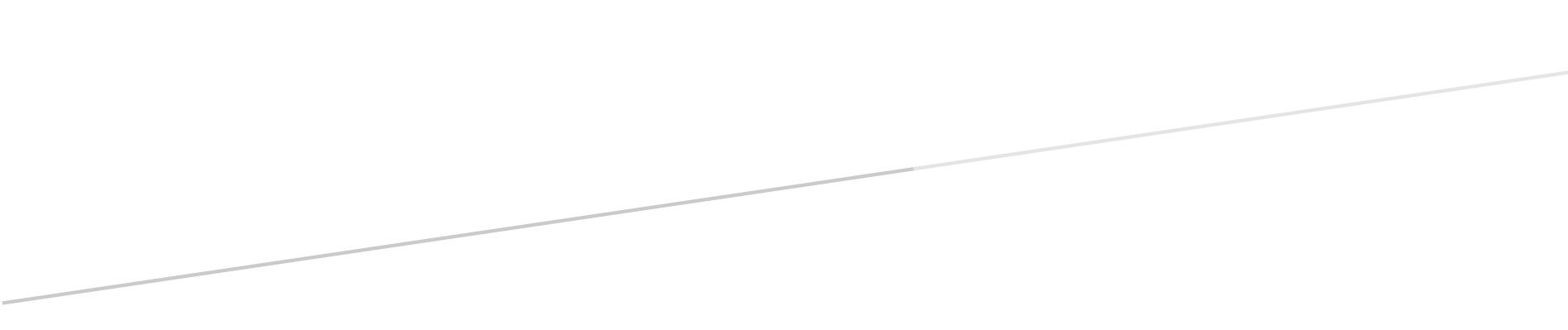
­



**Alex Carbajal**

**Washington State University Tri-Cities**

**April 30, 2024**

Homework 4

**Table of Contents:**

|  |
| --- |
|  |
| 1. Introduction |
| 1.0 Purpose |
|  |
| 1. Problems |
| 2.0 Data Abstraction |
| 2.1 Synchronization |
| 2.2 Experiments |
| 2.3 Deadlock |
|  |
| 1. Design/Solution |
| 3.0 Design Overview |
| 3.1 Design Process |
| 3.1.1 Data Abstraction |
| 3.1.2 Synchronization |
| 3.1.3 Experiments |
| 3.1.4 Deadlock |
|  |
| 1. Experiment Result |
|  |
| 1. Experiment Result Analysis |
|  |
| 1. Analysis of Learning |
|  |
| Appendix |
| Source Code |
| Commands for Running |

**1 Introduction:**

This homework’s focus is on problems related to concurrency and the exploration of solutions to those problems. These problems include concepts such as synchronization, resource sharing, and deadlock. The complication used to explore these problems in this document is the dining philosophers problem.

1. Purpose:

The goal of this document is to outline the problems and solutions to the ideas that the dining philosophers problem tackles. Experiments are performed in order to analyze the solutions and their results.

**2 Problems:**

2.0 Data Abstraction: To abstract the data for the dining philosophers problem, I implemented several classes in C++ that work in conjunction to execute the processes of the dining philosophers. The main class that does the work of concurrent programming is the Philosopher class. This class includes several member variables that indicate important data about a Philosopher object and several methods that run the instructions required for the dining problem. Each Philosopher runs in its own thread and works concurrently with multiple other Philosophers. The Philosophers share mutexes between each other to ensure the program meets the expectations of the dining problem and no erroneous experiences occur. The Philosopher class utilizes the Syncro (more information in 2.1) and Chopstick classes to share data between the Philosophers. Each Philosopher has a left and right Chopstick and each Chopstick object includes a mutex to ensure they can only be accessed by one Philosopher at a time. The method that executes all the instructions for the Philosopher is the *run* method, which runs a loop for a specified amount of time that performs the action of the Philosopher (think, take Chopsticks, eat, and release Chopsticks).

2.1 Synchronization: The system that keeps all the Philosophers synchronized and behaving as expected is the Syncro class. The Syncro class is relatively simple and contains a limited amount of variables and methods to keep it quick and easy to manage. The only two variables it has are the *chopsticks* and *states* variables. They both are lists of a size corresponding to the total number of Philosophers. One object of this class is shared between all of the Philosophers in order to keep them synchronized by sharing each Philosopher’s state and the state of the Chopsticks with each other. If a Philosopher want to pick up or put down a Chopstick, they have to perform that action through this class. The class checks the status of each Chopstick being picked up or put down to ensure there is no violation of the rules. A Philosopher can also check the state of other Philosophers through this class (e.g. is another Philosopher eating?). This is how the Philosophers are kept in sync with each other.

2.2 Experiments: The results of the experiments and the analysis of the results are displayed in sections 4 and 5 below, respectively. The results come from running the program for 20 minutes with an eating time of 0.5 seconds, thinking time of 1 second, and N = 5, 6, 7, and 16. The recorded times are in milliseconds.

2.3 Deadlock: To prevent deadlock from occurring in the program, it is important to make sure that all of the Philosophers get an opportunity to pick up the Chopsticks and eat and are not left starving for the entire running time of the program. There are a multitude of ways to design a solution for deadlock, though I believe it is best to try and keep it simple so in the case that a deadlock does occur, debugging it will be simpler.

**3 Design/Solution:**

3.0 Design Overview: This section goes over the design process and solution to each of the problems listed in section 2. Each solution was designed diligently by carefully following the instructions stated in the homework document in order to ensure accuracy.

3.1 Design Process:

3.1.1 Data Abstraction: In order to design the Philosopher class, I first considered how each of the Philosophers (threads) should be able to communicate with one another, since that is a vital part of running a concurrent program smoothly. I decided to have all Philosophers share a single Syncro object which will synchronize all their actions together. For the *run* method, I had each Philosopher perform four actions for a specified amount of time. Those actions are the methods *think*, *take\_chopsticks*, *eat*, and *release\_chopsticks*. Each one of these methods have checks that analyze the state of the Philosopher to determine which action to take and what to print out. The *test* method is the method for checking if a Philosopher can and should pickup both Chopsticks. For my solution, I decided to check if both the neighbors of the Philosopher are eating and to only pickup the Chopsticks if both are currently not eating. This ensures that the correct Chopsticks are picked up only when it is appropriate.

3.1.2 Synchronization: To design the Syncro class, I had to make sure the class had all of the data required to be shared between the Philosophers, otherwise the concurrency in the program would not work properly. The only data required to be shared is the state of each of the Philosophers and the Chopsticks that the Philosophers are sharing. For the methods of the class, the only ones that I found necessary to implement were getters, setters, and methods to put down and pick up the chopsticks. The methods *putDownChopstick* and *pickUpChopstick* both perform checks to make sure the Chopstick is in the opposite state before being put down up or picked up.

3.1.3 Experiments: To perform all of the experiments, I ran each test for 20 minutes and awaited the results at the end. I received the results by having each thread print out the eating, thinking, and starving time of its respective Philosopher after the Philosopher finished all of its other processes. I kept track of the times with variables within the class that added up the time that a Philosopher spent on a certain action.

3.1.4 Deadlock: My solution to the deadlock program involves two parts. The first part is present in the *test* method and involves the check done to make sure a Philosopher can pickup both chopsticks in a valid way. The check analyzes the state of both of the neighbors of the Philosopher attempting to pick up the Chopsticks by making sure that they are not currently in the eating state. If they are in the eating state, then the Philosopher simply skips picking up the Chopsticks and reattempts it the next time it runs *test*. The second part involves forcing a Philosopher that has been in the starving state for too long to pickup the Chopsticks as soon as possible. It does this in the *take\_chopsticks* method and it works by running a while loop until the state of the Philosopher is in the eating state, which means that they have successfully picked up the Chopsticks. Both of these parts work well together to prevent any deadlock from occurring. I did not experience any deadlocks after implementing this solution.

**4 Experiment Result:**

**Results for N = 5:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Philosopher 0 | Philosopher 1 | Philosopher 2 | Philosopher 3 | Philosopher 4 |
| Eating | 1.95e8 | 1.92e8 | 1.91e8 | 1.88e8 | 2.17e8 |
| Thinking | 3.86e8 | 3.87e8 | 4.02e8 | 3.9e8 | 3.7e8 |
| Starving | 1.95e8 | 1.93e8 | 1.69e8 | 1.87e8 | 1.89e8 |

**Results for N = 6:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Philosopher 0 | Philosopher 1 | Philosopher 2 | Philosopher 3 | Philosopher 4 | Philosopher 5 |
| Eating | 1.88e8 | 2.06e8 | 1.94e8 | 1.92e8 | 2.12e8 | 2.08e8 |
| Thinking | 3.94e8 | 4.06e8 | 3.84e8 | 3.8e8 | 3.9e8 | 3.9e8 |
| Starving | 1.94e8 | 1.78e8 | 2.06e8 | 2.02e8 | 1.72e8 | 1.91e8 |

**Results for N = 7:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Philosopher 0 | Philosopher 1 | Philosopher 2 | Philosopher 3 | Philosopher 4 | Philosopher 5 | Philosopher 6 |
| Eating | 1.89e8 | 1.92e8 | 1.94e8 | 1.99e8 | 2.13e8 | 2.09e8 | 2.07e8 |
| Thinking | 3.93e8 | 4.26e8 | 4.12e8 | 3.9e8 | 4.32e8 | 3.81e8 | 4.02e8 |
| Starving | 1.83e8 | 1.65e8 | 1.86e8 | 2.04e8 | 1.83e8 | 2.08e8 | 1.79e8 |

**Results for N = 16:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Philosopher 0 | Philosopher 1 | Philosopher 2 | Philosopher 3 | Philosopher 4 | Philosopher 5 | Philosopher 6 | Philosopher 7 | Philosopher 8 | Philosopher 9 | Philosopher 10 | Philosopher 11 | Philosopher 12 | Philosopher 13 | Philosopher 14 | Philosopher 15 |
| Eating | 2.06e8 | 1.87e8 | 1.83e8 | 2e8 | 2.01e8 | 2.05e8 | 1.88e8 | 2.12e8 | 1.95e8 | 1.85e8 | 1.87e8 | 1.74e8 | 2e8 | 1.92e8 | 2.12e8 | 1.87e8 |
| Thinking | 4.08e8 | 3.92e8 | 3.9e8 | 3.88e8 | 3.87e8 | 4.14e8 | 3.78e8 | 4.1e8 | 3.88e8 | 3.96e8 | 3.82e8 | 3.74e8 | 4.22e8 | 3.78e8 | 4.12e8 | 3.7e8 |
| Starving | 1.79e8 | 2.01e8 | 1.95e8 | 2.02e8 | 2.05e8 | 1.53e8 | 1.91e8 | 1.82e8 | 2.02e8 | 2.12e8 | 1.79e8 | 2.14e8 | 1.75e8 | 2.13e8 | 1.7e8 | 2.26e8 |

**5 Experiment Result Analysis:**

The only issue I faced with my experiments is that all the Philosophers spent some time starving, although the starving times are very close to being the same for all of the Philosophers, so it did not occur in an unfair manner. This most likely happened because the coin flip to put down the chopsticks had unfortunate luck and failed multiple times in a row which resulted in a neighbor Philosopher having to wait for longer to eat than they normally have to. Besides that, the eating and thinking times between the Philosophers in all the experiments are fairly close to each other, meaning that no single Philosopher spent a significantly larger portion of time eating or thinking than the others. Also, in all of the experiments, a deadlock never occurred.

* 1. **6 Analysis of Learning:**

In this programming assignment I got the opportunity to learn about concurrent programming and the issues that could plague a program that implements this style of programming. Running a program concurrently can be exceedingly beneficial since it often results in significantly faster program execution due to the program utilizing more processors at once than if it was not written in a concurrent manner. The issues with concurrent programming come from the fact it is often more difficult to write programs in a way where they utilize concurrency effectively without significant issues ensuing. Issues include elements such as resource sharing between processes and deadlock. These issues are avoidable and there are solutions to them, but they present additional concerns that can make a program harder to read and maintain due to the greater complexity of the problems. After completion of this assignment, I recognize the importance of concurrency, the fundamentals of how to implement a concurrent program, and how concurrency works.

* 1. **Appendix**

**Source Code:**

**hw4.cpp:**

#include <iostream>

#include <string>

#include "philosopher.hpp"

#include "syncro.hpp"

#include "helpers.hpp"

int main(int argc, char \*argv[])

{

Philosopher \*philosophers[NUM\_PHILOSOPHERS];

Syncro syncro;

// names for the philosophers

const std::string nameArray[] = {"Yoda", "Obi-Wan", "Rey", "Kanan", "Leia", "Luke", "Ahsoka",

"Mace Windu", "Ezra", "Palpatine", "Anakin", "Kylo Ren", "Dooku",

"Kit Fitso", "Luminara", "Plo Koon", "Revan", "Thrawn", "Zeb", "Sabine"};

// allocate philosophers[]

for (int i = 0; i < NUM\_PHILOSOPHERS; i++)

{

philosophers[i] = new Philosopher(nameArray[i], i, syncro.getChopstick(getLeft(i)), syncro.getChopstick(getRight(i)), syncro);

}

// free the memory for philosophers[]

for (int i = 0; i < NUM\_PHILOSOPHERS; i++)

{

delete philosophers[i];

}

return 0;

}

**philosopher.hpp:**

#pragma once

#include <iostream>

#include <string>

#include <chrono>

#include <thread>

#include <mutex>

#include <unistd.h>

#include "syncro.hpp"

#include "chopstick.hpp"

#include "helpers.hpp"

std::mutex outputMutex; // synchronize cout print statements

class Philosopher : std::thread

{

private:

std::string name;

int id;

int state;

double eatTime; // total amount of time spent eating

double thinkTime; // total amount of time spent thinking

double starveTime; // total amount of time spent starving (wants to eat but can't)

double currentThinkTime;

double currentStarveTime;

Syncro &syncro; // Syncro object that is shared among all philosophers

Chopstick &left, &right; // identify the connections with the set of chopsticks

std::thread mainThread;

// philosopher is thinking

void think(int id);

// attempt to take the left and right chopsticks

void take\_chopsticks(int id);

// perform the eating action

void eat(int id);

// put down the left and right chopsticks so other philosophers can utilize them

void release\_chopsticks(int id);

// check status of a philosopher neighbor and potentially change status - used to know when to request the chopsticks

void test(int id);

// print out information about the philosopher

void printStatistics(void);

public:

Philosopher(std::string newName, int newId, Chopstick &newLeft, Chopstick &newRight, Syncro &newSyncro)

: syncro(newSyncro), left(newLeft), right(newRight), mainThread(&Philosopher::run, this)

{

name = newName;

id = newId;

state = THINKING;

eatTime = 0.0;

thinkTime = 0.0;

starveTime = 0.0;

currentThinkTime = 0.0;

currentStarveTime = 0.0;

}

~Philosopher()

{

mainThread.join();

}

void run(void);

};

void Philosopher::think(int id)

{

if (state == THINKING && currentThinkTime >= IS\_HUNGRY\_TIME)

{

currentThinkTime = 0.0;

state = HUNGRY;

syncro.setState(id, state);

outputMutex.lock();

std::cout << "philosopher " << id << " is hungry\n";

outputMutex.unlock();

}

else if (state == THINKING)

{

outputMutex.lock();

std::cout << "philosopher " << id << " is thinking\n";

outputMutex.unlock();

double randomThinkTime = randomTime(MIN\_RANGE, MAX\_RANGE);

// double randomThinkTime = 1.0 \* 1e6; // thinking time of 1 second

usleep(randomThinkTime); // pause for a certain amount of time

currentThinkTime += randomThinkTime;

thinkTime += randomThinkTime;

}

}

void Philosopher::take\_chopsticks(int id)

{

if (state == HUNGRY || state == STARVING)

{

test(id);

// philosopher couldn't eat, add to time until starving

if (state != EATING)

{

double randomStarveTime = randomTime(MIN\_RANGE, MAX\_RANGE);

usleep(randomStarveTime);

currentStarveTime += randomStarveTime;

if (state == STARVING)

{

starveTime += randomStarveTime;

}

// begin starving

if (state != STARVING && currentStarveTime >= IS\_STARVING\_TIME)

{

state = STARVING;

outputMutex.lock();

std::cout << "philosopher " << id << " is starving!\n";

outputMutex.unlock();

starveTime += (currentStarveTime - IS\_STARVING\_TIME);

}

// starving for too long - force the philosopher to eat

if (currentStarveTime >= IS\_STARVING\_TIME \* 2)

{

while (state != EATING)

{

test(id);

}

}

}

}

}

void Philosopher::eat(int id)

{

if (state == EATING)

{

outputMutex.lock();

std::cout << "philosopher " << id << " is eating\n";

outputMutex.unlock();

double randomEatTime = randomTime(MIN\_RANGE, MAX\_RANGE);

// double randomEatTime = 0.5 \* 1e6; // eating time of 0.5 seconds

usleep(randomEatTime);

eatTime += randomEatTime;

}

}

void Philosopher::release\_chopsticks(int id)

{

if (state == EATING)

{

state = THINKING;

syncro.setState(id, state);

left.unlockChopstick();

right.unlockChopstick();

outputMutex.lock();

std::cout << "philosopher " << id << " finished eating\n";

outputMutex.unlock();

}

}

void Philosopher::test(int id)

{

int leftNeighbor = getLeft(id);

int rightNeighbor = getRight(id);

if ((state == HUNGRY || state == STARVING) &&

syncro.getState(leftNeighbor) != EATING && syncro.getState(rightNeighbor) != EATING)

{

state = EATING;

syncro.setState(id, state);

left.lockChopstick();

right.lockChopstick();

outputMutex.lock();

std::cout << "philosopher " << id << " picked up chopsticks "

<< leftNeighbor << " & " << rightNeighbor << std::endl;

outputMutex.unlock();

// reset the starve time

if (currentStarveTime >= 0.0)

{

currentStarveTime = 0.0;

}

}

}

void Philosopher::printStatistics(void)

{

std::cout << "\n\*\*\* Philosopher " << id << " Statistics \*\*\*\n";

std::cout << "Eating Time: " << eatTime << "ms\n";

std::cout << "Thinking Time: " << thinkTime << "ms\n";

std::cout << "Starving Time: " << starveTime << "ms\n\n";

}

void Philosopher::run(void)

{

// for keeping track of time

auto runStart = std::chrono::high\_resolution\_clock::now();

auto runEnd = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> currentTime = runEnd - runStart;

// let the philosopher dine for a certain amount of time TOTAL\_RUN\_TIME

while (currentTime.count() < TOTAL\_RUN\_TIME)

{

// perform the actions

think(id);

take\_chopsticks(id);

eat(id);

release\_chopsticks(id);

// get the current elapsed time

runEnd = std::chrono::high\_resolution\_clock::now();

currentTime = runEnd - runStart;

}

// force the chopsticks to be put down to prevent possible freezes

while (state == EATING)

{

release\_chopsticks(id);

}

usleep(1 \* 1e6);

outputMutex.lock();

printStatistics();

outputMutex.unlock();

}

**syncro.hpp:**

#pragma once

#include "chopstick.hpp"

#include "helpers.hpp"

class Syncro

{

private:

Chopstick chopsticks[NUM\_PHILOSOPHERS];

int states[NUM\_PHILOSOPHERS] = {THINKING}; // the state of each of the philosophers

public:

// getters

Chopstick &getChopstick(int id);

int getState(int id);

// setters

void setState(int id, int newState);

// validate conditions and put down chopstick id

void putDownChopstick(int id);

// validate conditions and pick up chopstick id

void pickUpChopstick(int id);

};

Chopstick &Syncro::getChopstick(int id)

{

return chopsticks[id];

}

int Syncro::getState(int id)

{

return states[id];

}

void Syncro::setState(int id, int newState)

{

states[id] = newState;

}

void Syncro::putDownChopstick(int id)

{

if (chopsticks[id].getStatus() == CHOPSTICK\_UP)

{

chopsticks[id].unlockChopstick();

}

}

void Syncro::pickUpChopstick(int id)

{

if (chopsticks[id].getStatus() == CHOPSTICK\_DOWN)

{

chopsticks[id].lockChopstick();

}

}

**chopstick.hpp:**

#pragma once

#include <mutex>

#include "helpers.hpp"

class Chopstick

{

private:

std::mutex chopTex;

int status;

public:

Chopstick()

{

status = CHOPSTICK\_DOWN;

}

// getters

int getStatus(void);

void unlockChopstick();

void lockChopstick();

};

int Chopstick::getStatus(void)

{

return status;

}

void Chopstick::unlockChopstick()

{

chopTex.unlock();

status = CHOPSTICK\_DOWN;

}

void Chopstick::lockChopstick()

{

chopTex.lock();

status = CHOPSTICK\_UP;

}

**helpers.hpp:**

#pragma once

#include <random>

#define NUM\_PHILOSOPHERS 5

// chopstick states

#define CHOPSTICK\_DOWN 1

#define CHOPSTICK\_UP 0

// time in milliseconds

#define TOTAL\_RUN\_TIME 60.0 // the philosophers will eat for this amount of time

#define IS\_HUNGRY\_TIME 6.0 \* 1e6 // the philosopher gets hungry after this amount of time

#define IS\_STARVING\_TIME 4.0 \* 1e6 // the philosopher is starving after this amount of time spent hungry

// range to generate random numbers in

#define MIN\_RANGE 1.0

#define MAX\_RANGE 5.0

// a philosopher is always in one of these states

enum States

{

THINKING = 0,

EATING = 1,

HUNGRY = 2,

STARVING = 3 // occurs after being hungry for a certain amount of time

};

// get the left neighbor of id

int getLeft(int id)

{

return (id - 1 + NUM\_PHILOSOPHERS) % NUM\_PHILOSOPHERS;

}

// get the right neighbor of id

int getRight(int id)

{

return (id + 1) % NUM\_PHILOSOPHERS;

}

// generate a random integer that is either 0 or 1

int randomCoinFlip(void)

{

std::random\_device seed; // random seed

std::mt19937 generate(seed()); // seed the random number generation engine

std::uniform\_int\_distribution<> range(0, 1); // set the range for integers

return range(generate); // generate a random number within the range

}

// generate a random double between min and max (time is in milliseconds)

double randomTime(double min, double max)

{

std::random\_device seed;

std::mt19937 generate(seed());

std::uniform\_real\_distribution<> range(min, max); // set the range for real numbers

return range(generate) \* 1e6;

}

**Makefile:**

# Name of executable

BINS = hw4

# Setting the flags

CFLAGS = -g -Wall -Wextra -Wstrict-prototypes

# Link in support for debugging

LDFLAGS = -g

# Default target produced by entering "make" alone

.PHONY: default

default: $(BINS)

# Compile \*.cpp into \*.o

src/%.o: src/%.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

# Remove \*.o files, but leave executable

.PHONY: clean

clean:

rm -f core\* src/\*.o \*~

# Link \*.o files into an executable

hw4: src/hw4.o

$(CXX) $(CXXFLAGS) $^ -o $@

# Run tests

test: hw4

./hw4

# Check for memory leaks

mem: hw4

valgrind --leak-check=full --show-leak-kinds=all -s ./hw4

# Remove all files that can be reconstructed through "make"

.PHONY: immaculate

immaculate: clean

rm -f $(BINS)

* 1. **Commands for Running**: A Makefile is provided, so simply type *make* in the directory to build the executable. Run the executable with *./hw4*. This program was built on and is expected to run on a Linux system, though it might work on others (I did not perform tests for this).