name, state, forks):
        self.name = name
        self.state = state
        self.forks = forks
def leds(n, onoff):
    if n < 4:    base.leds[n].write(onoff)
    elif n == 4: led4.write( 0x2 * onoff ) # toggle and keep it green pls
    else:        led5.write( 0x2 * onoff )
def blink(t, d, n):
    for i in range(t):       #[t]: times to blink the LED
        if n < 4:    base.leds[n].toggle()
        elif n == 4: led4.write( 0x2 * (not led4.read())) # toggle and keep it green p
        else:        led5.write( 0x2 * (not led5.read()))

        time.sleep(d)        #[d]: duration (in seconds) for the LED to be on/off
    if n < 4:    base.leds[n].off()
    elif n == 4: led4.write(0x0)
    else:        led5.write(0x0)
def blinkbydelay(totalDelay, blinkingRate, n):
    times2blink = int(totalDelay/blinkingRate)
    blink(times2blink,blinkingRate,n)
def CanIHaveBothForks(_l, forklocks, num):
    fork1 = Philosophers[num].forks.fork1[1];fork2 = Philosophers[num].forks.fork2[1]
    if forklocks[fork1-1].locked() == False and forklocks[fork2-1].locked() == False:
        forklocks[fork1-1].acquire();forklocks[fork2-1].acquire()
        print("{} fork {} and {} are available, now locked".format(Philosophers[num].r
        return True
    else:#print("{} forks NOT available".format(Philosophers[num].name))
        return False
def worker_t(_l, forklocks, num):# Worker function to try and acquire resource, [_l]:
    while threads_flag:
        if Philosophers[num].state == State[0]:#STARVING
            #print("{} Can I ask? ".format(Philosophers[num].name))
            if _l.locked() == False:#Are you available for asking for both forks?
                _l.acquire()
                res = CanIHaveBothForks(_l, forklocks, num)
                if res == True:# Here we have both forks!
                    leds(num,1)
                    Philosophers[num].state = State[1]#EATING   #move on machine state
                    print("{} changed state to: {}".format(Philosophers[num].name, Sta
                else:
```

```
                    time.sleep(0.5)#was 3
                _l.release()
            else:
                time.sleep(askagain_time)
        elif Philosophers[num].state == State[1]:#EATING
            print("{} Eating".format(Philosophers[num].name))
            blinkbydelay(eating_time, eating_blinkingRate, num)#time.sleep(eating_time
            Philosophers[num].state = State[2]#NAPPING
            print("{} Done Eating, changed state to: {}\n".format(Philosophers[num].na
            # Relinquish forks
            fork1 = Philosophers[num].forks.fork1[1];fork2 = Philosophers[num].forks.f
            forklocks[fork1-1].release();forklocks[fork2-1].release()
            leds(num,0)
        elif Philosophers[num].state == State[2]:#NAPPING
            print("{} Napping".format(Philosophers[num].name))
            blinkbydelay(napping_time, napping_blinkingRate, num)#time.sleep(napping_t
            Philosophers[num].state = State[3]#DONE
            print("\n{} Done Napping, changed state to: {}".format(Philosophers[num].r
            leds(num,1)
        elif Philosophers[num].state == State[3]:#DONE
            time.sleep(1)

        time.sleep(1)
async def get_btns(_loop):
    btns = base.btns_gpio
    global threads_flag, leds, btn0_flag, btn1_flag, btn2_flag, btn3_flag, Philosopher
    while threads_flag:
        await asyncio.sleep(0.1)
        if btns.read() != 0:
            if (btns[0].read()==1):
                await asyncio.sleep(0.3)
                print("button 0 pressed, Restart the Simulation")
                time.sleep(0.1)
                print("Simulation Re-started")
                enum_dining_sequence = enumerate(dining_sequence)# launch philosophers
                for idx, e in enum_dining_sequence:
                    Philosophers[idx].state = State[0]
                    time.sleep(0.2)
                    leds(idx,0)
                btn0_flag = True
            elif (btns[1].read()==1):
                await asyncio.sleep(0.3)
                print("button 1 pressed")
                btn0_flag = True
            elif (btns[2].read()==1):
                await asyncio.sleep(0.3)
                print("button 2 pressed")
                btn0_flag = True
            else:
                await asyncio.sleep(0.3)
                btn3_flag = True
                print("button 3 pressed, Exit program")
                threads_flag = False
                await asyncio.sleep(1)
                Names = ['Descartes','Aristotle','Socrates ','Thoreau  ','Th. Paine']
                for name in enumerate(Names):#Show a Status Summary
                    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[nam
                print("Forks lock status: {} {} {} {} {}".format(forklocks[0].locked()

                _loop.stop()
```

```
print("Press buttons 0-3\n   0 - Restart simulation\n   1 - No use\n   2 - No use\n

enum_ForkNumbers = enumerate(ForkNumbers)
for name in enumerate(Names):
    neededforks = forks(next(enum_ForkNumbers),next(enum_ForkNumbers))
    p = Philosopher(name[1], State[0],neededforks)
    Philosophers.append(p)

for name in enumerate(Names):#Show a Status Summary
    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[name[0]].forks.fork

askingforboth = threading.Lock()#atomic locking/release
for n in range(5):
    leds(n,0)
    singlelock = threading.Lock()
    forklocks.append(singlelock)

for i in range(5):#Philosophers will show up to the table to be seated, every 2 second
    t = threading.Thread(target=worker_t, args=(askingforboth, forklocks, i))
    threads.append(t)

enum_dining_sequence = enumerate(dining_sequence)# launch philosophers' thread in an o
for idx, e in enum_dining_sequence:
    threads[e].start()

loop = asyncio.new_event_loop();  # Instance event_loop object
loop.create_task(get_btns(loop)); # take user input buttons

loop.run_forever()
loop.close()

for t in threads:
    t.join()
    print('{} joined'.format(t.name))

for name in enumerate(Names):#Show a Status Summary
    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[name[0]].forks.fork

print("Forks lock status: {} {} {} {} {}".format(forklocks[0].locked(),forklocks[1].lo

##input("\n\t\t...press any key to exit...")
for n in range(5):
    leds(n,0)
print("Done.")
```

In [ ]:

In [1]:
```
from IPython.display import display, HTML; display(HTML("<style>.container { width:100
import threading;import time;import pynq.lib.rgbled as rgbled;from pynq.overlays.base
import random
base = BaseOverlay("base.bit")

Names, dining_sequence = ['Descartes','Aristotle','Socrates ','Thoreau  ','Th. Paine']
State, ForkNumbers = ['STARVING','EATING','NAPPING','DONE'], [1, 2, 2, 3, 3, 4, 4, 5,
Philosophers, threads, forklocks = [], [], []
led4, led5 = rgbled.RGBLED(4), rgbled.RGBLED(5)
askagain_time = 0.3
EATING_MIN, EATING_MAX, DINING_MIN = 4, 8, 2
eating_blinkingRate, napping_blinkingRate, askagain_blinkingRate = 0.03, 1, 0.03
sz = len(Names)
threads_flag = True    # "cond" will keep threads live or kill them all
class forks:
    def __init__(self, fork1, fork2):
        self.fork1 = fork1
        self.fork2 = fork2
class Philosopher:
    def __init__(self, name, state, forks):
        self.name = name
        self.state = state
        self.forks = forks
def leds(n, onoff):
    if n < 4:    base.leds[n].write(onoff)
    elif n == 4: led4.write( 0x2 * onoff ) # toggle and keep it green pls
    else:        led5.write( 0x2 * onoff )
def blink(t, d, n):
    for i in range(t):       #[t]: times to blink the LED
        if n < 4:    base.leds[n].toggle()
        elif n == 4: led4.write( 0x2 * (not led4.read())) # toggle and keep it green p
        else:        led5.write( 0x2 * (not led5.read()))

        time.sleep(d)        #[d]: duration (in seconds) for the LED to be on/off
    if n < 4:    base.leds[n].off()
    elif n == 4: led4.write(0x0)
    else:        led5.write(0x0)
def blinkbydelay(totalDelay, blinkingRate, n):
    times2blink = int(totalDelay/blinkingRate)
    blink(times2blink,blinkingRate,n)
def CanIHaveBothForks(_l, forklocks, num):
    fork1 = Philosophers[num].forks.fork1[1];fork2 = Philosophers[num].forks.fork2[1]
    if forklocks[fork1-1].locked() == False and forklocks[fork2-1].locked() == False:
        forklocks[fork1-1].acquire();forklocks[fork2-1].acquire()
        print("{} fork {} and {} are available, now locked".format(Philosophers[num].r
        return True
    else:#print("{} forks NOT available".format(Philosophers[num].name))
        return False
def worker_t(_l, forklocks, num):# Worker function to try and acquire resource, [_l]:
    while threads_flag:
        if Philosophers[num].state == State[0]:#STARVING
            #print("{} Can I ask? ".format(Philosophers[num].name))
            if _l.locked() == False:#Are you available for asking for both forks?
                _l.acquire()
                res = CanIHaveBothForks(_l, forklocks, num)
                if res == True:# Here we have both forks!
                    leds(num,1)
                    Philosophers[num].state = State[1]#EATING   #move on machine state
```

```
                    print("{} changed state to: {}".format(Philosophers[num].name, Sta
                else:
                    time.sleep(0.5)#was 3
                _l.release()
            else:
                time.sleep(askagain_time)
        elif Philosophers[num].state == State[1]:#EATING
            print("{} Eating".format(Philosophers[num].name))
            blinkbydelay(eating_time, eating_blinkingRate, num)#time.sleep(eating_time
            Philosophers[num].state = State[2]#NAPPING
            print("{} Done Eating, changed state to: {}\n".format(Philosophers[num].na
            # Relinquish forks
            fork1 = Philosophers[num].forks.fork1[1];fork2 = Philosophers[num].forks.f
            forklocks[fork1-1].release();forklocks[fork2-1].release()
            leds(num,0)
        elif Philosophers[num].state == State[2]:#NAPPING
            print("{} Napping".format(Philosophers[num].name))
            blinkbydelay(napping_time, napping_blinkingRate, num)#time.sleep(napping_t
            Philosophers[num].state = State[3]#DONE
            print("\n{} Done Napping, changed state to: {}".format(Philosophers[num].r
            leds(num,1)
        elif Philosophers[num].state == State[3]:#DONE
            time.sleep(1)

        time.sleep(1)
async def get_btns(_loop):
    btns = base.btns_gpio
    global threads_flag, leds, btn0_flag, btn1_flag, btn2_flag, btn3_flag, Philosopher
    while threads_flag:
        await asyncio.sleep(0.1)
        if btns.read() != 0:
            if (btns[0].read()==1):
                await asyncio.sleep(0.3)
                print("button 0 pressed, Restart the Simulation")
                time.sleep(0.1)
                print("Simulation Re-started")
                enum_dining_sequence = enumerate(dining_sequence)# launch philosophers
                for idx, e in enum_dining_sequence:
                    Philosophers[idx].state = State[0]
                    time.sleep(0.2)
                    leds(idx,0)
                btn0_flag = True
            elif (btns[1].read()==1):
                await asyncio.sleep(0.3)
                print("button 1 pressed")
                btn0_flag = True
            elif (btns[2].read()==1):
                await asyncio.sleep(0.3)
                print("button 2 pressed")
                btn0_flag = True
            else:
                await asyncio.sleep(0.3)
                btn3_flag = True
                print("button 3 pressed, Exit program")
                threads_flag = False
                await asyncio.sleep(1)
                Names = ['Descartes','Aristotle','Socrates ','Thoreau  ','Th. Paine']
                for name in enumerate(Names):#Show a Status Summary
                    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[nam
                print("Forks lock status: {} {} {} {} {}".format(forklocks[0].locked()
```

```
                _loop.stop()

print("Press buttons 0-3\n   0 - Restart simulation\n   1 - No use\n   2 - No use\n

enum_ForkNumbers = enumerate(ForkNumbers)
for name in enumerate(Names):
    neededforks = forks(next(enum_ForkNumbers),next(enum_ForkNumbers))
    p = Philosopher(name[1], State[0],neededforks)
    Philosophers.append(p)

eating_time = random.randint(EATING_MIN, EATING_MAX)
print("Random eating_time: {}".format(eating_time))
napping_time = random.randint(DINING_MIN, eating_time)#napping is not longer than eati
print("Random napping_time: {}".format(napping_time))

for name in enumerate(Names):#Show a Status Summary
    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[name[0]].forks.fork

askingforboth = threading.Lock()#atomic locking/release
for n in range(5):
    leds(n,0)
    singlelock = threading.Lock()
    forklocks.append(singlelock)

for i in range(5):#Philosophers will show up to the table to be seated, every 2 second
    t = threading.Thread(target=worker_t, args=(askingforboth, forklocks, i))
    threads.append(t)

enum_dining_sequence = enumerate(dining_sequence)# launch philosophers' thread in an o
for idx, e in enum_dining_sequence:
    threads[e].start()

loop = asyncio.new_event_loop();  # Instance event_loop object
loop.create_task(get_btns(loop)); # take user input buttons

loop.run_forever()
loop.close()

for t in threads:
    t.join()
    print('{} joined'.format(t.name))

for name in enumerate(Names):#Show a Status Summary
    print(Philosophers[name[0]].name,"\thas\tForks ", Philosophers[name[0]].forks.fork

print("Forks lock status: {} {} {} {} {}".format(forklocks[0].locked(),forklocks[1].lo

##input("\n\t\t...press any key to exit...")
for n in range(5):
    leds(n,0)
print("Done.")
```

```
Press buttons 0-3
   0 - Restart simulation
   1 - No use
   2 - No use
   3 - Exit program
Random eating_time: 6
Random napping_time: 5
Descartes       has     Forks  1  and  2        State:  STARVING
Aristotle       has     Forks  2  and  3        State:  STARVING
Socrates        has     Forks  3  and  4        State:  STARVING
Thoreau         has     Forks  4  and  5        State:  STARVING
Th. Paine       has     Forks  5  and  1        State:  STARVING
Descartes fork 1 and 2 are available, now locked
Descartes changed state to: EATING
Socrates  fork 3 and 4 are available, now locked
Socrates  changed state to: EATING
Socrates  EatingDescartes Eating

Descartes Done Eating, changed state to: NAPPING
Socrates  Done Eating, changed state to: NAPPING


Socrates  Napping
Descartes Napping
Th. Paine fork 5 and 1 are available, now locked
Th. Paine changed state to: EATING
Aristotle fork 2 and 3 are available, now locked
Aristotle changed state to: EATING
Th. Paine Eating
Aristotle Eating

Socrates  Done Napping, changed state to: DONE

Descartes Done Napping, changed state to: DONE
Th. Paine Done Eating, changed state to: NAPPING

Aristotle Done Eating, changed state to: NAPPING

Thoreau   fork 4 and 5 are available, now locked
Thoreau   changed state to: EATING
Th. Paine Napping
Aristotle Napping
Thoreau   Eating

Th. Paine Done Napping, changed state to: DONE

Aristotle Done Napping, changed state to: DONE
Thoreau   Done Eating, changed state to: NAPPING

Thoreau   Napping

Thoreau   Done Napping, changed state to: DONE
button 0 pressed, Restart the Simulation
Simulation Re-started
Thoreau   fork 4 and 5 are available, now locked
Thoreau   changed state to: EATING
Thoreau   Eating
Descartes fork 1 and 2 are available, now locked
Descartes changed state to: EATING
Descartes Eating
```

```
Thoreau   Done Eating, changed state to: NAPPING

Socrates  fork 3 and 4 are available, now locked
Socrates  changed state to: EATING
Thoreau   Napping
Socrates  Eating
Descartes Done Eating, changed state to: NAPPING

Th. Paine fork 5 and 1 are available, now locked
Th. Paine changed state to: EATING
Descartes Napping
Th. Paine Eating

Thoreau   Done Napping, changed state to: DONE
Socrates  Done Eating, changed state to: NAPPING


Descartes Done Napping, changed state to: DONE
Aristotle fork 2 and 3 are available, now locked
Aristotle changed state to: EATING
Socrates  Napping
Aristotle Eating
Th. Paine Done Eating, changed state to: NAPPING

Th. Paine Napping

Socrates  Done Napping, changed state to: DONE

Th. Paine Done Napping, changed state to: DONE
Aristotle Done Eating, changed state to: NAPPING

Aristotle Napping

Aristotle Done Napping, changed state to: DONE
button 3 pressed, Exit program
Descartes       has     Forks  1  and  2        State:  DONE
Aristotle       has     Forks  2  and  3        State:  DONE
Socrates        has     Forks  3  and  4        State:  DONE
Thoreau         has     Forks  4  and  5        State:  DONE
Th. Paine       has     Forks  5  and  1        State:  DONE
Forks lock status: False False False False False
Thread-5 (worker_t) joined
Thread-6 (worker_t) joined
Thread-7 (worker_t) joined
Thread-8 (worker_t) joined
Thread-9 (worker_t) joined
Descartes       has     Forks  1  and  2        State:  DONE
Aristotle       has     Forks  2  and  3        State:  DONE
Socrates        has     Forks  3  and  4        State:  DONE
Thoreau         has     Forks  4  and  5        State:  DONE
Th. Paine       has     Forks  5  and  1        State:  DONE
Forks lock status: False False False False False
Done.
```

In [ ]: