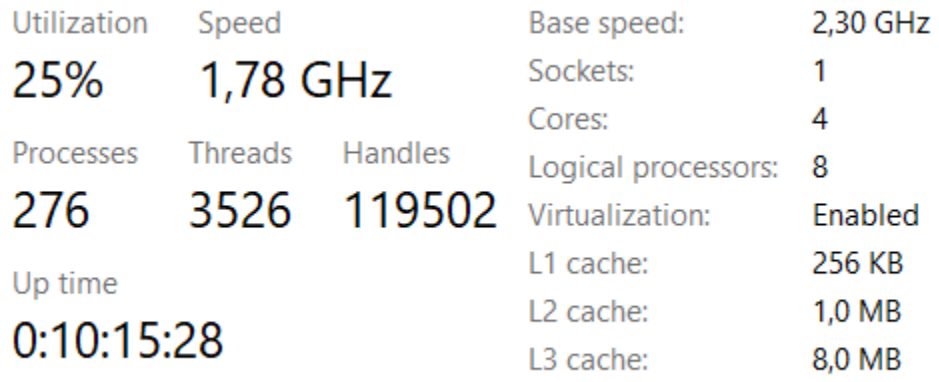
Project Programming 1

1. Evan Christopher (202000187)
2. Stefannus Christian (202000138)
3. Jessica Ong (202000204)

Sum Array Reciprocal



Program *Sum Array Reciprocal* dijalankan pada laptop dengan 8 *Logical Processors*.

Cara Menjalankan Program

```
PS E:\VS_Code\SOP\ProjectProgramming1> python arrayreciprocal.py --thread 1 --array 1000000
Mau Berapa Iterasi? 10
```

Program akan meminta *input user* berupa berapa iterasi yang diinginkan oleh *user*. Misalnya pada contoh diatas, *user* meminta 10x iterasi, maka program akan menjalankan 1 eksekusi dengan *random array* sebesar 1.000.000 element sebanyak 10x dan dari 10x iterasi tersebut, program akan membandingkan performa *Sum Array Reciprocal* dengan 1 *thread* dan dengan 0 *thread* (*conventional sum*). Kami tidak memakai fungsi *built-in sum* dari *python* karena menurut riset kami, fungsi *built-in sum* dari *python* sudah dioptimasi, sehingga kami membuat fungsi sendiri untuk menjumlahkan elemen-elemen pada *array.*

Penjelasan Code

Import Library yang dibutuhkan.
Library Threading untuk menjalankan *parallel programming.*
Library Time untuk menghitung waktu eksekusi program. Method yang digunakan adalah *perf_counter()*.
Library argparse untuk input dari cmd berupa *--thread* dan *--arraysize*.
Library numpy untuk mengubah *array* menjadi *reciprocal array*. *Reciprocal Array* adalah sebuah vektor atau array yang melibatkan penambahan kebalikan dari semua elemen array. Kebalikan nilai dari elemen tersebut dapat dihitung dengan 1/elemen. *Method* yang digunakan adalah numpy.reciprocal. Kami menggunakan numpy.reciprocal karena numpy.reciprocal sudah

dioptimasi python sehingga dengan menggunakan *method* ini, program akan mengubah *array* menjadi *reciprocal array* dengan waktu yang lebih cepat.
From numpy.random *import uniform* adalah untuk menghasilkan *random float elements* yang nantinya akan diubah menjadi *reciprocal array* untuk perhitungan.

```
1   import threading
2   import time
3   import argparse
4   import numpy as np
5   from numpy.random import uniform
```

Kode dibawah adalah fungsi untuk menghasilkan *random array* menggunakan *method uniform* yang disediakan oleh *numpy*. *Output* dari kode di bawah adalah menghasilkan *array* dengan panjang *size* berisikan elemen acak yang bervariasi antara float 1.0 sampai 101.0.

```
def generate_random_arr(size):
    return uniform(1.0, 101.0, size)
```

Kode dibawah adalah fungsi untuk menjumlahkan seluruh elemen dari array yang sebelumnya sudah diubah dahulu ke dalam *reciprocal array*. Seperti yang sudah dijelaskan diatas, kami membuat fungsi sendiri untuk sum karena *built-in* sum python sudah dioptimisasi.

```
def conventional_sum(arr):
    res = 0
    for num in arr:
        res += num
    return res
```

Kode dibawah adalah fungsi pembulatan untuk membulatkan hasil menjadi 4 desimal di belakang koma.

```
def pembulatan(n):
    return round(n, 4)
```

Kode dibawah adalah fungsi untuk meminta *input* dari CMD. *Thread* dan *array size* harus di convert dulu menjadi integer karena jika *thread* dan *array size* di input dari CMD, python akan menganggap tipe data *thread* dan *array size* tersebut sebagai string sehingga kedua variabel string ini harus diubah menjadi integer sebelum dikembalikan oleh fungsi.

```
def parser():
    import_cmd = argparse.ArgumentParser()
    import_cmd.add_argument('--thread')
    import_cmd.add_argument('--array')
    args = import_cmd.parse_args()
    thread = int(args.thread)
    arraysize = int(args.array)
    return thread, arraysize
```

Kode dibawah adalah fungsi untuk menjalankan *parallel programming*. Untuk menjalankan *parallel programming,* dibutuhkan fungsi yang akan dipanggil, oleh karena itu *thread sum* adalah fungsi yang akan digunakan untuk *threading*. Fungsi di bawah akan menjumlah array yang dipotong sesuai dengan jumlah *thread*-nya. Misalkan, jika *thread*-nya 8 maka *array* akan dibagi menjadi 8 bagian yang berbeda dan 8 *array* tersebut akan dijumlahkan secara bersamaan. Contoh lainnya, jika *array* tersebut memiliki besar 1000000, maka *array* tersebut akan dibagi menjadi 8 *array* berukuran 125000.

```
def thread_sum(array, start, end):
    result = conventional_sum(array[start:end])
    result_list.append(result)
```

Kode dibawah adalah fungsi *make thread* yang akan membuat *thread* sebanyak *thread* yang dimasukkan oleh *user* pada CMD. *Thread* tersebut akan disimpan pada *thread list* kemudian *thread* tersebut akan dijalankan bersama-sama kemudian digabungkan. thread.join() adalah fungsi yang menyebabkan *main thread* menunggu *thread* selesai. Jika tidak menggunakan thread.join(), maka *thread* akan berjalan dengan sendirinya.

```python
def make_thread(array, size, n):
    thread_list = []
    for i in range(n):
        th = threading.Thread(target=thread_sum, args=(
            array, i*(size//n), (i+1)*(size//n)))
        thread_list.append(th)

    for thread in thread_list:
        thread.start()

    for thread in thread_list:
        thread.join()
```

Penjelasan Fungsi Main

- Membuat variabel iterasi untuk menghitung berapa kali iterasi yang dilakukan
- Cout *with* adalah untuk melacak apakah program berjalan lebih cepat ketika dijalankan menggunakan N thread
- Cout *without* adalah untuk melacak apakah program berjalan lebih cepat ketika dijalankan tanpa thread
- Penjelasan di dalam *for loop*
    - Untuk mengetahui berapa kali iterasi, juga menambah variabel iterasi dengan 1 untuk setiap loop
    - Kemudian kita mengambil *input* dari CMD berupa berapa *thread* dan *array size* yang diinginkan
    - Catat waktu untuk membuat *array* sebesar *user input* untuk membuat *array* sebesar *input* dari CMD kemudian *array* tersebut akan diubah menggunakan fungsi yang sudah kami jelaskan diatas yaitu fungsi untuk mengubah *array* menjadi *reciprocal array*.
    - Setelah itu *print* waktu yang dibutuhkan untuk membuat *array* dan memberi keterangan *size* dari *array* yang dibuat

```
def main(number_of_iterations):
    iterasi = 0
    cout_with = 0
    cout_without = 0
    for _ in range(number_of_iterations):
        iterasi += 1
        n, size = parser()

        t_awal_generate_random_array = time.perf_counter()
        arr_not_res = generate_random_arr(size)
        arr = convert_array_to_reciprocal(arr_not_res)
        t_end_generate_random_array = time.perf_counter()
        t_generate_random_array = pembulatan(
            t_end_generate_random_array - t_awal_generate_random_array)

        print(f'Menggenerate array {size} processing dan reciprocal.')
        print(f'Waktu pengerjaan {t_generate_random_array} detik.')
```

- Kode di bawah ini membuat *thread* dan hasilnya disimpan ke dalam *result list,* dan dijumlahkan secara konvensional.
- *Print* keterangan bahwa program sedang menghitung *sum* dari *array reciprocal processing* beserta waktu pengerjaannya dengan n *thread*

```
t_thread_start = time.perf_counter()
make_thread(arr, size, n)
res_sum = pembulatan(conventional_sum(result_list))
t_thread_end = time.perf_counter()
t_total = pembulatan(t_thread_end - t_thread_start)

print(f'Menghitung sum dari array reciprocal processing.')
print(f'Waktu pengerjaan dengan {n} thread = {t_total} detik.')
```

- Setelah itu program akan mencatat waktu yang diperlukan untuk *conventional sum*.
- Program akan mencetak hasil jumlah dengan *thread* dengan hasil jumlah tanpa thread untuk membandingkan apakah hasil nya sudah sama atau tidak. Fungsi ini hanya untuk mengecek saja.

```
t_start_py_sum = time.perf_counter()
py_sum = pembulatan(conventional_sum(arr))
t_end_py_sum = time.perf_counter()
t_py_sum = t_end_py_sum - t_start_py_sum

print(
    f'Waktu pengerjaan tanpa thread: {pembulatan(t_py_sum)} detik.')

print(f'Sum Result With Thread: {res_sum}')
print(f'Sum Result Without Thread: {py_sum}')
```

- Kode dibawah ini membandingkan waktu yang diperlukan program untuk menjalankan *sum array reciprocal* dengan *thread* dan tanpa menggunakan *thread* dan akan mencatat hasilnya. Selain itu program juga akan mencatat beda waktu yang dibutuhkan.
- Jika *conventional sum* lebih cepat, maka variabel cout *without* akan ditambah 1 dan jika *sum* dengan *thread* lebih cepat maka variabel cout *with* akan ditambah 1 kemudian program akan mencetak siapa yang lebih cepat beserta dengan perbedaan waktunya.

```
if t_py_sum < t_total:
    beda = t_total - t_py_sum
    cout_without += 1
    print(f'Without Thread Lebih Cepat {pembulatan(beda)} detik!\n')
else:
    beda = t_py_sum - t_total
    cout_with += 1
    print(f'With Thread Lebih Cepat {pembulatan(beda)} detik!\n')
```

- Ketika program membuat *thread*, program akan membagi *thread* tersebut berdasarkan *input* dari CMD dan akan menyimpan hasil dari *thread* tersebut ke dalam *result list*. *Result list* merupakan *global list* sehingga *result list* ini perlu dikosongkan sehingga untuk iterasi berikutnya, *result list* sudah kosong. Kemudian program akan memprint *with thread* lebih unggul berapa kali dari n iterasi dan *without thread* lebih unggul berapa kali dari n iterasi

```
    result_list.clear()
print(
    f'Dari {iterasi} iterasi. With Thread Lebih Cepat {cout_with}x, Lebih Lama {cout_without}x.')
```

- Kode dibawah ini membuat result list untuk menyimpan hasil dari jumlah dari *thread array reciprocal*. Kemudian akan meminta *input user* untuk berapa kali iterasi. *Try except*

dan *while loop* digunakan untuk mencegah adanya salah *input* dari *user* sehingga *input user* dijamin akan berupa *integer* yang valid. Kemudian program *main* dijalankan sebanyak iterasi yang user inginkan.

```
if __name__ == '__main__':
    result_list = []
    while True:
        try:
            n = int(input('Mau Berapa Iterasi? '))
            break
        except ValueError:
            print('Invalid Input!\n')
    print()
    main(n)
```

Evaluasi Project (semua dijalankan 10x dan hanya satu yang ditampilkan dari kesepuluh hasil)

1. 1 eksekusi dengan *random array sebesar* 1.000.000 elemen

```
Menggenerate array 1000000 processing dan reciprocal.
Waktu pengerjaan 0.0248 detik.
Menghitung sum dari array reciprocal processing.
Waktu pengerjaan dengan 1 thread = 0.2059 detik.
Waktu pengerjaan tanpa thread: 0.2155 detik.
Sum Result With Thread: 46280.5699
Sum Result Without Thread: 46280.5699
With Thread Lebih Cepat 0.0096 detik!
```

```
Dari 10 iterasi. With Thread Lebih Cepat 9x, Lebih Lama 1x.
```

2. 1 eksekusi dengan *random array* sebesar 100.000.000 elemen

```
Menggenerate array 100000000 processing dan reciprocal.
Waktu pengerjaan 2.2441 detik.
Menghitung sum dari array reciprocal processing.
Waktu pengerjaan dengan 1 thread = 21.4088 detik.
Waktu pengerjaan tanpa thread: 24.4718 detik.
Sum Result With Thread: 4615481.3956
Sum Result Without Thread: 4615481.3956
With Thread Lebih Cepat 3.063 detik!
```

```
Dari 10 iterasi. With Thread Lebih Cepat 7x, Lebih Lama 3x.
```

3. 8 eksekusi dengan *random array* sebesar 1.000.000 elemen

```
Menggenerate array 1000000 processing dan reciprocal.
Waktu pengerjaan 0.023 detik.
Menghitung sum dari array reciprocal processing.
Waktu pengerjaan dengan 8 thread = 0.206 detik.
Waktu pengerjaan tanpa thread: 0.2137 detik.
Sum Result With Thread: 46283.0074
Sum Result Without Thread: 46283.0074
With Thread Lebih Cepat 0.0077 detik!
```

```
Dari 10 iterasi. With Thread Lebih Cepat 6x, Lebih Lama 4x.
```

4. 8 eksekusi dengan *random array* sebesar 100.000.000 element

```
Menggenerate array 100000000 processing dan reciprocal.
Waktu pengerjaan 2.3395 detik.
Menghitung sum dari array reciprocal processing.
Waktu pengerjaan dengan 8 thread = 21.539 detik.
Waktu pengerjaan tanpa thread: 23.3045 detik.
Sum Result With Thread: 4614355.5168
Sum Result Without Thread: 4614355.5168
With Thread Lebih Cepat 1.7655 detik!
```

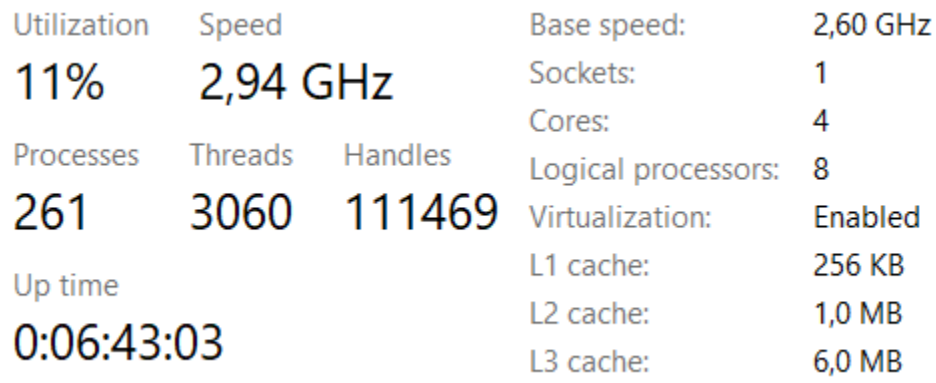
```
Dari 10 iterasi. With Thread Lebih Cepat 6x, Lebih Lama 4x.
```

Dari beberapa *screenshot* di atas, dapat terlihat bahwa semakin banyak elemen maka dengan *threading* akan menjadi lebih cepat dibandingkan dengan tanpa menggunakan *threading*. Namun jika dibandingkan dalam jumlah iterasi dengan *threading* dan tanpa *threading*, satu

eksekusi dengan satu juta elemen dan delapan eksekusi dengan satu juta elemen hanya memiliki perbedaan 0,0001 detik.
Jika dibandingkan antara 1 eksekusi 100 juta elemen dengan 8 eksekusi 100 juta elemen, keduanya memiliki perbedaan sebesar 0,1302 detik.
Kedua hal ini dapat disebabkan oleh beberapa faktor, seperti ketidakmampuan kami dalam *parallel programming*, prosesor yang kami gunakan untuk proses iterasi, maupun sistem *built-in python* yang tidak mengizinkan penjumlahan dengan sekaligus.

Keseluruhan evaluasi ini dapat menunjukkan bahwa dengan *threading* maka iterasi akan dijalankan lebih cepat, dengan jumlah iterasi menggunakan *threading* yang lebih dominan dalam hal kecepatan jika dibandingkan tanpa *threading*.

Dining Philosophers Problem



Program Dining Philosopher dijalankan pada laptop dengan 8 logical processors

Cara Menjalankan Program

```
PS C:\Users\Intel\Downloads\Telegram Desktop> python diningphilosopher.py --filsuf 5 --makan 1
```

Program akan dijalankan melalui terminal IDE atau *command prompt* dari *device* masing-masing dengan format “python (nama file) --filsuf (jumlah filsuf yang diinginkan) --makan (berapa kali setiap filsuf makan)

Penjelasan Code
Import library yang dibutuhkan
Library threading untuk menjalankan *parallel programing*
Library time untuk menghitung waktu eksekusi program. Method yang digunakan adalah sleep().
Library argparse untuk *input* dari CMD berupa --filsuf dan --makan.

```
import threading
import time
import argparse
```

Pertama, kami membuat kelas *Philosopher* yang *inherit* dari kelas threading.Thread
Terdapat variabel is_all_eat untuk mengecek apakah semua filsuf sudah selesai makan atau belum.

```
class Philosopher(threading.Thread):


    is_all_eat = True
```

Sebelum melakukan hal lain, karena *subclass* ini *override constructor*, maka *subclass* ini harus *invoke base class constructor* dengan threading.Thread.__init__(self).
Terdapat 4 *constructor*, yaitu index, forkOnLeft, forkOnRight, dan capacity.
Index berfungsi untuk "menandai" filsuf, dari filsuf 1, filsuf 2, .. filsuf n.
forkOnLeft berfungsi untuk "menandai" garpu yang berada di kiri filsuf tertentu, dan *forkOnRight* untuk garpu yang berada di kanan filsuf tertentu.
Capacity berfungsi untuk menandai berapa kali filsuf tertentu akan makan.

```
def __init__(self, index, forkOnLeft, forkOnRight, capacity):
    threading.Thread.__init__(self)
    self.index = index
    self.forkOnLeft = forkOnLeft
    self.forkOnRight = forkOnRight
    self.capacity = capacity
```

Fungsi run() untuk menjalankan program dan mencetak siapa saja yang dalam keadaan lapar.
Dengan fungsi *while*, kapasitas akan berkurang 1 selama kapasitas tersebut masih belum mencapai nilai 0. Jika telah mencapai nilai 0, maka *loop* ini akan diputuskan.
Jika *loop* ini telah terputuskan, maka filsuf tertentu akan mencapai keadaan tertidur selama beberapa detik tertentu yang didukung oleh fungsi time.sleep(3) (dalam hal ini, 3 detik), di mana pada kasus program *single thread,* sleep() menangguhkan eksekusi *thread* dan *proses*. Namun, fungsi ini akan menangguhkan utas daripada seluruh proses dalam program *multithreading*.
Lalu program akan mencetak "Philosopher (ke-berapa) is hungry.", dan filsuf tersebut akan mulai makan (dengan fungsi self.eat()) jika kedua semafor (garpu) bebas, di mana garpu kiri

akan menjalankan operasi *wait* (menunggu), dan jika garpu kanan tidak tersedia maka akan meninggalkan garpu kiri. Setelah melakukan proses “makan”, kedua garpu akan dilepaskan.

```
def run(self):
    while (self.is_all_eat):
        if self.capacity != 0:
            self.capacity -= 1
        else:
            break

        # Philosopher is thinking (but really is sleeping).
        time.sleep(3)
        print('Philosopher %s is hungry.' % self.index)
        self.eat()
```

Fungsi eat() akan menunjukkan proses makan dari para filsuf, di mana akan didefinisikan di awal bahwa garpu pertama adalah *forkOnLeft* yang berada di constructor awal, dan garpu kedua adalah *forkOnRight* agar lebih memudahkan dalam melakukan pemrograman.
Ketika filsuf sedang dalam keadaan makan, maka filsuf akan “mengambil” garpu pertama dengan fungsi acquire() yang merupakan fungsi built in dari kelas *Lock* dari modul *threading* dalam *Python*. Metode ini digunakan untuk memperoleh kunci, baik memblokir atau *non-blocking*.
Jika metode ini dipanggil tanpa argumen, maka metode itu akan memblokir utas panggilan hingga kunci dibuka oleh utas yang menggunakannya saat ini.
Secara singkat, jika keadaan sedang terkunci, maka *method* acquire() ini akan memblokir hingga *method* release() dipanggil pada thread lainnya, sehingga method release() ini akan merubah keadaan terkunci menjadi terbuka. Method release() ini hanya bisa dipanggil jika keadaan sedang terkunci.

```
def eat(self):
    fork1, fork2 = self.forkOnLeft, self.forkOnRight

    while self.is_all_eat:
        fork1.acquire()
        locked = fork2.acquire(False)
        if locked:
            break
        fork1.release()
    else:
        return

    self.dining()

    fork2.release()
    fork1.release()
```

Fungsi dining() ini hanya digunakan untuk mencetak filsuf yang mana saja yang mulai dan selesai makan.

```
def dining(self):
    print('Philosopher %s starts eating. ' % self.index)
    time.sleep(3)
    print('Philosopher %s finishes eating and leaves to think.' % self.index)
```

Fungsi main() akan pertama kali menginisialisasi *array semaphore*, yaitu garpu-garpu yang akan digunakan oleh para filsuf.
(i+1)%filsuf akan digunakan untuk mendapatkan garpu kanan dan kiri secara sirkuler antara 1-N jumlah filsuf.
Semaphore sendiri merupakan sebuah “sistem” atau “alat” untuk memberikan pesan dengan menentukan posisi tetap berdasarkan suatu kode tertentu.
Thread akan masuk ke dalam *semaphore* dengan memanggil *method* WaitOne, yaitu *method* yang diturunkan dari kelas WaitHandle, kemudian melepaskan *semaphore* dengan memanggil *method release*. Setiap kali sebuah *thread* masuk ke dalam *semaphore*, maka hitungan dalam sebuah *semaphore* akan berkurang, dan akan bertambah jika sebuah *thread* melepaskan

semaphore.

```
def main(filsuf, makan):
    forks = [threading.Semaphore() for n in range(filsuf)]

    philosophers = [Philosopher(i, forks[i % filsuf], forks[(i + 1) % filsuf], makan)
                    for i in range(filsuf)]

    Philosopher.is_all_eat = True
    for p in philosophers:
        p.start()
    time.sleep(100)
    Philosopher.is_all_eat = False
    print("Now we're finishing.")
```

Fungsi ini untuk meminta *input* dari CMD.

```
def parser():
    parser = argparse.ArgumentParser()
    parser.add_argument("--filsuf")
    parser.add_argument("--makan")
    args = parser.parse_args()
    filsuf = int(args.filsuf)
    makan = int(args.makan)
    return filsuf, makan
```

Kode ini untuk menjalankan program menggunakan fungsi main dengan parameter banyak filsuf dan berapa kali makan yang telah diambil dari CMD melalui fungsi parser.

```
if __name__ == "__main__":
    filsuf, makan = parser()
    main(filsuf, makan)
```

1. Evaluasi program menggunakan 5 filsuf dan 1 makan

```
PS C:\Users\Intel\Downloads\Telegram Desktop> python diningphilosopher.py --filsuf 5 --makan 1
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 3 starts eating.
Philosopher 2 is hungry.
Philosopher 0 is hungry.
Philosopher 1 is hungry.
Philosopher 0 starts eating.
Philosopher 3 finishes eating and leaves to think.
Philosopher 2 starts eating.
Philosopher 0 finishes eating and leaves to think.
Philosopher 4 starts eating.
Philosopher 2 finishes eating and leaves to think.
Philosopher 1 starts eating.
Philosopher 4 finishes eating and leaves to think.
Philosopher 1 finishes eating and leaves to think.
Now we're finishing.
```

2. Evaluasi program menggunakan 10 filsuf dan 2 makan

```
PS C:\Users\Intel\Downloads\Telegram Desktop> python diningphilosopher.py --filsuf 10 --makan 2
Philosopher 7 is hungry.
Philosopher 7 starts eating.
Philosopher 2 is hungry.
Philosopher 2 starts eating.
Philosopher 4 is hungry.
Philosopher 4 starts eating.
Philosopher 8 is hungry.
Philosopher 5 is hungry.
Philosopher 6 is hungry.
Philosopher 0 is hungry.
Philosopher 1 is hungry.
Philosopher 9 is hungry.
Philosopher 3 is hungry.
Philosopher 0 starts eating.
Philosopher 2 finishes eating and leaves to think.
Philosopher 7 finishes eating and leaves to think.
Philosopher 6 starts eating.
Philosopher 4 finishes eating and leaves to think.
Philosopher 8 starts eating.
Philosopher 0 finishes eating and leaves to think.
Philosopher 3 starts eating.
Philosopher 1 starts eating.
Philosopher 2 is hungry.
Philosopher 7 is hungry.
Philosopher 6 finishes eating and leaves to think.
Philosopher 5 starts eating.
```

```
Philosopher 4 is hungry.
Philosopher 0 is hungry.
Philosopher 8 finishes eating and leaves to think.
Philosopher 7 starts eating.
Philosopher 1 finishes eating and leaves to think.
Philosopher 3 finishes eating and leaves to think.
Philosopher 9 starts eating.
Philosopher 2 starts eating.
Philosopher 5 finishes eating and leaves to think.
Philosopher 6 is hungry.
Philosopher 4 starts eating.
Philosopher 8 is hungry.
Philosopher 1 is hungry.
Philosopher 9 finishes eating and leaves to think.
Philosopher 3 is hungry.
Philosopher 7 finishes eating and leaves to think.
Philosopher 2 finishes eating and leaves to think.
Philosopher 6 starts eating.
Philosopher 8 starts eating.
Philosopher 1 starts eating.
Philosopher 4 finishes eating and leaves to think.
Philosopher 5 is hungry.
Philosopher 3 starts eating.
Philosopher 9 is hungry.
Philosopher 6 finishes eating and leaves to think.
Philosopher 8 finishes eating and leaves to think.
Philosopher 1 finishes eating and leaves to think.
```

```
Philosopher 5 starts eating.
Philosopher 9 starts eating.
Philosopher 3 finishes eating and leaves to think.
Philosopher 9 finishes eating and leaves to think.
Philosopher 5 finishes eating and leaves to think.
Philosopher 0 starts eating.
Philosopher 0 finishes eating and leaves to think.
Now we're finishing.
```

Dari beberapa *screenshot* di atas, dapat terlihat bahwa program ini dapat menyelesaikan *dining philosopher problem* ini dengan baik dan sesuai dengan contoh yang diberikan.