Anggota Kelompok:
- Christianto Vinsen B. (00000028917)
- Delvin Chianardi (00000028583)

Dining Philosophers Problem

Formulasi

Permasalahan **Dining Philosophers** menyatakan bahwa ada sebanyak **2 sampai 100** Philosopher duduk di meja bundar dengan sumpit berjumlah Philosopher yang ada. Setiap sumpit berada di antara philosopher - philosopher. Philosopher dapat makan jika memiliki kedua sumpit dan satu sumpit hanya dapat diambil oleh satu Philosopher tetapi tidak kedua philosopher yang berdampingan. Makanan yang dimakan oleh Philosopher adalah **Bakpao** yang merupakan resource yang dapat diubah secara dinamis saat runtime, dengan jumlah **1 sampai 100**. Bakpao akan berkurang 1 jika dimakan oleh Philosopher, dan jika Bakpao sudah habis maka Philosopher tidak akan dapat makan lagi dan kegiatan Dining Philosopher akan berhenti.

Pemaparan dari permasalahan Dining Philosophers dan sistem yang ada :

1. Input jumlah Philosopher (P) dinamis saat runtime, dimana $2 \leq P \leq 100$. Nilai default untuk jumlah Philosopher adalah 5.
2. Bakpao adalah makanan atau Resources yang akan dimakan oleh Philosopher ketika Philosopher memenuhi persyaratan untuk makan. Program akan berhenti ketika semua Bakpao telah habis dimakan Philosopher.
3. Input jumlah Bakpao (B) dinamis saat runtime, dimana $1 \leq B \leq 100$ dan Bakpao merupakan makanan atau Resources yang akan dimakan oleh Philosopher ketika Philosopher memenuhi persyaratan untuk makan. Nilai default untuk jumlah Bakpao adalah 10.
4. Aksi yang dapat dilakukan oleh masing-masing philosopher:
    a. Makan Bakpao
    Philosopher makan Bakpao ketika sudah mendapatkan kedua sumpit dan Bakpao masih tersedia.
    b. Berpikir
    Philosopher berpikir ketika sudah kenyang makan Bakpao.
    c. Mengambil Sumpit
    Philosopher mencari kedua sumpit sebelum makan Bakpao.
5. Kondisi yang dapat terjadi pada masing-masing philosopher:
    a. Kenyang
    Kondisi setelah philosopher selesai makan Bakpao.
    b. Lapar
    Kondisi setelah philosopher berpikir terlalu keras dan ingin makan Bakpao.
6. Aturan Dining Philosopher
    a. Setiap Philosopher diberi nama Philosopher 1 sampai ke N, dan setiap kali Philosopher makan akan terhitung (Increment 1 setiap makan).

b. Philosopher untuk makan harus mempunyai kedua sumpit (kanan dan kiri) secara eksklusif dan setiap Philosopher hanya dapat mengambil sumpit yang tepat berada di kiri dan kanannya saja.
c. Philosopher hanya bisa dan pasti melakukan satu aksi setiap saat.
d. Setiap Philosopher mempunyai waktu berpikir dan makan yang berbeda-beda setiap saatnya. Dimana waktu makan antara 1s - 5s, dan waktu berpikir bergantung pada kecepatan setiap komputer dalam mengeksekusi bubble sort untuk mengurutkan setiap angka pada array yang berukuran 30.000 sampai 50.000 dengan kelipatan 2.500. Sehingga urutan makan dan berpikir akan berbeda setiap program berjalan.
e. Philosopher akan selalu mengambil sumpit kiri terlebih dahulu, kemudian mengambil sumpit kanan jika tersedia untuk makan. Jika sumpit kanan tidak tersedia dalam waktu tertentu, maka Philosophers akan meletakan kembali sumpit kiri dan tidak makan.
f. Philosopher akan meletakan kedua sumpit setelah selesai makan dan kemudian akan berpikir lagi.
g. Setiap Philosopher makan, jumlah Bakpao akan dikurangi 1, dan simulasi program akan berhenti ketika Bakpao habis (Bakpao = 0) dan akan menampilkan hasil statistik dari simulasi program Dining Philosopher.

Solusi Masalah Dining Philosopher:

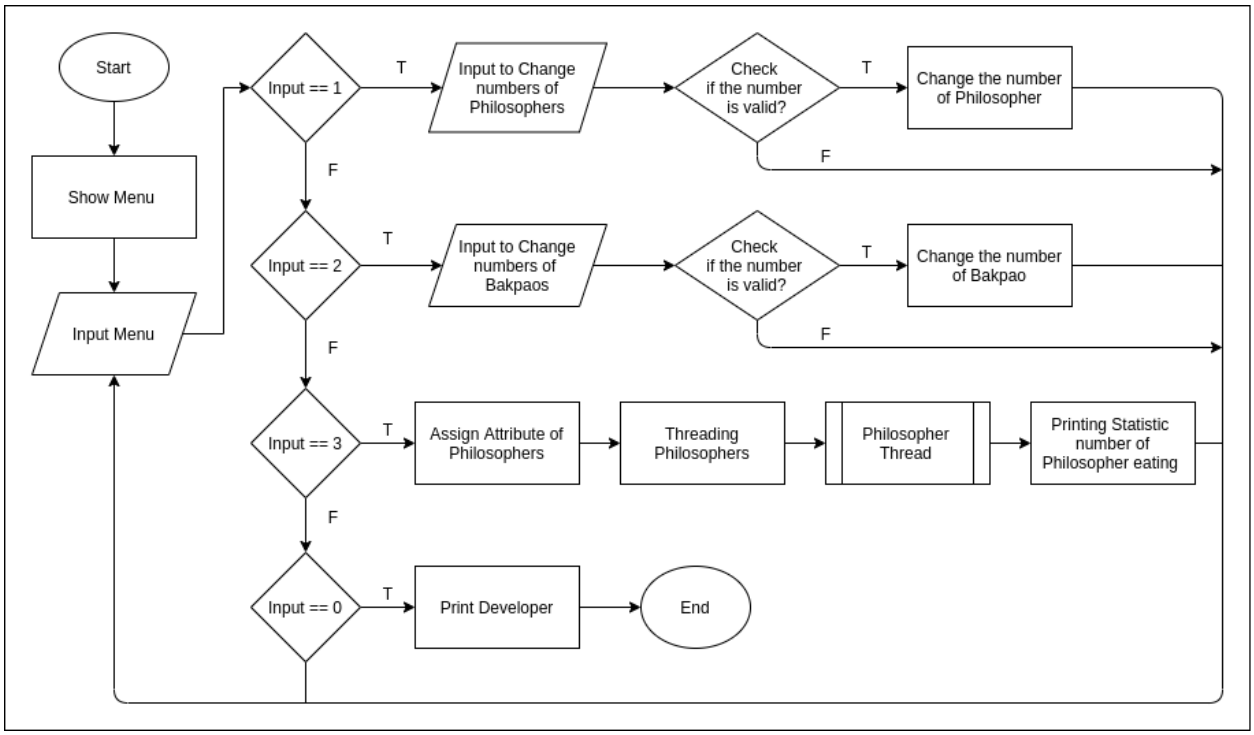
- Setiap Philosopher mempunyai masing-masing **leftChopstickIndex** dan **rightChopstickIndex** yang bertujuan untuk membatasi sumpit yang dapat diambil oleh setiap Philosopher.
- Setiap Sumpit diberikan masing-masing **mutex** dengan tujuan menghindari sumpit dipakai oleh lebih dari 1 philosopher pada waktu yang bersamaan.
    - Ketika akan mengambil sumpit dari meja, digunakan **sem_trywait** pada mutex untuk mengecek apakah sumpit sedang digunakan atau tidak oleh Philosopher lain. Jika sedang digunakan, maka Philosopher tidak dapat mengambil sumpit tersebut, tetapi jika sumpit tersedia, maka Philosopher akan melakukan **lock** pada sumpit tersebut yang menandakan bahwa sumpit tersebut masih digunakan dan Philosopher lain tidak dapat menggunakannya.
    - Ketika akan mengembalikan sumpit ke meja, digunakan **sem_post** pada mutex untuk melakukan **unlock** pada sumpit tersebut yang menandakan bahwa sumpit tersebut sudah tidak digunakan lagi dan Philosopher lain dapat menggunakan sumpit tersebut.
- Setiap Philosopher yang ingin makan akan selalu mengambil sumpit sebelah kiri terlebih dahulu, jika sukses mengambil sumpit kiri, maka akan ada **waiting time** untuk mengambil sumpit sebelah kanannya. Jika waiting time habis dan Philosopher tidak berhasil mengambil sumpit kanannya, maka sumpit kiri yang sudah diambilnya akan dikembalikan ke meja dan Philosopher tidak jadi makan dan kembali berpikir. Sebaliknya, jika Philosopher berhasil mengambil sumpit kanannya ketika waiting time masih ada, maka Philosopher akan makan. Setelah Philosopher selesai makan, maka kedua sumpit akan dikembalikan ke meja.

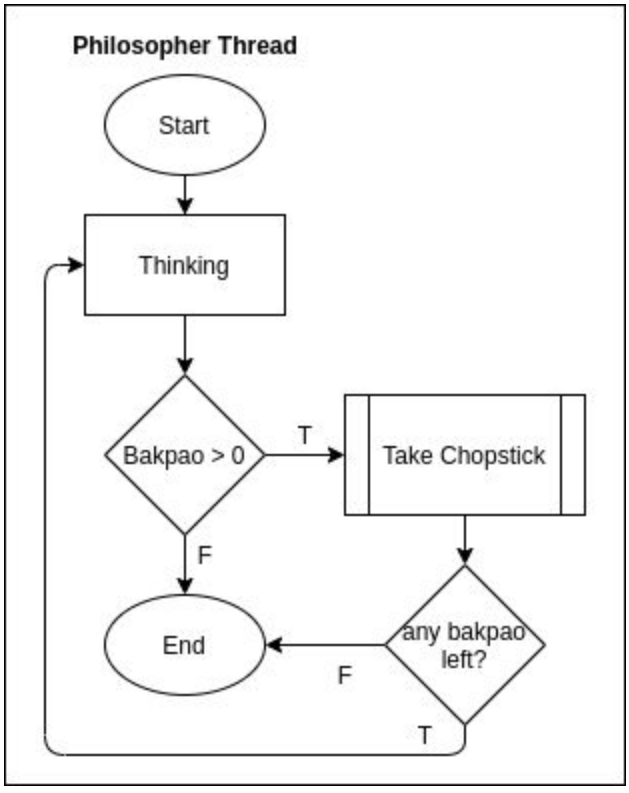
Contoh Ekspektasi:

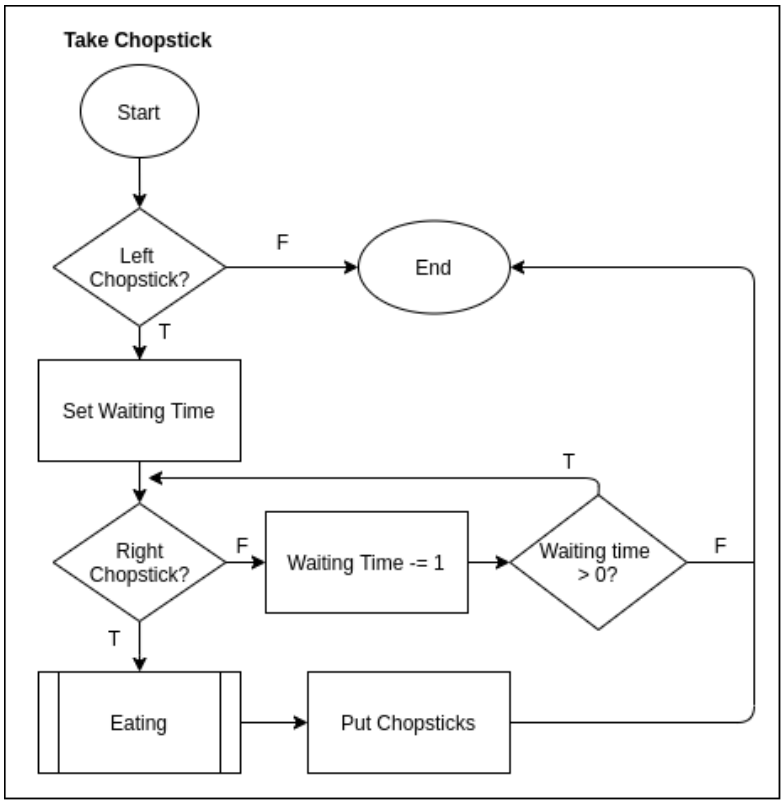
- **Input:**
  - Philosopher = 3
  - Bakpao = 3
- **Output:**

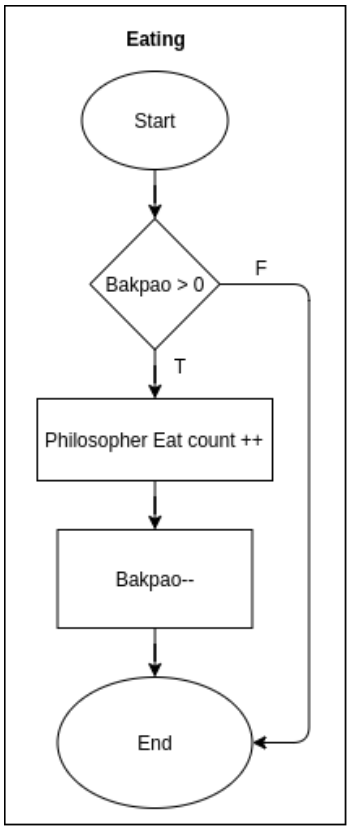
  Bakpao = 3
  Philosopher A Berpikir
  Philosopher B Berpikir
  Philosopher C Berpikir
  Philosopher A lapar dan ingin makan
  Philosopher A mulai makan Bakpao
  Bakpao = 2
  Philosopher A selesai makan Bakpao
  Philosopher B lapar dan ingin makan
  Philosopher C lapar dan ingin makan
  Philosopher B mulai makan Bakpao
  Bakpao = 1
  Philosopher B selesai makan Bakpao
  Philosopher C mulai makan Bakpao
  Philosopher A lapar dan ingin makan
  Bakpao = 0
  Philosopher C selesai makan Bakpao
  Karena Bakpao habis, maka Philosopher A tidak jadi makan dan Proses simulasi selesai

Flowchart


Start
Show Menu
Input Menu
Input == 1
T
Input to Change numbers of Philosophers
Check if the number is valid?
T
Change the number of Philosopher
F
F
Input == 2
T
Input to Change numbers of Bakpaos
Check if the number is valid?
T
Change the number of Bakpao
F
F
Input == 3
T
Assign Attribute of Philosophers
Threading Philosophers
Philosopher Thread
Printing Statistic number of Philosopher eating
F
Input == 0
T
Print Developer
End


Philosopher Thread
Start
Thinking
Bakpao > 0
T
Take Chopstick
F
End
any bakpao left?
F
T


Take Chopstick
Start
Left Chopstick?
F
End
T
Set Waiting Time
T
Right Chopstick?
F
Waiting Time -= 1
Waiting time > 0?
F
T
Eating
Put Chopsticks


Eating
Start
Bakpao > 0
F
T
Philosopher Eat count ++
Bakpao--
End

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>

struct Philosopher {
   int number;
   int eatenTimes;
   int leftChopstickIndex;
   int rightChopstickIndex;
   pthread_t threadID;
};

struct Chopstick {
   int index;
   sem_t mutex;
};

int is_finished();
void think(struct Philosopher *philosopher);
void eat(struct Philosopher *philosopher);
void put_chopsticks(struct Philosopher *philosopher, char chopstickPosition[]);
void take_chopsticks(struct Philosopher *philosopher);

void actionTime();

struct Chopstick* chopsticks;
sem_t globalMutex;
int currentFoods = 0;

void* philosopher_thread(void *argument) {
   struct Philosopher* philosopher = (struct Philosopher*)argument;

   while(1) {
       // Philosopher start thinking
       think(philosopher);

       // Philosopher wants to eat
       printf("Philosopher %d is hungry and wants to eat...\n", philosopher->number);

       // Check if Bakpao is exists
       if (currentFoods > 0) {
           // Philosopher take the chopsticks
           take_chopsticks(philosopher);
       } else {
           // Bakpao doesn't exists
```

```
            printf("Philosopher %d failed to Eat because the Bakpao ran out\n",
philosopher->number);
        }

        // Checks whether all Philosophers have finished eating
        if (is_finished()) break;
    }
}

void main() {
    struct Philosopher* philosophers;
    int optionChosen, i, numberOfPhilosophers = 5, numberOfPhilosophersRequested, numberOfFoods
= 10, numberOfFoodsRequested;
    srand((unsigned int)time(NULL));

    while(1) {
        printf("================================================================================\n");
        printf("          Welcome to DINING PHILOSOPHERS Simulation Program          \n");
        printf("================================================================================\n");
        printf("Current number of Philosopher = %d\n", numberOfPhilosophers);
        printf("Current number of Food = %d\n", numberOfFoods);
        printf("1. Change the number of Philosophers\n");
        printf("2. Change the number of Bakpao\n");
        printf("3. Start Simulation\n");
        printf("0. Exit\n");
        printf("Your choice: ");
        scanf("%d", &optionChosen);

        if (optionChosen == 1) {
            // Change the number of Philosophers
            printf("\nPlease enter the number of Philosophers (2 ~ 100) : ");
            scanf("%d", &numberOfPhilosophersRequested);

            // Check if the new number of Philosopher is valid
            if (numberOfPhilosophersRequested >= 2 && numberOfPhilosophersRequested <= 100) {
                numberOfPhilosophers = numberOfPhilosophersRequested;
                            printf("\nNumber of Philosophers has been changed to %d\n\n",
numberOfPhilosophers);
            } else {
                           printf("\nFailed to change the number of Philosophers to %d\n",
numberOfPhilosophersRequested);
                printf("The number of Philosophers allowed is between 2 - 100.\n\n");
            }
        } else if (optionChosen == 2) {
            // Change the number of Bakpao
            printf("\nPlease enter the number of Bakpao (1 ~ 100) : ");
            scanf("%d", &numberOfFoodsRequested);

            // Check if the new number of Bakpao is valid
            if (numberOfFoodsRequested >= 1 && numberOfFoodsRequested <= 100) {
                numberOfFoods = numberOfFoodsRequested;
                printf("\nNumber of Bakpao has been changed to %d\n\n", numberOfFoods);
```

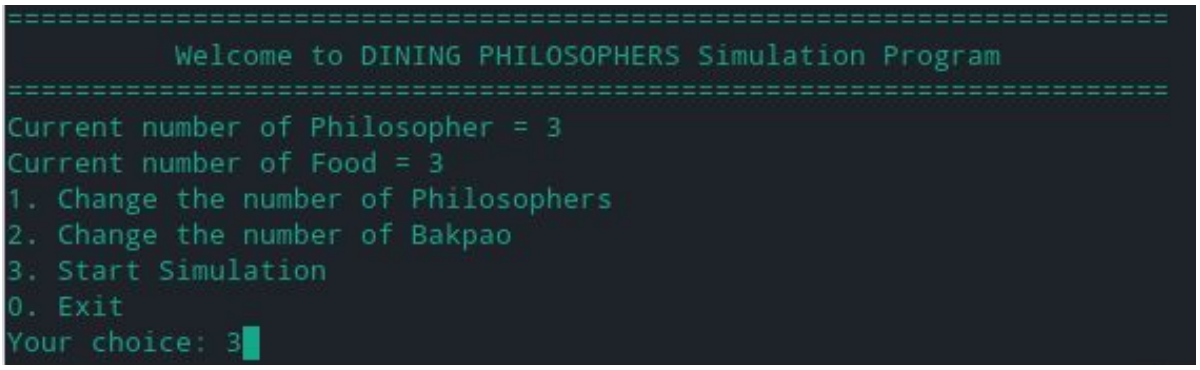
```
            } else {
                                printf("\nFailed to change the number of Bakpao to %d\n",
numberOfFoodsRequested);
                printf("The number of Bakpao allowed is between 1 - 100.\n\n");
            }
        } else if (optionChosen == 3) {
            // Simulation Dining Philosopher Program
            printf("================================================================\n");
            printf("                          SIMULATION BEGIN                      \n");
            printf("================================================================\n");

            // Create array of struct for philosophers
                    philosophers = (struct Philosopher*) malloc(sizeof(struct Philosopher) *
numberOfPhilosophers);

            // Create array of struct for chopsticks
                        chopsticks = (struct Chopstick*) malloc(sizeof(struct Chopstick) *
numberOfPhilosophers);

            // Initialize global mutex in order to protect the critical region
            sem_init(&globalMutex, 0, 1);

              // Set number of foods to current foods, current foods value will always decrease
every time the Philosopher eats
            currentFoods = numberOfFoods;
              printf("====================== AVAILABLE BAKPAO : %d ======================\n",
currentFoods);

            for(i = 0; i < numberOfPhilosophers; i++) {
                // Initialize mutex of chopstick
                sem_init(&chopsticks[i].mutex, 0, 1);

                // Set number to Philosopher (Naming Philosopher)
                philosophers[i].number = i + 1;
                // Set number of Philosopher eating to zero
                philosophers[i].eatenTimes = 0;
                // Set index of left chopstick
                philosophers[i].leftChopstickIndex = i;
                // Set index of right chopstick
                if ((i + 1) == numberOfPhilosophers)
                    philosophers[i].rightChopstickIndex = 0;
                else
                    philosophers[i].rightChopstickIndex = i+1;
            }

            // Create thread for each Philosopher
            for(i = 0; i < numberOfPhilosophers; i++)
                            pthread_create(&philosophers[i].threadID, NULL, philosopher_thread,
&philosophers[i]);

            // Wait until all threads are finished
            for(i = 0; i < numberOfPhilosophers; i++)
```

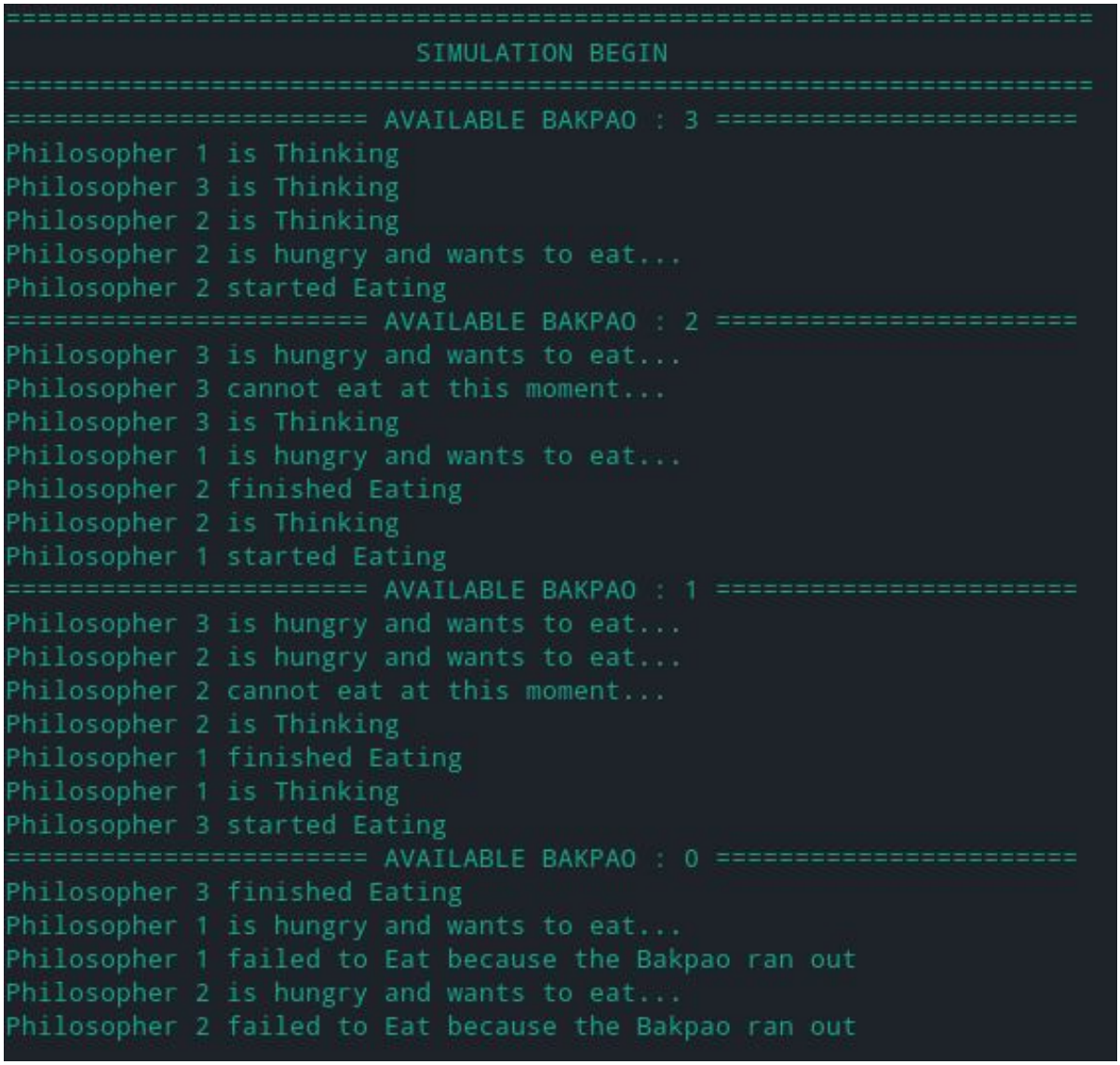
```
            pthread_join(philosophers[i].threadID, NULL);

        // Prints Philosopher Statistics

printf("\n=========================================================================\n");
        printf("                      PHILOSOPHERS STATISTICS                      \n");
        printf("=========================================================================\n");
        for(i = 0; i < numberOfPhilosophers; i++)
                printf("Philosopher %d eaten for %d times\n", philosophers[i].number,
philosophers[i].eatenTimes);

        // Free all dynamically allocated memory
        free(chopsticks);
        free(philosophers);


printf("\n=========================================================================\n");
        printf("                          SIMULATION ENDS                          \n");

printf("=========================================================================\n\n");
      } else if (optionChosen == 0) {
        printf("\nThank you for using this program\n");
        printf(" 1. Christianto Vinsen B. - 00000028917\n");
        printf(" 2. Delvin Chianardi      - 00000028583\n");
        break;
      } else {
        printf("The option you entered is wrong. Please try again\n\n");
      }
   }
}

int is_finished() {
   // Use global mutex to enter the critical region
   sem_wait(&globalMutex);
   int temp = currentFoods;
   sem_post(&globalMutex);

   // return 1 (true) if current foods is empty or zero
   return temp == 0;
}

void think(struct Philosopher *philosopher) {
   // Philosopher start thinking
   printf("Philosopher %d is Thinking\n", philosopher->number);

   // Do something for delay thinking
   actionTime();
}

void eat(struct Philosopher *philosopher) {
   // Philosopher start eating
   printf("Philosopher %d started Eating\n", philosopher->number);
```

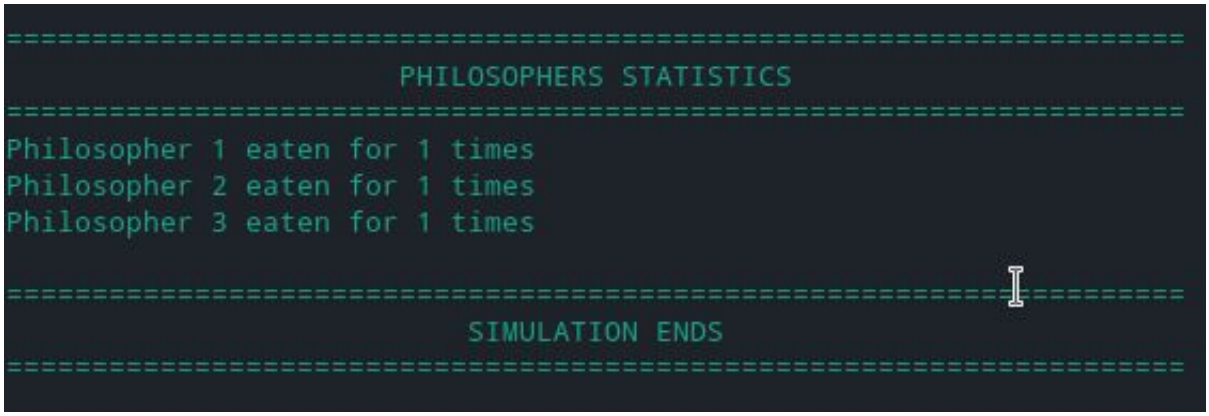
```
    // Use global mutex to enter the critical region
    sem_wait(&globalMutex);
    // Check if Bakpao is exists
    if (currentFoods > 0) {
        // Reduce the number of Bakpao by one
        currentFoods--;
            printf("======================== AVAILABLE BAKPAO : %d ======================\n",
currentFoods);
        // Increase the number of eaten times of Philosopher by one
        philosopher->eatenTimes++;
    } else {
        // Bakpao doesn't exists
                printf("Philosopher %d failed to Eat because the Bakpao ran out\n",
philosopher->number);
    }
    sem_post(&globalMutex);

    // Delay time for the Philosopher to eat. Random Delay from 1.00 to 5.00 second
    usleep(1000000 + (100000 * (rand() % 41)));
}

void put_chopsticks(struct Philosopher *philosopher, char chopstickPosition[]) {
    // Check which chopsticks to put back on the table
    if (strcmp(chopstickPosition, "left") == 0)
        // Put the left chopstick back on the table
        sem_post(&chopsticks[philosopher->leftChopstickIndex].mutex);
    else if (strcmp(chopstickPosition, "right") == 0)
        // Put the right chopstick back on the table
        sem_post(&chopsticks[philosopher->rightChopstickIndex].mutex);
}

void take_chopsticks(struct Philosopher *philosopher) {
     // Try to get left chopstick, here use sem_trywait instead of sem_wait because it makes
possible to resolve deadlock
    if (sem_trywait(&chopsticks[philosopher->leftChopstickIndex].mutex) == 0) {
         // Generate random waiting time for take the right chopstick. Random waiting time from
10 to 59 interval
        int waitingTimes = 10 + rand() % 50;

        // Check whether the waiting time to take the right chopstick is still available
        while(waitingTimes > 0) {
            // Try to get right chopstick
            if (sem_trywait(&chopsticks[philosopher->rightChopstickIndex].mutex)==0) {
                // Philosopher has both chopsticks and is ready to eat
                eat(philosopher);

                // After eating, the two chopsticks were put back on the table
                put_chopsticks(philosopher, "right");
                put_chopsticks(philosopher, "left");
```

```
                // Philosopher finished eating and also finished returning the chopsticks to
the table
                printf("Philosopher %d finished Eating\n", philosopher->number);

                break;
            } else {
                // Philosopher have not been able to take the right chopstick because chopstick
are still used by other Philosophers
                // Reduce timer for waiting right chopstick by one
                waitingTimes--;
                // Delay for 0.1 second
                usleep(100000);
            }
        }

        // If the waiting time is 0, it means Philosopher cannot get the right chopstick and
the waiting time is up
        // So the Philosopher will return the left chopstick to the table even though he is
hungry
        if (waitingTimes == 0) {
            printf("Philosopher %d cannot take second chopstick...\n", philosopher->number);
            // Put the left chopstick on the table
            put_chopsticks(philosopher, "left");
        }
    } else {
        // Philosopher have not been able to take the left chopstick because chopstick are
still used by other Philosophers
        printf("Philosopher %d cannot eat at this moment...\n", philosopher->number);
    }
}

void actionTime() {
    int arraySize;
    // Generate random array size from 30.000 to 50.000 with multiples of 2500
    arraySize = 30000 + (2500 * (rand() % 9));

    int arr[arraySize];
    int tempArraySize = arraySize;
    // Initialize array before, ex: random size = 3, then the array value = 3, 2, 1
    for (int i = 0; i < arraySize; i++) {
        arr[i] = tempArraySize--;
    }
    // Doing Bubble Sort, ex: array size = 3, Before = 3, 2, 1 --> After = 1, 2, 3
    int n = sizeof(arr)/sizeof(arr[0]);
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

Contoh Hasil:






PHILOSOPHERS STATISTICS
Philosopher 1 eaten for 1 times
Philosopher 2 eaten for 1 times
Philosopher 3 eaten for 1 times
SIMULATION ENDS