# Project 3 - Dining Philosophers Problem

Authors: Gabriella Garcia, Xiana Lara, Antonio Maldonado, Phillip Powell

## Abstract

This Project discusses the Dining Philosophers Problem. The problem is, there are x amount of philosophers and x-1 amount of chopsticks. This problem shows the natural problems that occur in resource allocation when there are fewer resources than processes that require the resource. Our program shows philosophers thinking, hungry, picking up chopsticks, and eating. We semaphores in our project to protect critical sections during resource allocation. To help us allocate our resources we have methods for eating, taking a pair of chopsticks, putting the chopsticks down, and the philosophers.

## Intro

The Dining Philosophers problem was the fifth project proposed to us by Dr. Wang. The problem here is that there are x-amount of philosophers and x-1 amount of chopsticks. To solve this we needed to figure out how to allocate resources that are less than the needed amount. We must look out for deadlocks while allocating our program, because it could cause our program to crash. We have learned that delays are very useful in preventing deadlocks, and that semaphores are great for protecting critical sections while allocating, dellocating and consuming resources.

## Methodology

Our experiment utilizes five global variables, which include the three statuses of each philosopher: Eating, Thinking, and Hungry. We used a semaphore, Mutex, to protect the critical section and a semaphore array to control the status of the philosophers. Four functions are created for eating, taking the left and right chopsticks, putting the chopsticks down, the philosophers, and lastly our main program. The philosopher function will be used with pthreads.

The “eating” function checks if the status of x-philosopher is starving and both chopsticks on both sides of the philosopher are not being used. After this condition check, the function will change the state of x-philosopher to eating, and sleep for 2 seconds. A print statement to the terminal will display which philosopher has taken a fork and has begun eating. Then the semaphore that is responsible for the philosopher will be unlocked.

The “takeChopsticks” function uses a semaphore to protect the critical section by utilizing a mutex. First we call sem_wait on the mutex, which is a lock for the semaphore. Then we check to see if the state of x-philosopher is starving using the “eat” function above, passing the n-philosopher’s index to the function. Afterwards we close the critical section and lock the semaphore array.

The “placeChopstick” function uses the mutex semaphore and checks to see if the philosopher’s state is “thinking”. If so, the philosopher will place their chopsticks down and it will print to the terminal that n-philosopher is thinking.

The “philosopher” function is simple but depends on each method prior. It is used exclusively for each thread. It involves setting a pointer equal to the pointer value passed, and

sending that to both the “takeChopsticks” and “placeChopsticks” functions. There is a sleep between each to ensure there’s no overlap with the threading process.

The “main” function initializes the semaphores used and their respective threads. Using a for loop, the threads are created based on the x-amount of philosophers.

## Results

*Program Code:*

```
// Authors: Gabriella Garcia, Xiana Lara, Antonio Maldonado, Phillip Powell
// Date: November 22, 2020
// CS 474 Project 3: Dining Philosophers Problem

// Define NUM of philosophers, Left side, and Right side.
#define NUM 5
#define LEFT (pNum + 4)%NUM
#define RIGHT (pNum+1)%NUM

// Libraries
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>

// Define our eat function in the global space.
void eating(int pNum);

// Define the two semaphores we will be using:
// Mutex, and a semaphore array for the philosophers.
sem_t mutex;
sem_t S[NUM];

// Define global variables, including
// the three statuses of our philosophers:
// Thinking, Hungry, and Ate.
int thinking = 0, hungry = 1, ate = 2;
int status[NUM];
int philos_num[NUM] = {0, 1, 2, 3, 4};
```

```
/*****************************************************************
 *  The following methods define eating, taking chopsticks,
 *  putting chopsticks, and the philosophers themselves.
 *****************************************************************/

// Eating Method:
// This method checks to see if the philosopher's status is hungry, and
then
// checks to see if the philosophers to the left and right of them do not
// have the ate status. Then we change the status to ate, and print to the
// terminal which philosopher took the chopsticks, and which philosopher is
eating.
void eating(int pNum){

    if(status[pNum] == hungry && status[LEFT] != ate && status[RIGHT] !=
ate){
        status[pNum] = ate;
        sleep(2);
        printf("Philosopher %d takes Chopstick %d and %d and begins to
eat.\n", pNum + 1, LEFT + 1, pNum + 1);
        sem_post(&S[pNum]);
    } // end if
} // end eat

void takeChopstick(int pNum) {
    sem_wait(&mutex);
    status[pNum] = hungry;
    printf("Philosopher number %d is Hungry\n", pNum + 1);
    eating(pNum);
    sem_post(&mutex);
    sem_wait(&S[pNum]);
    sleep(1);
} // end function

// Frees up resources for the other processes to function
void placeChopstick(int pNum){
```

```
    sem_wait(&mutex);
    status[pNum] = thinking;
    printf("Philosopher %d putting Chopstick %d and %d down and begins to
think.\n", pNum + 1, LEFT + 1, pNum + 1);
    eating(LEFT);
    eating(RIGHT);
    sem_post(&mutex);
} // end function

// Represents the philosophers picking up and putting down chopsticks
//  Used in a pthread.
void* philosopher(void *num){
    while(1) {
        int* i = num;
        sleep(1);
        takeChopstick(*i);
        placeChopstick(*i);
    } // end while
} // end function

int main(){
    int i;
    pthread_t thread_id[NUM];

    // Initializes the mutex semaphore
    sem_init(&mutex, 0, 1);

    // Initializes the chopsticks semaphores
    for(i = 0; i < NUM; i++){
        sem_init(&S[i], 0, 0);
    } // end for

    // Creates threads for each philosopher
    for(i = 0; i < NUM; i++){
        pthread_create(&thread_id[i], NULL, philosopher, &philos_num[i]);
        printf("Philosopher %d is Thinking\n", i+1);
```

```
    } // end for

    for(i = 0; i < NUM; i++) {
        pthread_join(thread_id[i], NULL);
    } // end for
} // end main
```

*Output:*

```
newton cs474/project3> gcc dining.c -lpthread
newton cs474/project3> ./a.out
Philosopher 1 is Thinking
Philosopher 2 is Thinking
Philosopher 3 is Thinking
Philosopher 4 is Thinking
Philosopher 5 is Thinking
Philosopher number 2 is Hungry
Philosopher 2 takes Chopstick 1 and 2 and begins to eat.
Philosopher number 1 is Hungry
Philosopher number 4 is Hungry
Philosopher 4 takes Chopstick 3 and 4 and begins to eat.
Philosopher number 3 is Hungry
Philosopher number 5 is Hungry
Philosopher 2 putting Chopstick 1 and 2 down and begins to think.
Philosopher 1 takes Chopstick 5 and 1 and begins to eat.
Philosopher 4 putting Chopstick 3 and 4 down and begins to think.
Philosopher 3 takes Chopstick 2 and 3 and begins to eat.
Philosopher 1 putting Chopstick 5 and 1 down and begins to think.
Philosopher 5 takes Chopstick 4 and 5 and begins to eat.
Philosopher number 2 is Hungry
Philosopher number 4 is Hungry
Philosopher 3 putting Chopstick 2 and 3 down and begins to think.
Philosopher 2 takes Chopstick 1 and 2 and begins to eat.
Philosopher number 1 is Hungry
Philosopher 5 putting Chopstick 4 and 5 down and begins to think.
Philosopher 4 takes Chopstick 3 and 4 and begins to eat.
Philosopher number 3 is Hungry
Philosopher 2 putting Chopstick 1 and 2 down and begins to think.
Philosopher 1 takes Chopstick 5 and 1 and begins to eat.
Philosopher number 5 is Hungry
Philosopher 4 putting Chopstick 3 and 4 down and begins to think.
Philosopher 3 takes Chopstick 2 and 3 and begins to eat.
Philosopher number 2 is Hungry
Philosopher 1 putting Chopstick 5 and 1 down and begins to think.
Philosopher 5 takes Chopstick 4 and 5 and begins to eat.
Philosopher 3 putting Chopstick 2 and 3 down and begins to think.
Philosopher 2 takes Chopstick 1 and 2 and begins to eat.
Philosopher number 4 is Hungry
Philosopher number 1 is Hungry
Philosopher 5 putting Chopstick 4 and 5 down and begins to think.
```

```
Philosopher 4 takes Chopstick 3 and 4 and begins to eat.
Philosopher number 3 is Hungry
Philosopher 2 putting Chopstick 1 and 2 down and begins to think.
Philosopher 1 takes Chopstick 5 and 1 and begins to eat.
Philosopher number 5 is Hungry
Philosopher 4 putting Chopstick 3 and 4 down and begins to think.
Philosopher 3 takes Chopstick 2 and 3 and begins to eat.
Philosopher 1 putting Chopstick 5 and 1 down and begins to think.
Philosopher 5 takes Chopstick 4 and 5 and begins to eat.
Philosopher number 2 is Hungry
Philosopher number 4 is Hungry
Philosopher 3 putting Chopstick 2 and 3 down and begins to think.
Philosopher 2 takes Chopstick 1 and 2 and begins to eat.
Philosopher number 1 is Hungry
Philosopher 5 putting Chopstick 4 and 5 down and begins to think.
Philosopher 4 takes Chopstick 3 and 4 and begins to eat.
Philosopher number 3 is Hungry
Philosopher 2 putting Chopstick 1 and 2 down and begins to think.
Philosopher 1 takes Chopstick 5 and 1 and begins to eat.
Philosopher number 5 is Hungry
Philosopher 4 putting Chopstick 3 and 4 down and begins to think.
Philosopher 3 takes Chopstick 2 and 3 and begins to eat.
Philosopher number 2 is Hungry
Philosopher 1 putting Chopstick 5 and 1 down and begins to think.
^C
newton cs474/project3>
```

## Conclusions

The Dining Philosophers Problem showcases the limitations and issues with resource allocation when there is less resources than processes. To create a solution for this problem we used a function to delay the time between processes. The delay allows the philosopher to finish eating and place down the chopsticks so that others can use it. Using this delay prevents cases of deadlock which would eventually crash a system. Semaphores were used to protect the critical sections that involve resource allocation, deallocation, and consumption. In conclusion, our team has learned the importance of deadlock prevention, because deadlocks can potentially crash a system or prevent processes from functioning, which sometimes can be a critical process needed.