Bonus Assignment

EE 463 – HA

(Operating Systems Lab)

Instructor: Eng. Turki Abdulhafiz

Submission Date: December 9, 2024

Mahdi Bathallath

2236809

Introduction:

The Dining Philosophers a common problem when considering multithreading and its problems. The basic idea is that there are 5 philosophers and 5 forks (eating utensils). At any given moment a philosopher is either thinking, eating or hungry. If they are hungry, they need to pick both forks from either side. A problem arises when adjacent philosophers are hungry and/or eating (mutually exclusive) since there would be a shortage in the required 2 forks. Thus, the philosopher needs to wait until the resource is released. Moreover, a group of three philosophers share any of the 5 forks so there might be contention over who gets the resource. This is known as a race condition. A mutex may be used to lock and unlock the resource and avoid the race condition when acquiring the resource. Another problem arises if one of the philosophers does not get to eat causing starvation. To avoid this issue a queue on each fork may be used, ensuring that the forks are acquired on a first-come first serve bases. Lastly, the process might stall completely if each philosopher has one fork (deadlock). To avoid this issue even numbered philosophers will check the fork with the same ID then the one after it. On the other hand, odd numbered philosophers will check the fork of the following ID before the similar one. By doing this common multithreading issues are mitigated. Thus, the final code should address the following:

1. Memory Leaks
2. Starvation
3. Race Conditions
4. Deadlocks

The Code:

Figure 1 shows the libraries needed for this code to work. The pthread library is needed to use the required mutex. On the other hand, the signal library is used to handle the interrupt invoked when the user cancels the process using Ctrl+C. The stdbool library is included to replace 0 &1 with their Boolean equivalent.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>
#include "queue.h"

#define NUM 5
```

Figure 1: Screenshot of Libraries Used

For readability an enumeration of the states was created to define the state of the philosopher. A C structure is used to define the characteristics of each philosopher. This includes their id and state. Then the variables needed are declared. A volatile sig_atomic_t is declared to exit the loop seen in the cycle function in Figure 4.

```c
typedef enum {                          //possible states the philosophers may exhibit
        THINKING,
        EATING,
        HUNGRY
} State_t;

typedef struct {                        //struct to define attributes of each philosopher
        int id;
        State_t state;
} Person_t;

pthread_mutex_t forks[NUM];             //create array of forks of type mutex
Person_t philosophers_details[NUM];     //Array to hold the details of each philosopher
volatile sig_atomic_t terminate = 0;    //Atomic Flag for proper termination
Queue_t fork_queue[NUM];
```

Figure 2: Enum def, sturct def & parameter declarations

The threads and philosopher's details are all initialized in the first for loop of the main function seen in Figure 3. Their state is also initialised to thinking. The second loop is used to ensure that all the threads are joined at the end and that the mutex is destroyed.

```c
int main() {
        pthread_t philosopher_threads[NUM];                     //create a thread for each philosopher

        signal(SIGINT, handle_cancel);                          //signal to ensure  that the program terminates properly

        for (int i = 0; i < NUM; i++) {
                pthread_mutex_init(&forks[i], NULL);            //initiate each thread
                philosophers_details[i].id = i;                 //allocate to specific profile/details
                philosophers_details[i].state = THINKING;       //initiate their state to thinking
                init_queue(&fork_queue[i]);                     //initilize  queue for each fork
                pthread_create(&philosopher_threads[i], NULL, cycle, &philosophers_details[i]); //thread creation
        }

        for (int i = 0; i < NUM; i++) {
                pthread_join(philosopher_threads[i], NULL);     //Ensures the threads are released
                pthread_mutex_destroy(&forks[i]);               //Release all mutexes
        }
    return 0;
}
```

Figure 3: Main Program

1. Avoiding Memory Leak

The cycle function simulates the behavior of the philosophers over time. The cycle function also unlocks the mutex of both forks after the signal interrupt (SIGINT) is invoked to ensure that all the resources are unlocked before terminating the process.

```
void* cycle(void* arg) {                            //A cycle to simulate the behavior of the philosophers over time
        Person_t* philosopher = (Person_t*)arg;     //casting the philophers to type person after being cast void
        while (!terminate) {                        // The loop will end when CTRL+C is pressed
                thinking();
                take_fork(*philosopher);
                eating();
                place_fork(*philosopher);
        }
        pthread_mutex_unlock(&forks[philosopher->id]);  //When the program ends each philospher should unlock forks/resources
        pthread_mutex_unlock(&forks[(philosopher->id+1)%NUM]);
}
```

Figure 4: Cycle to Simulate Philosophers' Behavior

The sample execution in Figure 5 shows how after invoking Ctrl+C the program waits until all Philosophers stop eating to terminate the program. This ensures that all resources are released.

```
Philosopher 2 has placed down both forks!
Philosopher 2 is thinking!
Philosopher 0 is putting down forks!
                -> Philosopher 4 has picked up both forks after waiting!
Philosopher 4 is eating!
              -> Philosopher 1 has picked up both forks after waiting!
Philosopher 1 is eating!
Philosopher 0 has placed down both forks!
Philosopher 0 is thinking!
Philosopher 4 is putting down forks!
Philosopher 4 has placed down both forks!
Philosopher 4 is thinking!
              -> Philosopher 3 has picked up both forks after waiting!
Philosopher 3 is eating!
        -> Philosopher 2 is hungry!
Philosopher 3 is putting down forks!
Philosopher 3 has placed down both forks!
Philosopher 3 is thinking!
^C      -> Philosopher 4 is hungry!
                -> Philosopher 4 has picked up both forks after waiting!
Philosopher 4 is eating!
        -> Philosopher 0 is hungry!
        -> Philosopher 3 is hungry!
Philosopher 1 is putting down forks!
Philosopher 1 has placed down both forks!
Philosopher 1 is thinking!
              -> Philosopher 2 has picked up both forks after waiting!
Philosopher 2 is eating!
Philosopher 4 is putting down forks!
Philosopher 4 has placed down both forks!
Philosopher 4 is thinking!
              -> Philosopher 0 has picked up both forks after waiting!
Philosopher 0 is eating!
Philosopher 2 is putting down forks!
Philosopher 2 has placed down both forks!
Philosopher 2 is thinking!
                -> Philosopher 3 has picked up both forks after waiting!
Philosopher 3 is eating!
Philosopher 0 is putting down forks!
Philosopher 0 has placed down both forks!
Philosopher 0 is thinking!
Philosopher 3 is putting down forks!
Philosopher 3 has placed down both forks!
Philosopher 3 is thinking!
```

Commence Termination

Figure 5: Termination

Figure 6 shows the flag change when SIGINIT is invoked and the initialization is seen in the main function just before the first for loop.

```c
void handle_cancel(int signal) {
    if (signal == SIGINT) {
        terminate= 1;                                   // Set flag to stop infinite loop in case of CTRL+C
    }
}
```

Figure 6: Function to handle SIGINT

2. Avoiding Starvation

As seen below in Figure 7, starvation occurs when a thread waits too long for a resource because other threads capture them faster. In the sample output seen below Philosopher 1 becomes hungry before Philosopher 0 but in the execution of the program Philosopher 0 was able to capture it earlier. As a result Philosopher 1 had to wait for the fork to be available.

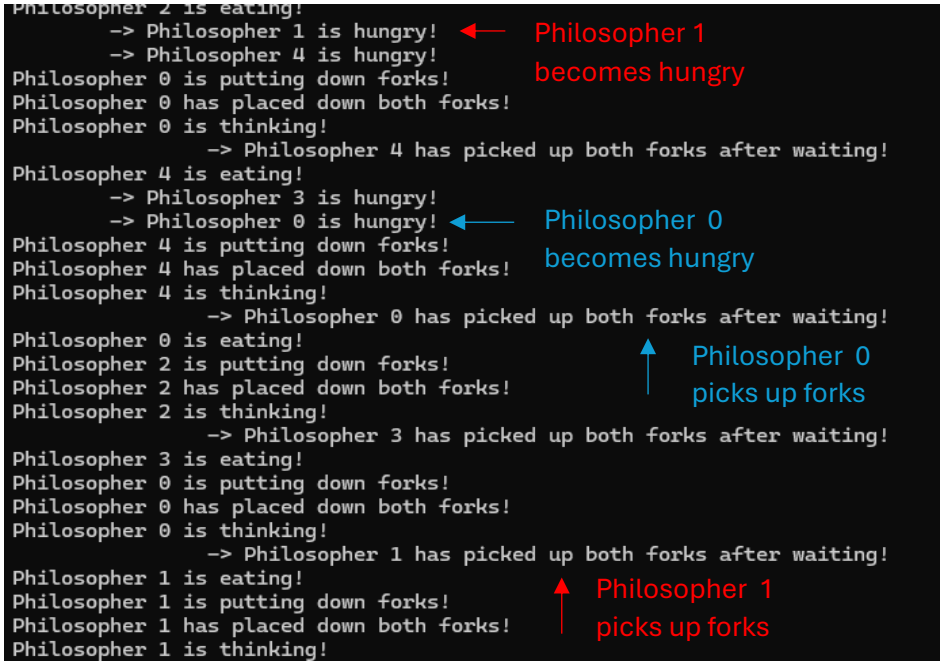

Figure 7: Example of Starvation

A special dequeue function is defined as follows in the user defined queue library. It only returns true if the head is equal to the requested ID. If this condition is not satisfied it will stay in a while loop until it is eventually satisfied. After the while loop the resources for each fork are unlocked to ensure that all threads are able to finish and to terminate to process. After adding this to the code the output became what is shown in Figure 9.

```c
int special_dequeue(Queue_t *queue, int num) {  // special equeue that returns true if the number is the head
    if (queue->data[queue->head] == num) {
        queue->data[queue->head] = -1;
        queue->head = (queue->head + 1) % QUEUE_SIZE;
        return 1;
    }

    return 0;
}
```

Figure 8: Special Dequeue Function

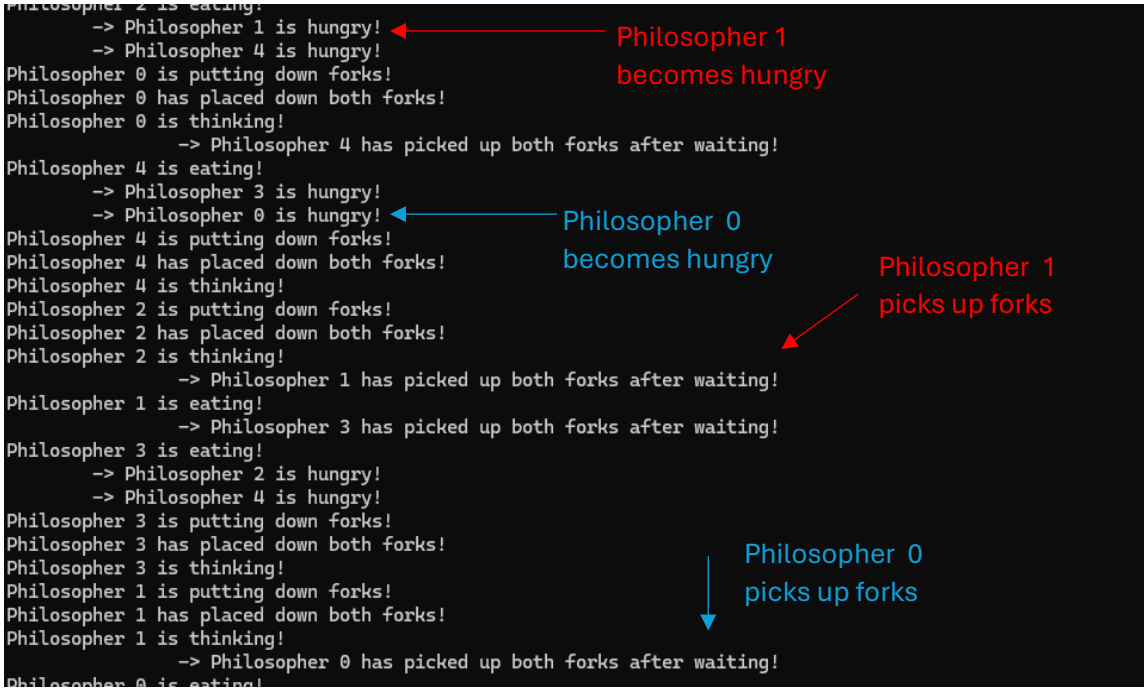

Figure 9: Output After Solving the Issue of Starvation

3. Avoiding Race

To avoid contention over a resource (fork), mutexes have been used to lock the resource on a first come first serve basis. This ensures that another thread is not able to access the same resource until the first thread releases it. The unlocking of threads is shown below in Figure 9 while the locking is shown in Figure 12.

```
void place_fork(Person_t philosopher) {
        int id = philosopher.id;
        int nxt = (id+1)%NUM;

        printf("Philosopher %d is putting down forks!\n", id);

        pthread_mutex_unlock(&forks[id]);                    //unlock resources
        pthread_mutex_unlock(&forks[nxt]);

        printf("Philosopher %d has placed down both forks!\n", id);

        philosophers_details[id].state = THINKING;

        printf("Philosopher %d is thinking!\n", id);
}
```

Figure 10: Placing the Fork Example, Unlocking mutex

4. Avoiding Deadlocks

To avoid deadlocks it was vital to ensure all philosophers do not hold a fork at no point in time. The sample output shown in figure 10 showcases this issue where the program is not able to terminate properly using Ctrl+C since all threads are stuck trying to get a locked resource.

```
        -> Philosopher 3 is hungry!
        -> Philosopher 1 is hungry!
        -> Philosopher 2 is hungry!
        -> Philosopher 0 is hungry!
        -> Philosopher 4 is hungry!
^C^C^C^C^C^C^C^C^C^C^C^C^C^C^C^C^C
```

Figure 11: Deadlock Example

This issue was avoided by letting even numbered philosophers check the fork with the same id first while odd numbers ones check the fork with the next ID. This means that every odd and even pair check the same fork first before proceeding. This ensures that all philosophers are not able to hold a fork at the same time by and create a circular wait condition.

```
void take_fork(Person_t philosopher) {
        int id = philosopher.id;                        //get the id to be used to get the right fork
        int nxt = (id+1)%NUM;

        philosopher.state = HUNGRY;
        printf("        -> Philosopher %d is hungry!\n", id);

        queue(&fork_queue[id], id);
        queue(&fork_queue[nxt], id);
        if(id%2==0){                                            //solution to cyclic deadlock; avoids having each philosopher holding a fork
                while(1){
                        if(special_dequeue(&fork_queue[id], id)){  //special dequeue only returns true if the id is at the head
                                pthread_mutex_lock(&forks[id]);
                                break;
                        }
                };

                while(1){
                        if(special_dequeue(&fork_queue[nxt], id)){
                                pthread_mutex_lock(&forks[nxt]);
                                break;
                        }
                };
        }else{
                while(1){
                        if(special_dequeue(&fork_queue[nxt], id)){  //special dequeue only returns true if the id is at the head
                                pthread_mutex_lock(&forks[nxt]);
                                break;
                        }
                };
                while(1){
                        if(special_dequeue(&fork_queue[id], id)){
                                pthread_mutex_lock(&forks[id]);
                                break;
                        }
                };
        }
        printf("                -> Philosopher %d has picked up both forks after waiting!\n", id);

        philosophers_details[id].state = EATING;

        printf("Philosopher %d is eating!\n", id);
}
```

Figure 12: take_fork Function

Conclusion:

Common multi-threading problems have been explored with a focus on memory leaks, starvation, race conditions and deadlocks. These issues have been explored through the problem of dinning philosophers. The appropriate solutions have been given as seen in the code. Below is a sample execution of the final working program.

```
        -> Philosopher 3 is hungry!
                -> Philosopher 3 has picked up both forks after waiting!
Philosopher 3 is eating!
        -> Philosopher 1 is hungry!
                -> Philosopher 1 has picked up both forks after waiting!
Philosopher 1 is eating!
Philosopher 3 is putting down forks!
Philosopher 3 has placed down both forks!
Philosopher 3 is thinking!
        -> Philosopher 2 is hungry!
        -> Philosopher 0 is hungry!
        -> Philosopher 4 is hungry!
Philosopher 1 is putting down forks!
                -> Philosopher 0 has picked up both forks after waiting!
Philosopher 0 is eating!
                -> Philosopher 2 has picked up both forks after waiting!
Philosopher 2 is eating!
Philosopher 1 has placed down both forks!
Philosopher 1 is thinking!
        -> Philosopher 3 is hungry!
Philosopher 2 is putting down forks!
Philosopher 2 has placed down both forks!
Philosopher 2 is thinking!
        -> Philosopher 1 is hungry!
Philosopher 0 is putting down forks!
Philosopher 0 has placed down both forks!
Philosopher 0 is thinking!
                -> Philosopher 4 has picked up both forks after waiting!
Philosopher 4 is eating!
                -> Philosopher 1 has picked up both forks after waiting!
Philosopher 1 is eating!
        -> Philosopher 2 is hungry!
        -> Philosopher 0 is hungry!
Philosopher 1 is putting down forks!
Philosopher 1 has placed down both forks!
Philosopher 1 is thinking!
Philosopher 4 is putting down forks!
Philosopher 4 has placed down both forks!
                -> Philosopher 0 has picked up both forks after waiting!
                -> Philosopher 3 has picked up both forks after waiting!
Philosopher 4 is thinking!
Philosopher 0 is eating!
Philosopher 3 is eating!
        -> Philosopher 1 is hungry!
Philosopher 0 is putting down forks!
Philosopher 0 has placed down both forks!
Philosopher 0 is thinking!
^C      -> Philosopher 4 is hungry!
```

```
^C      -> Philosopher 4 is hungry!
Philosopher 3 is putting down forks!
Philosopher 3 has placed down both forks!
Philosopher 3 is thinking!
                -> Philosopher 4 has picked up both forks after waiting!
Philosopher 4 is eating!
        -> Philosopher 0 is hungry!
                -> Philosopher 2 has picked up both forks after waiting!
Philosopher 2 is eating!
Philosopher 4 is putting down forks!
Philosopher 4 has placed down both forks!
Philosopher 4 is thinking!
Philosopher 2 is putting down forks!
Philosopher 2 has placed down both forks!
Philosopher 2 is thinking!
                -> Philosopher 1 has picked up both forks after waiting!
Philosopher 1 is eating!
Philosopher 1 is putting down forks!
Philosopher 1 has placed down both forks!
Philosopher 1 is thinking!
                -> Philosopher 0 has picked up both forks after waiting!
Philosopher 0 is eating!
Philosopher 0 is putting down forks!
Philosopher 0 has placed down both forks!
Philosopher 0 is thinking!
```