

# BUILDING SCALABLE DISTRIBUTED SYSTEMS (NCHCS767)

## COURSEWORK (AE1)

### Table of Contents

<b>1.</b>	<b>QUESTION 1 – CONCURRENCY, THE DINNING PHILOSOPHERS’ PROBLEM .....</b>	<b>2</b>
1.1	SCENARIO .....	2
1.2	PROBLEM DEFINITION.....	2
1.3	PROBLEM SOLUTION: .....	2
1.4	PSEUDO CODE: .....	4
1.5	PYTHON CODE: .....	5
1.6	OUTPUT .....	6
1.7	ADVANCED CONSIDERATIONS. ....	7
1.8	GITHUB COMMIT LOG. ....	7
1.9	GITHUB LINK.....	8
1.10	HOW TO RUN THE CODE.....	8
1.11	REFLECTION.....	8
<b>2.</b>	<b>QUESTION 2 – MULTITHREADING FOR SCALABLE DATA ANALYSIS.....</b>	<b>8</b>
2.1	SCENARIO.....	8
2.2	PROBLEM DEFINITION .....	9
2.3	PROBLEM SOLUTION .....	9
2.4	PYTHON CODE: .....	9
2.5	OUTPUT:.....	11
2.6	GITHUB LOG:.....	11
2.7	GITHUB LINK: .....	11
2.8	HOW TO RUN THE CODE .....	11
2.9	REFLECTION .....	12

# 1. QUESTION 1 – Concurrency, The Dining Philosophers' Problem

## 1.1 Scenario

The dining Philosopher's problem is used in concurrency to demonstrate how deadlocks gets created due to issues with allocation of resources. In this scenario, certain philosophers are seated around a dining table and ready to eat spaghetti, using forks placed between each pair of them. They can either think or eat with two forks, but they cannot do both.

## 1.2 Problem Definition

The objective here, is to simulate this scenario, and experiment with the result, while detailing the steps taken to arrive at the solution.

## 1.3 Problem Solution:

In line with software development best practice, the development of this proposed solution would be broken down into some more manageable phases – usually Analysis, design, Implementation, Validation, and deployment / maintenance. The agile approach would also be utilised to add flexibility and rapidity.

### **Analysis**

In the analysis phase, a requirements elicitation was carried out, and it resulted in the following:

1. The system will comprise of Philosophers and Forks. These would be our process and resource Entities or Objects respectively.
2. Each Philosopher can pick up forks – this would require a pickFork() method or similar.
3. Each Philosopher can eat - this would require an eat() method or similar (constraint: 2 forks required).
4. Each Philosopher can stop eating - this would require a stopEating() method or similar.
5. Each Philosopher can think - this would require a think() method or similar.
6. Each Philosopher can stop thinking - this would require a stopThinking() method or similar.
7. Each Fork can be picked up – pickedUp() method or similar might be required.
8. Each Fork can be dropped – dropped() method or similar might be required.

Please note that it might not be imperative to implement all the methods outlined here in order to create the solution.

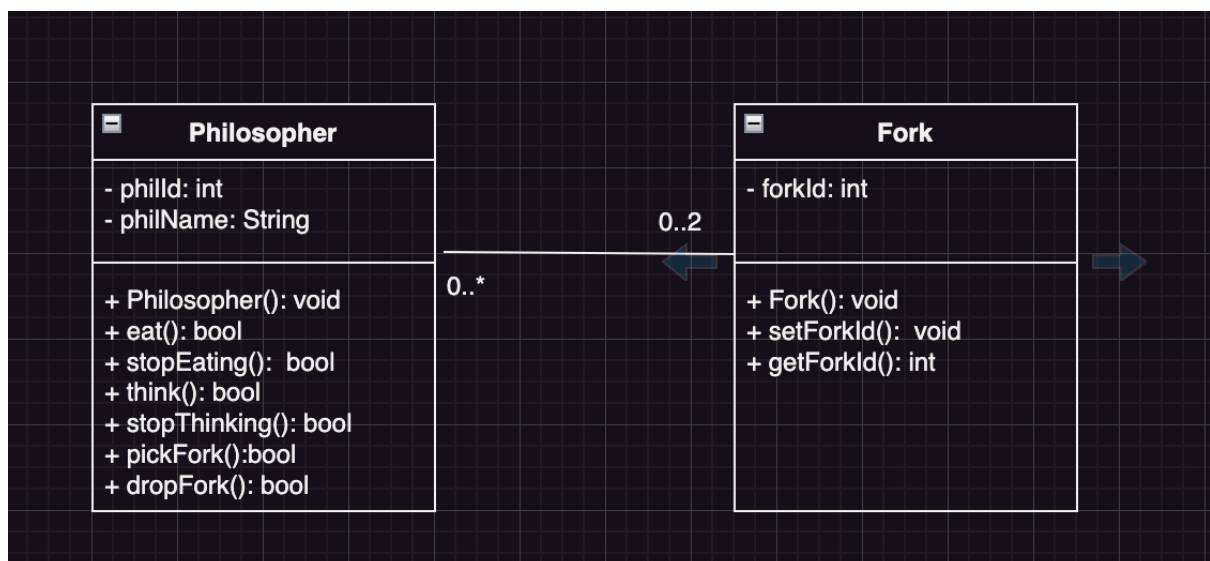
### **Design**

In the design phase, the simulation was conceptualised with the aid of two UML diagrams – The Use case diagram and the Class Diagram.

Figure 1: Use case diagram:

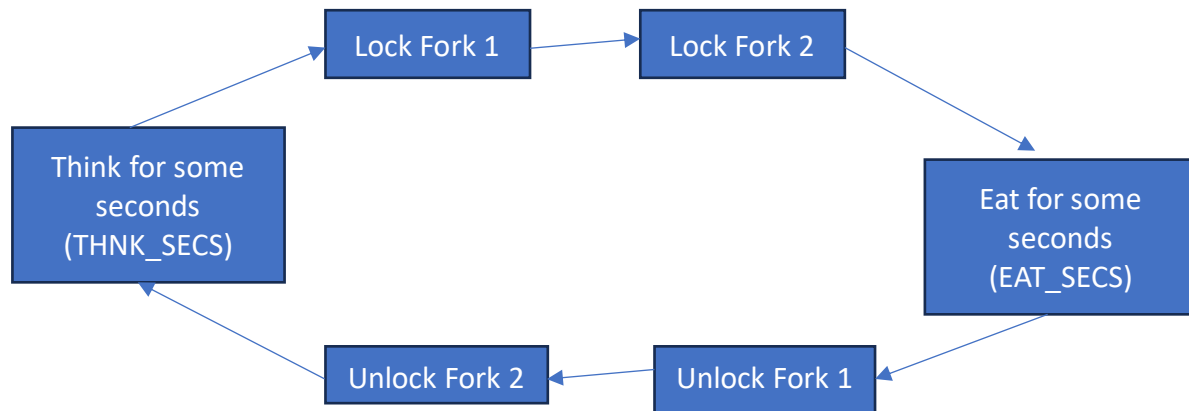


Figure 2: Class diagram.



The program flow will follow the pipeline portrayed below:

**Figure 3:** Program workflow



### Implementation

In the implementation phase, the pseudo code is first created as an initial step towards the development. This will facilitate and influence the flow of coding and subsequently, the choice of programming language or framework for the build.

#### 1.4 Pseudo Code:

```
#Assuming each philosopher is positioned on the table in such a way  
#that provides them access to 2 forks (one fork between two  
#philosophers)
```

```
#A philosophy function that accepts 3 argument, its ID and 2 forks  
philosopher(id, fork1, fork2):
```

```
#Loop between thinking, picking up forks, eating, and dropping, forks  
#in order.
```

```
while true:  
    think()  
    pickFork(fork1, fork2)  
    eat()  
    dropFork(fork1, fork2)
```

```
pickFork(fork1, fork2):  
    pick(fork1)  
    pick(fork2)
```

```
dropFork(fork1, fork2):  
    drop(fork1)
```

**drop(fork2)**

## 1.5 Python Code:

#Import libraries section

**import threading**

**import time**

#Create constant variables that are needed and set maximum values

#for objects and behaviours. These preset values can be used to

#observe the behaviour of the program under different parameters.

**THNK\_SECS = 3**

**TOTAL\_PHILS = 4**

**EAT\_SECS = 4**

#Create the Philosopher class which inherits from threading.Thread,

# and with a constructor that runs by default

#whenever the class is instantiated with no custom values.

**class Philosopher(threading.Thread):**

**def \_\_init\_\_(self, index, fork1, fork2):**

**threading.Thread.\_\_init\_\_(self)**

**self.index = index**

**self.fork1 = fork1**

**self.fork2 = fork2**

#Define other methods for this class to simulate the behaviour of

#the object, as shown on the UML class diagram.

**def run(self):**

**while True:**

**self.think()**

**self.eat()**

**def think(self):**

**print(f"Philosopher {self.index} : Can't eat now, I'm thinking")**

**time.sleep(THNK\_SECS)**

#The eat function first prints to indicate that a philosopher

#is starving before starting to eat. This helps to understand the

#thread operation better while the application is doing while running

**def eat(self):**

**print(f"Philosopher {self.index} : I'm starving")**

**with self.fork1:**

**with self.fork2:**

**print(f"Philosopher {self.index} : Can't think now, I'm eating")**

**time.sleep(EAT\_SECS)**

**print(f"Philosopher {self.index} : I'm done eating")**

```

#The main function that runs to kick off the program.
#create the number of philosophers specified by TOTAL_PHILS
#Start the threads, then call the join function on the
# philosopher instances,to make the thread wait until the
# current thread is completed

def main():
    forks = [threading.Lock() for _ in range(TOTAL_PHILS)]
    philosophers = [Philosopher(i, forks[i], forks[(i + 1) % TOTAL_PHILS]) for i in
range(TOTAL_PHILS)]

    for philosopher in philosophers:
        philosopher.start()

#Without the join method/function, the chances of deadlocks
# would be high.
    for philosopher in philosophers:
        philosopher.join()

if __name__ == "__main__":
    main()
#End of code.
#Code was run with different values for the global
# "constant variables"

```

## 1.6 Output

```

(base) aojegede@MacBook-Pro-4 ~ % /usr/bin/python3 "/Users/aojegede/Documents/Mac HD D
ata2/LSBU/NULONDON/08 NU-BSDS/Submission/philosopher.py"
Philosopher 0 : Can't eat now, I'm thinking
Philosopher 1 : Can't eat now, I'm thinking
Philosopher 2 : Can't eat now, I'm thinking
Philosopher 3 : Can't eat now, I'm thinking
Philosopher 0 : I'm starving
Philosopher 2 : I'm starving
Philosopher 1 : I'm starving
Philosopher 0 : Can't think now, I'm eating
Philosopher 3 : I'm starving
Philosopher 2 : Can't think now, I'm eating
Philosopher 2 : I'm done eating
Philosopher 2 : Can't eat now, I'm thinking
Philosopher 0 : I'm done eating
Philosopher 0 : Can't eat now, I'm thinking
Philosopher 3 : Can't think now, I'm eating
Philosopher 1 : Can't think now, I'm eating
Philosopher 2 : I'm starving
Philosopher 0 : I'm starving
Philosopher 1 : I'm done eating
Philosopher 1 : Can't eat now, I'm thinking
Philosopher 3 : I'm done eating

```

## 1.7 Advanced Considerations.

Should the resource, spaghetti be made finite, what happens?

In this case, the spaghetti would finish at some point, and there would be no eating time (i.e. EAT\_SECS would be equal to 0. We can achieve this by changing the constant EAT\_SECS into a variable instead and using a counter to decrement the value within a loop. When this happens, the philosophers would have to spend more time thinking, in the face of starvation, thereby, impairing their productivity.

## 1.8 GitHub commit log.

```
BSDS AE1 — -zsh — 80x34
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        AE1-UML.drawio
        DS AE1 Coursework.docx
        Readme.txt
        philosopher.py

nothing added to commit but untracked files present (use "git add" to track)
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git add .
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   AE1-UML.drawio
        new file:   DS AE1 Coursework.docx
        new file:   Readme.txt
        new file:   philosopher.py

(base) aojegede@MacBook-Pro-4 BSDS AE1 % git commit
[master (root-commit) 209fee2] This is the first commit on this coursework#
 4 files changed, 187 insertions(+)
 create mode 100644 AE1-UML.drawio
 create mode 100644 DS AE1 Coursework.docx
 create mode 100644 Readme.txt
 create mode 100644 philosopher.py
(base) aojegede@MacBook-Pro-4 BSDS AE1 %
```

```
BSDS AE1 — -zsh — 99x34
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git log
commit f2e9c15bfd88ef7db0914f8fb8a9a60f82e68378 (HEAD -> main, origin/main, origin/HEAD)
Author: AnthonyJegede <anthonyojegede@gmail.com>
Date: Sat Nov 18 07:57:01 2023 +0100

    First commit to main

commit 209fee2fed3f6cadd440871d67e814a0551d8343
Author: AnthonyJegede <anthonyojegede@gmail.com>
Date: Sat Nov 18 06:08:46 2023 +0100

    This is the first commit on this coursework#
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git log:
git: 'log:' is not a git command. See 'git --help'.

The most similar command is
    log
```

## 1.9 GitHub Link

[https://github.com/AnthonyJegade/nu\\_bsd\\_s\\_ae1.git](https://github.com/AnthonyJegade/nu_bsd_s_ae1.git)

### 1.10 How to run the code

This application is a simulation of the dining philosophers' program for demonstration deadlock issues in concurrency.

To run this application:

Open in VSCODE or other python run time environment or IDE

Right-click anywhere on the document and select RUN PYTHON followed by RUN PYTHON FILE.

Alternatively, you may run the program using the Command line Interface (CLI) by shelling into the file location and then typing python3 followed by the file name on the command line, i.e:

```
>Python3 philosopher.py
```

### 1.11 Reflection

During this coursework, my understanding of concurrency deepened, especially on how to manage multiple threads running, such that deadlocks are avoided. I was able to look closely at the threading.Thread superclass to understand how the philosophers' sub-class took advantage of the methods. I learnt that every task would look simply until you get down on it and start doing it. I was more familiar with Java, and so, I wanted to do it in Java because there would be no steep learning curve, but I later decided to go for python. Although the steep learning curve slowed me down a bit, it was well worth it since my python skills have received a boost as a result of this decision.

## 2. QUESTION 2 – Multithreading for Scalable Data Analysis

### 2.1 Scenario

The MapReduce algorithm helps divide or split tasks to other systems such that all the nodes perform the same task. Reduced nodes are the used to shuffle and collate the results.



## 2.2 Problem Definition

The objective here, is to simulate a MapReduce operation of this algorithm on a single computer by taking advantage of the multithreading capabilities of the machine, especially owing to the multiple CPU cores.

## 2.3 Problem Solution

The following would be required:

**A splitter function:** This will be responsible for splitting the dataset or large document into fragments. This process must be devoid of redundancy.

**A mapper function:** For analysing the fragmented dataset comparatively.

**A reducer function:** For combining the outcome of the preceding operation and correlating related words in such a way that reduces the results.

**A controller:** The job of the controller is for the creation and efficient operation and management of threads.

In the next section, Python code is used to simulate this process.

## 2.4 Python Code:

```
#This is the import libraries section
#These are necessary super classes for the subclasses
#and function work properly.
import concurrent.futures
from collections import Counter

# Store an example text (the main document) inside a variable,
# in this case, we call it - "main_doc"
main_doc = "My name is Anthony and Anthony is studying computer science. ... (main document)"

# Now we use the splitter function to split the main_doc
# into two parts or pages for simplicity
def splitter(text, num_pages):
    page_size = len(text) // num_pages
    pages = []
    start_idx = 0
    for _ in range(num_pages - 1):
        end_idx = start_idx + page_size
        if end_idx >= len(text):
```

```

    # In the event that end_idx > the length of text.
    pages.append(text[start_idx:])
    return pages
while end_idx < len(text) and text[end_idx] != ' ':
    end_idx += 1
pages.append(text[start_idx:end_idx])
start_idx = end_idx + 1 if end_idx < len(text) and text[end_idx] == ' ' else end_idx
pages.append(text[start_idx:])
return pages

```

# Next, we use the Mapper function to create our bag of words  
# i.e calculating how frequently each word occur in the txt or corpus.

```

def mapper(page):
    words = page.split()
    return Counter(words)

```

# This function (reducer function), brings together the counted words from the fragmented  
# text (pages) and creates a kind of summarisation.

```

def reducer(results):
    final_count = Counter()
    for count in results:
        final_count.update(count)
    return final_count

```

#This is the main function that runs when the program starts.  
# Here, all the major functions are called to perform their  
# tasks in order,

```

def main():
    num_pages = 5
    pages = splitter(main_doc, num_pages)

```

```

    # Map
    with concurrent.futures.ThreadPoolExecutor() as executor:
        mapped_results = executor.map(mapper, pages)

```

```

    # Store the returned value of the reducer function
    reduced_result = reducer(mapped_results)

```

```

    # Printing the bag of words
    print("Bag of words:") # This is the label
    print(reduced_result) # This is the actual result

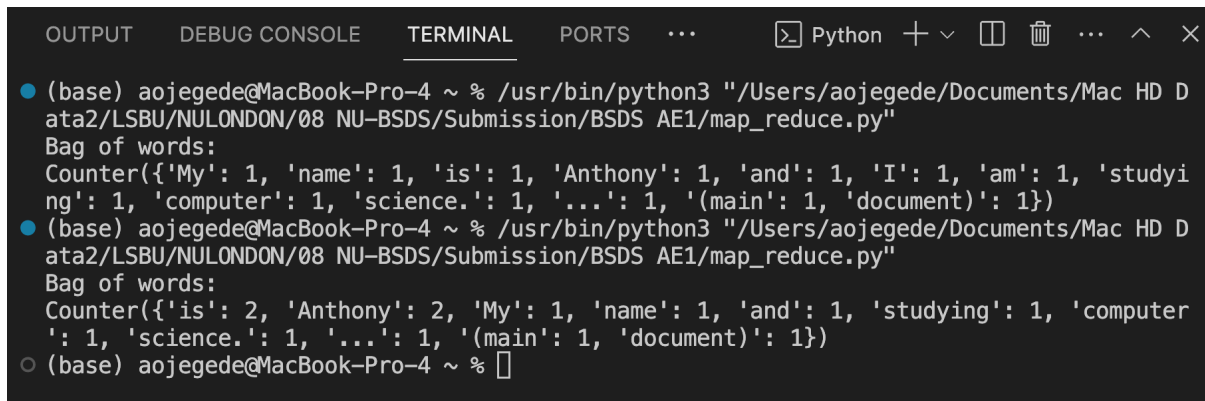
```

```

if __name__ == "__main__":
    main()
# main function gets called here if true.

```

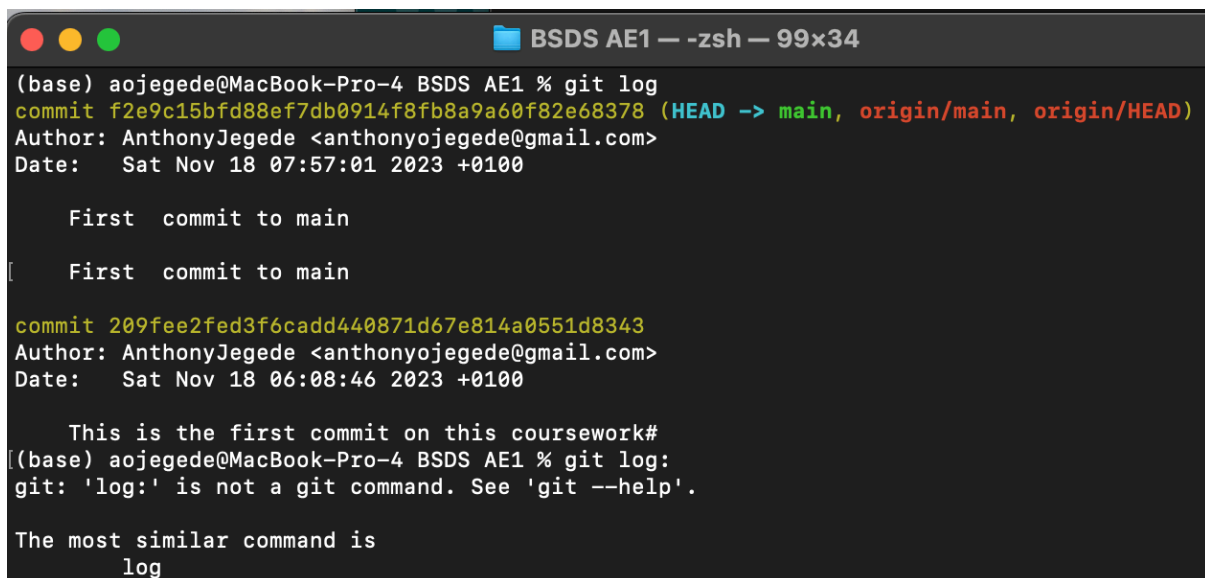
## 2.5 Output:



```
OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  ...  Python + - [ ] [ ] ... ^ X

• (base) aojegede@MacBook-Pro-4 ~ % /usr/bin/python3 "/Users/aojegede/Documents/Mac HD D
ata2/LSBU/NULONDON/08 NU-BSDS/Submission/BSDS AE1/map_reduce.py"
Bag of words:
Counter({'My': 1, 'name': 1, 'is': 1, 'Anthony': 1, 'and': 1, 'I': 1, 'am': 1, 'studyi
ng': 1, 'computer': 1, 'science.': 1, '...': 1, '(main': 1, 'document)': 1})
• (base) aojegede@MacBook-Pro-4 ~ % /usr/bin/python3 "/Users/aojegede/Documents/Mac HD D
ata2/LSBU/NULONDON/08 NU-BSDS/Submission/BSDS AE1/map_reduce.py"
Bag of words:
Counter({'is': 2, 'Anthony': 2, 'My': 1, 'name': 1, 'and': 1, 'studying': 1, 'computer
': 1, 'science.': 1, '...': 1, '(main': 1, 'document)': 1})
○ (base) aojegede@MacBook-Pro-4 ~ %
```

## 2.6 GitHub Log:



```
BSDS AE1 — zsh — 99x34

(base) aojegede@MacBook-Pro-4 BSDS AE1 % git log
commit f2e9c15bfd88ef7db0914f8fb8a9a60f82e68378 (HEAD -> main, origin/main, origin/HEAD)
Author: AnthonyJegede <anthonyojegede@gmail.com>
Date: Sat Nov 18 07:57:01 2023 +0100

    First commit to main

[
    First commit to main

commit 209fee2fed3f6cadd440871d67e814a0551d8343
Author: AnthonyJegede <anthonyojegede@gmail.com>
Date: Sat Nov 18 06:08:46 2023 +0100

    This is the first commit on this coursework#
(base) aojegede@MacBook-Pro-4 BSDS AE1 % git log:
git: 'log:' is not a git command. See 'git --help'.

The most similar command is
log
```

## 2.7 GitHub Link:

[https://github.com/AnthonyJegede/nu\\_bsd\\_s\\_ae1.git](https://github.com/AnthonyJegede/nu_bsd_s_ae1.git)

## 2.8 How to run the code:

This program is a simulation of Multithreading for Scalable Data Analysis

To run this application:

Open in VSCODE or other python run time environment or IDE

Right-click anywhere on the document and select RUN PYTHON followed by RUN PYTHON FILE.

Alternatively, you may run the program using the Command line Interface (CLI) by shelling into the file location and then typing python3 followed by the file name on the command line, i.e:

```
>Python3 map_reduce.py
```

## 2.9 Reflection

During and after this second part of the work, my python skills grew even further. The most amazing part of my experience is the relevance of the map reduce algorithm to my final project which is all about question answering systems. I really wished I had this knowledge prior to the submission of my main project proposal.

What I would really love to do more with this knowledge, were time not a constraint, is to analyse a lot of corpuses from file and directly from online repositories.