

UNIVERSITAS MUHAMMADIYAH PONTIANAK
FAKULTAS TEKNIK DAN ILMU KOMPUTER
PROGRAM STUDI TEKNIK INFORMATIKA

LAPORAN PRAKTIKUM
PERTEMUAN 6

Menyelesaikan Deadlock dengan Lock dan Semaphore



Mata Kuliah : Komputasi Paralel dan Terdistribusi
Dosen : Yulrio Brianorman, S.Si., M.T.
Kelas : 27

Disusun Oleh:

Name : Rivaldi

NIM : 231220056

TAHUN 2025

CONTENT

CONTENT.....	ii
1 Pendahuluan.....	1
1.1 Lata Belakang.....	1
1.2 Tujuan.....	1
1.3 Alat dan Bahan	1
2 Landasan Teori.....	2
2.1 Thread.....	2
2.2 Lock (Mutex).....	2
2.3 Semaphore	2
2.4 Race Condition.....	2
2.5 Deadlock.....	2
2.6 Solusi Deadlock.....	2
3 Hasil Pratikum	3
3.1 Latihan 1: Thread Dasar	3
3.1.1 Kode Program.....	3
3.1.2 Output Program	4
3.1.3 Analisis	4
3.2 Latihan 2: Rice Condition	4
3.2.1 Kode Program.....	4
3.2.2 Output Program	5
3.2.3 Analisis	5
3.3 Latihan 3: Solusi dengan Lock.....	5
3.3.1 Kode Program.....	5
3.3.2 Output Program	6
3.3.3 Analisis	6
3.4 Latihan 4: Semaphore.....	6
3.4.1 Kode Program.....	6
3.4.2 Output Program	8
3.4.3 Analisis	8
3.5 Latihan 5: Deadlock	8
3.5.1 Kode Program.....	8

3.5.2 Output Program	9
3.5.3 Analisis	10
3.6 Latihan 6: Solusi Deadlock-Lock Ordering	10
3.6.1 Kode Program.....	10
3.6.2 Output Program	11
3.6.3 Analisis	11
3.7 Latihan 7: Solusi Deadlock-Timeout	11
3.7.1 Kode Program.....	11
3.7.2 Output Program	12
3.7.3 Analisis	12
3.8 Latihan 8: Solusi Deadlock-Try-Lock.....	13
3.8.1 Kode Program.....	13
3.8.2 Output	14
3.8.3 Analisis	14
4 Tugas Mandiri	14
4.1 Tugas 1: Database Connection Pool.....	14
4.1.1 Kode Program.....	14
4.1.2 Output Program	15
4.1.3 Penjelasan	15
4.2 Tugas 2: Dining Philosophers Problem.....	16
4.2.1 Kode Program.....	16

4.2.2 Output Program

```

(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/T2philosophers_problem.py"
=====
DINING PHILOSOPHERS PROBLEM
Solusi: Lock Ordering
=====
Aturan: Selalu ambil garpu dengan nomor lebih kecil terlebih dahulu
=====
Filsuf-0 telah duduk di meja
Filsuf-1 telah duduk di meja
Filsuf-2 telah duduk di meja
Filsuf-3 telah duduk di meja
Filsuf-4 telah duduk di meja

===== MULAI =====
Filsuf-0 sedang berpikir...
Filsuf-1 sedang berpikir...
Filsuf-2 sedang berpikir...
Filsuf-3 sedang berpikir...
Filsuf-4 sedang berpikir...
Filsuf-3 mencoba mengambil garpu 3
Filsuf-3 berhasil mengambil garpu 3
Filsuf-3 mencoba mengambil garpu 4
Filsuf-3 berhasil mengambil garpu 4
Filsuf-3 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-4 mencoba mengambil garpu 0
Filsuf-4 berhasil mengambil garpu 0
Filsuf-4 mencoba mengambil garpu 4
Filsuf-1 mencoba mengambil garpu 1
Filsuf-1 berhasil mengambil garpu 1
Filsuf-1 mencoba mengambil garpu 2
Filsuf-1 berhasil mengambil garpu 2
Filsuf-1 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-0 mencoba mengambil garpu 0
Filsuf-2 mencoba mengambil garpu 2
Filsuf-1 melepaskan garpu 2
Filsuf-2 berhasil mengambil garpu 2
Filsuf-1 melepaskan garpu 1
Filsuf-2 mencoba mengambil garpu 3
Filsuf-1 selesai makan ke-1

```

....19

4.2.3 Penjelasan	21
5 Kesimpulan	23
6 Saran	23
7 Lampiran	23
7.1 SS hasil Program	23
7.2 Link Repository	29

1 Pendahuluan

1.1 Lata Belakang

Dalam Komputasi Paralel dan Terdistribusi, penggunaan thread memungkinkan program untuk menjalankan beberapa tugas secara bersamaan. Namun, penggunaan thread juga menimbulkan masalah seperti race condition dan deadlock. Race condition terjadi ketika dua atau lebih thread mengakses shared data secara bersamaan tanpa sinkronisasi yang tepat. Deadlock terjadi ketika thread saling menunggu resource yang dipegang oleh thread lain.

Praktikum ini bertujuan untuk memahami konsep thread, lock, semaphore, serta cara mengatasi deadlock dengan berbagai teknik.

1.2 Tujuan

Tujuan dari praktikum ini adalah:

1. Memahami konsep Thread, Lock, dan Semaphore
2. Mengidentifikasi kondisi yang menyebabkan deadlock
3. Mengimplementasikan solusi deadlock dengan Lock Ordering
4. Mengimplementasikan solusi deadlock dengan Timeout
5. Mengimplementasikan solusi deadlock dengan Try-Lock
6. Menggunakan Semaphore untuk membatasi akses resource

1.3 Alat dan Bahan

- Laptop/PC dengan Python 3.8 atau lebih tinggi
- Text Editor / IDE (VS Code, PyCharm, dll)
- Terminal / Command Prompt

2 Landasan Teori

2.1 Thread

Thread adalah unit eksekusi terkecil dalam sebuah proses yang dapat dijadwalkan oleh sistem operasi. Thread berbagi memory space yang sama dalam satu proses, sehingga dapat mengakses variabel dan resource yang sama.

2.2 Lock (Mutex)

Lock atau Mutex (Mutual Exclusion) adalah mekanisme sinkronisasi yang memastikan hanya satu thread dapat mengakses critical section pada satu waktu. Lock digunakan untuk mencegah race condition.

2.3 Semaphore

Semaphore adalah mekanisme sinkronisasi yang membatasi jumlah thread yang dapat mengakses resource secara bersamaan. Berbeda dengan lock yang hanya mengizinkan 1 thread, semaphore dapat mengizinkan N thread.

2.4 Race Condition

Race condition adalah situasi dimana hasil eksekusi program bergantung pada timing atau urutan eksekusi thread. Hal ini terjadi ketika beberapa thread mengakses shared data tanpa sinkronisasi yang tepat.

2.5 Deadlock

Deadlock adalah situasi dimana dua atau lebih thread saling menunggu resource yang dipegang oleh thread lain, sehingga tidak ada thread yang dapat melanjutkan eksekusi.

4 Syarat Terjadinya Deadlock:

1. **Mutual Exclusion:** Resource hanya dapat dipakai oleh satu thread
2. **Hold and Wait:** Thread memegang resource sambil menunggu resource lain
3. **No Preemption:** Resource tidak dapat direbut secara paksa
4. **Circular Wait:** Thread saling menunggu membentuk lingkaran

2.6 Solusi Deadlock

Terdapat beberapa strategi untuk mengatasi deadlock:

- Lock Ordering: Memastikan semua thread mengambil lock dengan urutan yang sama
- Timeout: Memberikan batas waktu pada lock acquisition
- Try-Lock: Menggunakan non-blocking lock dengan retry mechanism
- Lock Hierarchy: Memberikan prioritas pada setiap lock

3 Hasil Pratikum

3.1 Latihan 1: Thread Dasar

3.1.1 Kode Program

```
import threading
import time

def tugas_1():
    """Thread pertama"""
    for i in range(5):
        print(f"Thread 1: {i}")
        time.sleep(0.5)

def tugas_2():
    """Thread kedua"""
    for i in range(5):
        print(f"Thread 2: {i}")
        time.sleep(0.5)

# Buat thread
t1 = threading.Thread(target=tugas_1)
t2 = threading.Thread(target=tugas_2)

# Jalankan thread
print("Memulai thread...")
t1.start()
t2.start()

# Tunggu semua thread selesai
t1.join()
t2.join()

print("Semua thread selesai!")
```

Listing 1: Thread Dasar

3.1.2 Output Program

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L1thread_dasar.py
Memulai thread...
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Semua thread selesai!
```

3.1.3 Analisis

Jawaban Pertanyaan:

1. Apa yang terjadi jika tidak menggunakan join()?

Jawaban: “Program utama tidak akan menunggu thread selesai dan langsung melanjutkan eksekusi, sehingga output “Semua thread selesai!” mungkin dicetak sebelum thread selesai.”

2. Apakah output selalu sama?

Jawaban: “Tidak selalu sama, karena eksekusi thread bergantung pada penjadwalan system operasi (scheduling).”

3.2 Latihan 2: Rice Condition

3.2.1 Kode Program

```
import threading

counter = 0

def tambah_counter():
    """Fungsi untuk menambah counter"""
    global counter
    for _ in range(100000):
        counter += 1

# Buat 2 thread
t1 = threading.Thread(target=tambah_counter)
t2 = threading.Thread(target=tambah_counter)

# Jalankan
t1.start()
```

```
t2.start()

# Tunggu selesai
t1.join()
t2.join()

print(f"Counter akhir: {counter}")
print(f"Harusnya: 200000")
print(f"Selisih: {200000 - counter}")
```

Listing 2: Race Condition

3.2.2 Output Program

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L2race_condition.py
Counter akhir: 200000
Harusnya: 200000
Selisih: 0
```

3.2.3 Analisis

Jawaban Pertanyaan:

1. Berapa nilai counter yang diharapkan?

Jawaban: "200000"

2. Mengapa nilai counter tidak akurat?

Jawaban: "Karena terjadi race condition saat beberapa thread mengubah variabel *counter* secara bersamaan tanpa sinkronisasi."

3.3 Latihan 3: Solusi dengan Lock

3.3.1 Kode Program

```
i import threading

counter = 0
lock = threading.Lock() # Buat lock

def tambah_counter_aman():
    """Fungsi aman dengan lock"""
    global counter
    for _ in range(100000):
        with lock: # Kunci critical section
            counter += 1
        # Lock otomatis dilepas di sini

# Buat 2 thread
t1 = threading.Thread(target=tambah_counter_aman)
t2 = threading.Thread(target=tambah_counter_aman)
```

```
# Jalankan
t1.start()
t2.start()

# Tunggu selesai
t1.join()
t2.join()

print(f"Counter akhir: {counter}")
print(f"Harusnya: 200000")
print(f"Selisih: {200000 - counter}")

if counter == 200000:
    print("SUKSES! Data aman dengan Lock")
```

Listing 3: Race Condition with Lock

3.3.2 Output Program

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L3lock.py
Counter akhir: 200000
Harusnya: 200000
Selisih: 0
SUKSES! Data aman dengan Lock
```

3.3.3 Analisis

Jawaban Pertanyaan:

1. Apa fungsi dari with lock?

Jawaban: “Memastikan hanya satu thread yang dapat mengakses critical section pada satu waktu.”

2. Apakah nilai counter selalu tepat 200000?

Jawaban: “Iya, karena lock mencegah race condition.”

3.4 Latihan 4: Semaphore

3.4.1 Kode Program

```
import threading
import time

# Semaphore dengan maksimal 2 thread
printer = threading.Semaphore(2)

def print_dokumen(karyawan, halaman):
    """Simulasi print dokumen"""
    print(f"{karyawan} menunggu printer...")
```

```
with printer: # Tunggu sampai ada printer tersedia
    print(f"{karyawan} mulai print {halaman} halaman")
    time.sleep(halaman * 0.5) # Simulasi waktu print
    print(f"{karyawan} selesai print")

# 5 karyawan ingin print
karyawan_list = [
    ("Ipal", 1),
    ("Ipul", 3),
    ("Yogi", 4),
    ("Yoga", 2),
    ("Udins", 1),
]

threads = []
for nama, halaman in karyawan_list:
    t = threading.Thread(target=print_dokumen, args=(nama, halaman))
    threads.append(t)
    t.start()

# Tunggu semua selesai
for t in threads:
    t.join()

print("\nSemua dokumen selesai dicetak!")
```

Listing 4: Semaphore-Printer Kantor

3.4.2 Output Program

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L4semaphore.py
Ipal menunggu printer...
Ipal mulai print 1 halaman
Ipul menunggu printer...
Yogi menunggu printer...
Ipul mulai print 3 halaman
Yoga menunggu printer...
Udins menunggu printer...
Ipal selesai print
Yogi mulai print 4 halaman
Ipul selesai print
Yoga mulai print 2 halaman
Yogi selesai print
Udins mulai print 1 halaman
Yoga selesai print
Udins selesai print

Semua dokumen selesai dicetak!
```

3.4.3 Analisis

1. Beberapa maksimal yang bisa print bersamaan?

Jawaban: "2 orang (karena `Semaphore(2)`)."

2. Apa perbedaan Semaphore dengan lock?

Jawaban: "Lock hanya mengizinkan 1 thread, sedangkan Semaphore dapat mengizinkan N thread."

3.5 Latihan 5: Deadlock

3.5.1 Kode Program

```
import threading
import time

rekening_A = threading.Lock()
rekening_B = threading.Lock()

def transfer_A_ke_B():
    """Transfer dari A ke B"""
    print("Transfer A->B: Mengunci rekening A...")
    with rekening_A:
        print("Transfer A->B: Rekening A terkunci!")
        time.sleep(0.5)
```

```

        print("Transfer A->B: Mencoba kunci rekening B...")
        with rekening_B:
            print("Transfer A->B: Berhasil!")

def transfer_B_ke_A():
    """Transfer dari B ke A"""
    print("Transfer B->A: Mengunci rekening B...")
    with rekening_B:
        print("Transfer B->A: Rekening B terkunci!")
        time.sleep(0.5)

    print("Transfer B->A: Mencoba kunci rekening A...")
    with rekening_A:
        print("Transfer B->A: Berhasil!")

# Jalankan
t1 = threading.Thread(target=transfer_A_ke_B)
t2 = threading.Thread(target=transfer_B_ke_A)

t1.start()
t2.start()

# Gunakan timeout untuk deteksi deadlock
t1.join(timeout=3)
t2.join(timeout=3)

if t1.is_alive() or t2.is_alive():
    print("\n*** DEADLOCK TERDETEKSI! ***")
    print("Thread 1 pegang Lock A, tunggu Lock B")
    print("Thread 2 pegang Lock B, tunggu Lock A")

```

Listing 5: Deadlock

3.5.2 Output Program

```

(venv) PS C:\Users\User\Python\pertemuan ke 6> python L5deadlock.py
Transfer A->B: Mengunci rekening A...
Transfer A->B: Rekening A terkunci!
Transfer B->A: Mengunci rekening B...
Transfer B->A: Rekening B terkunci!
Transfer A->B: Mencoba kunci rekening B...
Transfer B->A: Mencoba kunci rekening A...

*** DEADLOCK TERDETEKSI! ***
Thread 1 pegang Lock A, tunggu Lock B
Thread 2 pegang Lock B, tunggu Lock A

```

3.5.3 Analisis

1. Mengapa program mengalami deadlock?

Jawaban: “Karena kedua thread saling menunggu lock yang dipegang oleh thread lain.”

2. Syarat deadlock mana yang terpenuhi?

Jawaban: “Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait.”

3.6 Latihan 6: Solusi Deadlock-Lock Ordering

3.6.1 Kode Program

```
import threading
import time

rekening_A = threading.Lock()
rekening_B = threading.Lock()

def transfer_dengan_ordering(dari, ke, jumlah):
    """Transfer dengan urutan lock konsisten"""
    # KUNCI: Semua thread HARUS ambil A dulu, baru B
    print(f"Transfer {dari}->{ke}: Kunci A dulu...")
    with rekening_A:
        print(f"Transfer {dari}->{ke}: A terkunci!")
        time.sleep(0.3)

        print(f"Transfer {dari}->{ke}: Kunci B...")
        with rekening_B:
            print(f"Transfer {dari}->{ke}: Berhasil! Rp{jumlah}!\n")

# Jalankan
t1 = threading.Thread(target=transfer_dengan_ordering, args=("A", "B",
10000))
t2 = threading.Thread(target=transfer_dengan_ordering, args=("B", "A",
5000))

t1.start()
t2.start()

t1.join()
t2.join()

print("SUKSES! tidak ada deadlock karena urutan lock sama")
```

Listing 6: Deadlock-Lock Ordering

3.6.2 Output Program

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/L6lock_ordering.py"
Transfer A->B: Kunci A dulu...
Transfer A->B: A terkunci!
Transfer B->A: Kunci A dulu...
Transfer A->B: Kunci B...
Transfer A->B: Berhasil! Rp10000!

Transfer B->A: A terkunci!
Transfer B->A: Kunci B...
Transfer B->A: Berhasil! Rp5000!

SUKSES! tidak ada deadlock karena urutan lock sama
```

3.6.3 Analisis

1. Mengapa Solusi ini bisa berhasil?

Jawaban: “Karena semua thread mengambil lock dengan urutan yang sama (A lalu B).”

2. Syarat deadlock mana yang diputus?

Jawaban: “Circular Wait.”

3.7 Latihan 7: Solusi Deadlock-Timeout

3.7.1 Kode Program

```
import threading
import time

lock_X = threading.Lock()
lock_Y = threading.Lock()

def proses_dengan_timeout(name, lock1, lock2):
    """Coba ambil lock dengan timeout"""
    for percobaan in range(5):
        # Coba ambil lock 1
        if lock1.acquire(timeout=1):
            print(f"{name}: Lock 1 didapat (percobaan {percobaan + 1})")
            time.sleep(0.2)

            # Coba ambil lock 2 dengan timeout
            if lock2.acquire(timeout=1):
                print(f"{name}: Lock 2 didapat! SELESAI\n")
                lock2.release()
                lock1.release()
                return
            else:
                lock1.release()
```



```
        # Timeout! Lepas lock 1
        print(f"{name}: Timeout! Lepas lock 1, coba lagi\n")
        lock1.release()
        time.sleep(0.1)
    else:
        print(f"{name}: Timeout pada lock 1\n")
        time.sleep(0.1)

    print(f"{name}: Gagal setelah 5 percobaan\n")

# Jalankan
t1 = threading.Thread(target=proses_dengan_timeout, args=("Thread-1",
lock_X, lock_Y))
t2 = threading.Thread(target=proses_dengan_timeout, args=("Thread-2",
lock_Y, lock_X))

t1.start()
t2.start()

t1.join()
t2.join()

print("Selesai! Timeout mencegah deadlock permanent")
```

Listing 7: Deadlock-Timeout

3.7.2 Output Program

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.
exe "c:/Users/User/Python/pertemuan ke 6/L7timeout.py"
Thread-1: Lock 1 didapat (percobaan 1)
Thread-2: Lock 1 didapat (percobaan 1)
Thread-1: Timeout! Lepas lock 1, coba lagi

Thread-2: Lock 2 didapat! SELESAI

Thread-1: Lock 1 didapat (percobaan 2)
Thread-1: Lock 2 didapat! SELESAI

Selesai! Timeout mencegah deadlock permanent
```

3.7.3 Analisis

1. Bagaimana timeout mencegah deadlock?

Jawaban: “Dengan melepaskan lock setelah timeout dan mencoba lagi, sehingga tidak terjadi tunggu tak terhingga.”

2. Apa kekurangan dari Solusi timeout?

Jawaban: “Dapat menyebabkan starvation dan performa menurun karena percobaan berulang.”

3.8 Latihan 8: Solusi Deadlock-Try-Lock

3.8.1 Kode Program

```
import threading
import time

lock_P = threading.Lock()
lock_Q = threading.Lock()

def proses_dengan_trylock(name, lock1, lock2):
    """Try-lock pattern"""
    for percobaan in range(10):
        # Ambil lock 1 (blocking)
        lock1.acquire()
        print(f"{name}: Lock 1 didapat (percobaan {percobaan + 1})")

        # COBA ambil lock 2 (NON-BLOCKING)
        if lock2.acquire(blocking=False):
            print(f"{name}: Lock 2 juga didapat! SELESAI\n")
            time.sleep(0.1)
            lock2.release()
            lock1.release()
            return
        else:
            # Gagal! Lepas semua lock
            print(f"{name}: Lock 2 tidak tersedia, lepas lock 1\n")
            lock1.release()
            time.sleep(0.05)

    print(f"{name}: Gagal setelah 10 percobaan\n")

# Jalankan
t1 = threading.Thread(target=proses_dengan_trylock, args=("Thread-A",
lock_P, lock_Q))
t2 = threading.Thread(target=proses_dengan_trylock, args=("Thread-B",
lock_Q, lock_P))

t1.start()
t2.start()

t1.join()
t2.join()

print("Selesai! Try-lock mencegah deadlock")
```

Listing 8: Deadlock-Try-Lock

3.8.2 Output

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python
exe "c:/Users/User/Python/pertemuan ke 6/L8try_lock.py"
Thread-A: Lock 1 didapat (percobaan 1)
Thread-A: Lock 2 juga didapat! SELESAI

Thread-B: Lock 1 didapat (percobaan 1)
Thread-B: Lock 2 juga didapat! SELESAI

Selesai! Try-lock mencegah deadlock
```

3.8.3 Analisis

1. Apa perbedaan try-lock dengan timeout?

Jawaban: "Try-lock langsung gagal jika lock tidak tersedia, sedangkan timeout menunggu sampai waktu tertentu."

2. Mana yang lebih cepat?

Jawaban: "Try-lock lebih cepat karena tidak menunggu."

4 Tugas Mandiri

4.1 Tugas 1: Database Connection Pool

4.1.1 Kode Program

```
import threading
import time
import random

# Semaphore untuk membatasi koneksi database maksimal 3
db_semaphore = threading.Semaphore(3)

def akses_database(user_id):
    """Simulasi akses database dengan connection pool"""
    print(f"User-{user_id} menunggu koneksi database...")

    with db_semaphore:
        print(f"User-{user_id} tersambung! Eksekusi query...")

        # Simulasi waktu eksekusi query (1-2 detik)
        waktu_query = random.uniform(1, 2)
        time.sleep(waktu_query)

        print(f"User-{user_id} selesai. Waktu query: {waktu_query:.2f}
detik")
```

```
def main():
    # Buat 6 user yang ingin mengakses database
    threads = []
    for i in range(1, 7):
        t = threading.Thread(target=akses_database, args=(i,))
        threads.append(t)
        t.start()

    # Tunggu semua thread selesai
    for t in threads:
        t.join()

    print("\nSemua user telah selesai mengakses database!")

if __name__ == "__main__":
    main()
```

Listing 9: Database Connection Pool

4.1.2 Output Program

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/T1database_pool.py"
User-1 menunggu koneksi database...
User-1 tersambung! Eksekusi query...
User-2 menunggu koneksi database...
User-2 tersambung! Eksekusi query...
User-3 menunggu koneksi database...
User-3 tersambung! Eksekusi query...
User-4 menunggu koneksi database...
User-6 menunggu koneksi database...
User-5 menunggu koneksi database...
User-3 selesai. Waktu query: 1.08 detik
User-4 tersambung! Eksekusi query...
User-2 selesai. Waktu query: 1.63 detik
User-6 tersambung! Eksekusi query...
User-1 selesai. Waktu query: 1.83 detik
User-5 tersambung! Eksekusi query...
User-4 selesai. Waktu query: 1.47 detik
User-5 selesai. Waktu query: 1.44 detik
User-6 selesai. Waktu query: 1.82 detik

Semua user telah selesai mengakses database!
```

4.1.3 Penjelasan

Implementasi Database Connection Pool:

- Menggunakan *Semaphore(3)* untuk membatasi maksimal koneksi database bersamaan
- 6 user mencoba mengakses database, tetapi hanya 3 yang bisa terhubung secara simultan
- User yang lain akan menunggu sampai ada koneksi yang tersedia
- Setiap query memakan waktu random antara 1-2 detik untuk mensimulasikan eksekusi query database
- Semaphore secara otomatis mengatur antrian dan mengalokasikan koneksi yang tersedia
- Keuntungan: Menghemat resource database dengan membatasi koneksi simultan

4.2 Tugas 2: Dining Philosophers Problem

4.2.1 Kode Program

```
import threading
import time
import random

class DiningPhilosophers:
    def __init__(self, num_philosophers=5):
        self.num_philosophers = num_philosophers
        # Buat lock untuk setiap garpu
        self.forks = [threading.Lock() for _ in range(num_philosophers)]
        # Lock untuk memastikan hanya satu filsuf yang mengambil garpu
        # sekaligus
        self.table_lock = threading.Lock()

    def filosof(self, philosopher_id):
        """Simulasi perilaku filsuf dengan solusi lock ordering"""
        left_fork = philosopher_id
        right_fork = (philosopher_id + 1) % self.num_philosophers

        for makan_ke in range(2): # Setiap filsuf makan 2 kali
            # Fase berpikir
            print(f"Filsuf-{philosopher_id} sedang berpikir...")
            time.sleep(random.uniform(0.1, 0.3))

            # **SOLUSI: LOCK ORDERING - selalu ambil garpu dengan nomor
            # lebih kecil dulu**
            first_fork = min(left_fork, right_fork)
            second_fork = max(left_fork, right_fork)

            # Ambil garpu pertama (yang nomornya lebih kecil)
            print(f"Filsuf-{philosopher_id} mencoba mengambil garpu
            {first_fork}")
            self.forks[first_fork].acquire()
            print(f"Filsuf-{philosopher_id} berhasil mengambil garpu
            {first_fork}")
```

```

        # Ambil garpu kedua (yang nomornya lebih besar)
        print(f"Filsuf-{philosopher_id} mencoba mengambil garpu
{second_fork}")
        self.forks[second_fork].acquire()
        print(f"Filsuf-{philosopher_id} berhasil mengambil garpu
{second_fork}")

        # Fase makan
        print(f"Filsuf-{philosopher_id} ✓✓✓ MAKAN untuk ke-{makan_ke +
1} kali")
        time.sleep(random.uniform(0.5, 1.0))

        # Lepas garpu (urutan tidak penting)
        self.forks[second_fork].release()
        print(f"Filsuf-{philosopher_id} melepaskan garpu {second_fork}")

        self.forks[first_fork].release()
        print(f"Filsuf-{philosopher_id} melepaskan garpu {first_fork}")

        print(f"Filsuf-{philosopher_id} selesai makan ke-{makan_ke +
1}\n")

def main():
    print("=" * 70)
    print("DINING PHILOSOPHERS PROBLEM")
    print("Solusi: Lock Ordering")
    print("=" * 70)
    print("Aturan: Selalu ambil garpu dengan nomor lebih kecil terlebih
dahulu")
    print("=" * 70)

    dining_table = DiningPhilosophers(5)

    # Buat thread untuk setiap filsuf
    philosophers = []
    for i in range(5):
        philosopher = threading.Thread(target=dining_table.filosof,
args=(i,))
        philosophers.append(philosopher)
        print(f"Filsuf-{i} telah duduk di meja")

    print("\n" + "="*30 + " MULAI " + "="*30)

    # Jalankan semua thread
    for philosopher in philosophers:
        philosopher.start()

```

```
# Tunggu semua filsuf selesai
for philosopher in philosophers:
    philosopher.join()

print("="*50)
print("SEMUA FILSUF TELAH SELESAI MAKAN!")
print("Tidak terjadi deadlock berkat Lock Ordering")
print("="*50)

if __name__ == "__main__":
    main()
```

Listing 10: Dining Philosophers Problem

4.2.2 Output Program

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/T2philosophers_problem.py"
=====
DINING PHILOSOPHERS PROBLEM
Solusi: Lock Ordering
=====
Aturan: Selalu ambil garpu dengan nomor lebih kecil terlebih dahulu
=====
Filsuf-0 telah duduk di meja
Filsuf-1 telah duduk di meja
Filsuf-2 telah duduk di meja
Filsuf-3 telah duduk di meja
Filsuf-4 telah duduk di meja

===== MULAI =====
Filsuf-0 sedang berpikir...
Filsuf-1 sedang berpikir...
Filsuf-2 sedang berpikir...
Filsuf-3 sedang berpikir...
Filsuf-4 sedang berpikir...
Filsuf-3 mencoba mengambil garpu 3
Filsuf-3 berhasil mengambil garpu 3
Filsuf-3 mencoba mengambil garpu 4
Filsuf-3 berhasil mengambil garpu 4
Filsuf-3 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-4 mencoba mengambil garpu 0
Filsuf-4 berhasil mengambil garpu 0
Filsuf-4 mencoba mengambil garpu 4
Filsuf-1 mencoba mengambil garpu 1
Filsuf-1 berhasil mengambil garpu 1
Filsuf-1 mencoba mengambil garpu 2
Filsuf-1 berhasil mengambil garpu 2
Filsuf-1 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-0 mencoba mengambil garpu 0
Filsuf-2 mencoba mengambil garpu 2
Filsuf-1 melepaskan garpu 2
Filsuf-2 berhasil mengambil garpu 2
Filsuf-1 melepaskan garpu 1
Filsuf-2 mencoba mengambil garpu 3
Filsuf-1 selesai makan ke-1
```



```
Filsuf-1 sedang berpikir...
Filsuf-3 melepaskan garpu 4
Filsuf-4 berhasil mengambil garpu 4
Filsuf-4 √√√ MAKAN untuk ke-1 kali
Filsuf-3 melepaskan garpu 3
Filsuf-2 berhasil mengambil garpu 3
Filsuf-3 selesai makan ke-1
Filsuf-2 √√√ MAKAN untuk ke-1 kali

Filsuf-3 sedang berpikir...
Filsuf-1 mencoba mengambil garpu 1
Filsuf-1 berhasil mengambil garpu 1
Filsuf-1 mencoba mengambil garpu 2
Filsuf-3 mencoba mengambil garpu 3
Filsuf-4 melepaskan garpu 4
Filsuf-4 melepaskan garpu 0
Filsuf-0 berhasil mengambil garpu 0
Filsuf-0 mencoba mengambil garpu 1
Filsuf-4 selesai makan ke-1

Filsuf-4 sedang berpikir...
Filsuf-4 mencoba mengambil garpu 0
Filsuf-2 melepaskan garpu 3
Filsuf-3 berhasil mengambil garpu 3
Filsuf-1 berhasil mengambil garpu 2
Filsuf-3 mencoba mengambil garpu 4
Filsuf-1 √√√ MAKAN untuk ke-2 kali
Filsuf-3 berhasil mengambil garpu 4
Filsuf-2 melepaskan garpu 2
Filsuf-3 √√√ MAKAN untuk ke-2 kali
Filsuf-2 selesai makan ke-1

Filsuf-2 sedang berpikir...
Filsuf-2 mencoba mengambil garpu 2
Filsuf-1 melepaskan garpu 2
Filsuf-2 berhasil mengambil garpu 2
Filsuf-2 mencoba mengambil garpu 3
Filsuf-0 berhasil mengambil garpu 1
Filsuf-1 melepaskan garpu 1
Filsuf-0 √√√ MAKAN untuk ke-1 kali
Filsuf-1 selesai makan ke-2

Filsuf-3 melepaskan garpu 4
Filsuf-3 melepaskan garpu 3
Filsuf-3 selesai makan ke-2
```

```

Filsuf-2 berhasil mengambil garpu 3
Filsuf-2 ✓✓✓ MAKAN untuk ke-2 kali
Filsuf-2 melepaskan garpu 3
Filsuf-2 melepaskan garpu 2
Filsuf-2 selesai makan ke-2

Filsuf-0 melepaskan garpu 1
Filsuf-0 melepaskan garpu 0
Filsuf-0 selesai makan ke-1

Filsuf-0 sedang berpikir...
Filsuf-4 berhasil mengambil garpu 0
Filsuf-4 mencoba mengambil garpu 4
Filsuf-4 berhasil mengambil garpu 4
Filsuf-4 ✓✓✓ MAKAN untuk ke-2 kali
Filsuf-0 mencoba mengambil garpu 0
Filsuf-4 melepaskan garpu 4
Filsuf-4 melepaskan garpu 0
Filsuf-0 berhasil mengambil garpu 0
Filsuf-4 selesai makan ke-2
Filsuf-0 mencoba mengambil garpu 1
Filsuf-0 berhasil mengambil garpu 1

Filsuf-0 ✓✓✓ MAKAN untuk ke-2 kali
Filsuf-0 melepaskan garpu 1
Filsuf-0 melepaskan garpu 0
Filsuf-0 selesai makan ke-2

=====
SEMUA FILSUF TELAH SELESAI MAKAN!
Tidak terjadi deadlock berkat Lock Ordering
=====

```

4.2.3 Penjelasan

Implementasi Dining Philosophers Problem menggunakan solusi Lock Ordering. Setiap filsuf harus selalu mengambil garpu dengan nomor lebih kecil terlebih dahulu, baru kemudian mengambil garpu dengan nomor lebih besar. Strategi ini mencegah deadlock dengan memutus circular wait. Program berjalan tanpa deadlock, semua filsuf berhasil makan sesuai jadwal, dan resource (garpu) digunakan secara efisien.

1. Masalah Dining Philosophers:

- 5 filsuf duduk melingkar dengan 5 garpu di antara mereka
- Setiap filsuf membutuhkan 2 garpu (kiri dan kanan) untuk bisa makan
- Jika semua filsuf mengambil garpu kiri bersamaan, terjadi deadlock – semua menunggu garpu kanan yang tidak pernah tersedia

2. Solusi yang Diimplementasikan: Lock Ordering

- a. Strategi Lock Ordering
 - Setiap filsuf harus selalu mengambil garpu dengan nomor lebih kecil terlebih dahulu
 - Baru kemudian mengambil garpu dengan nomor lebih besar
 - b. Contoh Implementasi:
 - Filsuf-0: garpu kiri=0, garpu kanan=1 → ambil garpu 0 dulu, baru garpu 1
 - Filsuf-1: garpu kiri=1, garpu kanan=2 → ambil garpu 1 dulu, baru garpu 2
 - Filsuf-4: garpu kiri=4, garpu kanan=0 → ambil garpu 0 dulu, baru garpu 4
3. Mengapa Solusi Ini Mencegah Deadlock:
- a. Memutuskan Circular Wait:
 - Dengan aturan urutan yang konsisten tidak mungkin terjadi siklus tunggu melingkar
 - Jika Filsuf-4 ingin makan, dia harus menunggu garpu 0 (yang lebih kecil dari garpu 4)
 - b. Konsisten Global:
 - Semua filsuf mengikuti aturan yang sama
 - Tidak ada konflik dalam urutan pengambilan lock
 - c. Starvation Prevention:
 - Setiap filsuf pasti mendapatkan kesempatan makan
 - Tidak ada filsuf yang harus terus menerus ditolak aksesnya
4. Struktur Program:
- Kelas *DiningPhilosophers*: Mengelola state meja makan
 - Array *forks*: 5 lock yang merepresentasikan garpu
 - Method *filosof*: Perilaku setiap filsuf (berpikir → ambil garpu → makan → lepas garpu)
 - Loob makan: Setiap filsuf makan 2 kali untuk demonstrasi
5. Keuntungan Solusi Ini:
- Sederhana dan mudah dipahami
 - Efisien tanpa overhead yang besar
 - Deterministik – perilaku dapat diprediksi
 - Tidak memerlukan mekanisme kompleks seperti timeout atau try-lock
- Hasil: Program berjalan tanpa deadlock, semua filsuf berhasil makan sesuai jadwal, dan resource (garpu) digunakan secara efisien.

5 Kesimpulan

Berdasarkan praktikum yang telah dilakukan, dapat disimpulkan:

1. Lock dan Semaphore efektif untuk menyelesaikan masalah sinkronisasi dalam pemrograman paralel. Lock mencegah race condition, sedangkan Semaphore membatasi akses ke resource terbatas.
2. Deadlock dapat dicegah dengan tiga strategi utama: Lock Ordering (paling efektif), Timeout, dan Try-Lock. Lock Ordering berhasil menyelesaikan Dining Philosophers Problem tanpa deadlock.
3. Implementasi praktis berhasil ditunjukkan melalui Database Connction Pool dengan Semaphore dan Dining Philosophers dengan Lock Ordering.
4. Python threading library menyediakan tools yang memadai untukn implementasi konkurensi dengan sintaks yang sederhana.

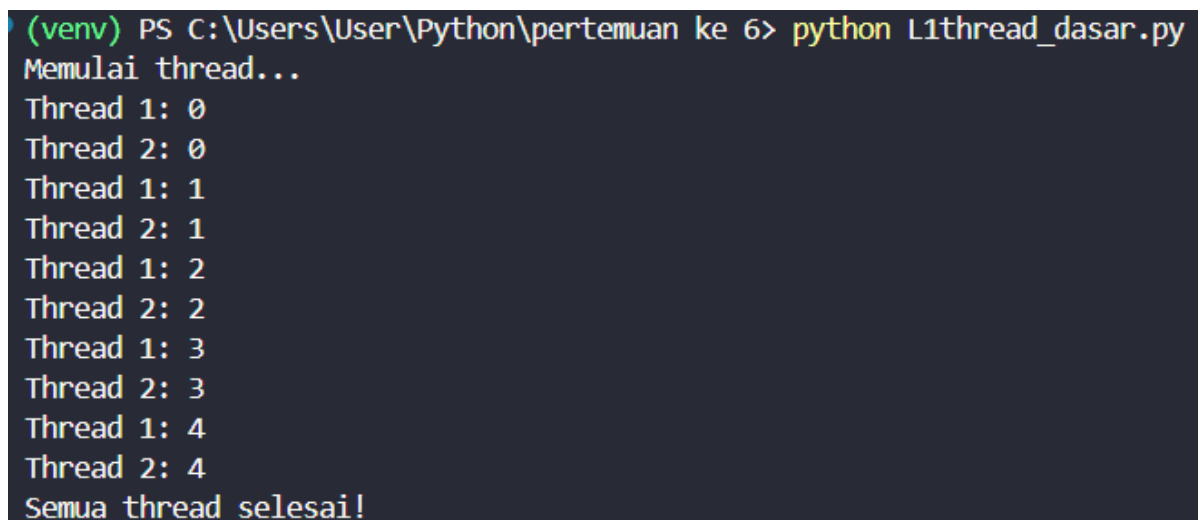
6 Saran

Untuk pengembangan praktikum selanjutnya:

1. Tambahkan studi kasus real-word seperti web server pooling atau messagr queue systems.
2. Sertakan tools debugging konkurensi dan teknis profilinf untuk analisis performance.

7 Lampiran

7.1 SS hasil Program



```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L1thread_dasar.py
Memulai thread...
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Semua thread selesai!
```

Gambar 1. Hasil Thread Dasar

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L2race_condition.py
Counter akhir: 200000
Harusnya: 200000
Selisih: 0
```

Gambar 2. Hasil Race Condition

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L3lock.py
Counter akhir: 200000
Harusnya: 200000
Selisih: 0
SUKSES! Data aman dengan Lock
```

Gambar 3. Hasil dengan Lock

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L4semaphore.py
Ipal menunggu printer...
Ipal mulai print 1 halaman
Ipul menunggu printer...
Yogi menunggu printer...
Ipul mulai print 3 halaman
Yoga menunggu printer...
Udins menunggu printer...
Ipal selesai print
Yogi mulai print 4 halaman
Ipul selesai print
Yoga mulai print 2 halaman
Yogi selesai print
Udins mulai print 1 halaman
Yoga selesai print
Udins selesai print

Semua dokumen selesai dicetak!
```

Gambar 4. Hasil dengan Semaphore

```
(venv) PS C:\Users\User\Python\pertemuan ke 6> python L5deadlock.py
Transfer A->B: Mengunci rekening A...
Transfer A->B: Rekening A terkunci!
Transfer B->A: Mengunci rekening B...
Transfer B->A: Rekening B terkunci!
Transfer A->B: Mencoba kunci rekening B...
Transfer B->A: Mencoba kunci rekening A...

*** DEADLOCK TERDETEKSI! ***
Thread 1 pegang Lock A, tunggu Lock B
Thread 2 pegang Lock B, tunggu Lock A
```

Gambar 5. Hasil jika terjadi Deadlock

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.
exe "c:/Users/User/Python/pertemuan ke 6/L6lock_ordering.py"
Transfer A->B: Kunci A dulu...
Transfer A->B: A terkunci!
Transfer B->A: Kunci A dulu...
Transfer A->B: Kunci B...
Transfer A->B: Berhasil! Rp10000!

Transfer B->A: A terkunci!
Transfer B->A: Kunci B...
Transfer B->A: Berhasil! Rp5000!

SUKSES! tidak ada deadlock karena urutan lock sama
```

Gambar 6. Hasil solusi deadlock dengan lock ordering

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.
exe "c:/Users/User/Python/pertemuan ke 6/L7timeout.py"
Thread-1: Lock 1 didapat (percobaan 1)
Thread-2: Lock 1 didapat (percobaan 1)
Thread-1: Timeout! Lepas lock 1, coba lagi

Thread-2: Lock 2 didapat! SELESAI

Thread-1: Lock 1 didapat (percobaan 2)
Thread-1: Lock 2 didapat! SELESAI

Selesai! Timeout mencegah deadlock permanent
```

Gambar 7. Hasil solusi deadlock dengan timeout

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/L8try_lock.py"
Thread-A: Lock 1 didapat (percobaan 1)
Thread-A: Lock 2 juga didapat! SELESAI

Thread-B: Lock 1 didapat (percobaan 1)
Thread-B: Lock 2 juga didapat! SELESAI

Selesai! Try-lock mencegah deadlock
```

Gambar 8. Hasil solusi deadlock dengan try lock

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/T1database_pool.py"
User-1 menunggu koneksi database...
User-1 tersambung! Eksekusi query...
User-2 menunggu koneksi database...
User-2 tersambung! Eksekusi query...
User-3 menunggu koneksi database...
User-3 tersambung! Eksekusi query...
User-4 menunggu koneksi database...
User-6 menunggu koneksi database...
User-5 menunggu koneksi database...
User-3 selesai. Waktu query: 1.08 detik
User-4 tersambung! Eksekusi query...
User-2 selesai. Waktu query: 1.63 detik
User-6 tersambung! Eksekusi query...
User-1 selesai. Waktu query: 1.83 detik
User-5 tersambung! Eksekusi query...
User-4 selesai. Waktu query: 1.47 detik
User-5 selesai. Waktu query: 1.44 detik
User-6 selesai. Waktu query: 1.82 detik

Semua user telah selesai mengakses database!
```

Gambar 9. Hasil tugas 1 database connection pool

```
(venv) PS C:\Users\User\Python> & C:/Users/User/Python/venv/Scripts/python.exe "c:/Users/User/Python/pertemuan ke 6/T2philosophers_problem.py"
=====
DINING PHILOSOPHERS PROBLEM
Solusi: Lock Ordering
=====
Aturan: Selalu ambil garpu dengan nomor lebih kecil terlebih dahulu
=====
Filsuf-0 telah duduk di meja
Filsuf-1 telah duduk di meja
Filsuf-2 telah duduk di meja
Filsuf-3 telah duduk di meja
Filsuf-4 telah duduk di meja

===== MULAI =====
Filsuf-0 sedang berpikir...
Filsuf-1 sedang berpikir...
Filsuf-2 sedang berpikir...
Filsuf-3 sedang berpikir...
Filsuf-4 sedang berpikir...
Filsuf-3 mencoba mengambil garpu 3
Filsuf-3 berhasil mengambil garpu 3
Filsuf-3 mencoba mengambil garpu 4
Filsuf-3 berhasil mengambil garpu 4
Filsuf-3 √√√ MAKAN untuk ke-1 kali
Filsuf-4 mencoba mengambil garpu 0
Filsuf-4 berhasil mengambil garpu 0
Filsuf-4 mencoba mengambil garpu 4
Filsuf-1 mencoba mengambil garpu 1
Filsuf-1 berhasil mengambil garpu 1
Filsuf-1 mencoba mengambil garpu 2
Filsuf-1 berhasil mengambil garpu 2
Filsuf-1 √√√ MAKAN untuk ke-1 kali
Filsuf-0 mencoba mengambil garpu 0
Filsuf-2 mencoba mengambil garpu 2
Filsuf-1 melepaskan garpu 2
Filsuf-2 berhasil mengambil garpu 2
Filsuf-1 melepaskan garpu 1
Filsuf-2 mencoba mengambil garpu 3
Filsuf-1 selesai makan ke-1
```



```
Filsuf-1 sedang berpikir...
Filsuf-3 melepaskan garpu 4
Filsuf-4 berhasil mengambil garpu 4
Filsuf-4 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-3 melepaskan garpu 3
Filsuf-2 berhasil mengambil garpu 3
Filsuf-3 selesai makan ke-1
Filsuf-2 ✓✓✓ MAKAN untuk ke-1 kali

Filsuf-3 sedang berpikir...
Filsuf-1 mencoba mengambil garpu 1
Filsuf-1 berhasil mengambil garpu 1
Filsuf-1 mencoba mengambil garpu 2
Filsuf-3 mencoba mengambil garpu 3
Filsuf-4 melepaskan garpu 4
Filsuf-4 melepaskan garpu 0
Filsuf-0 berhasil mengambil garpu 0
Filsuf-0 mencoba mengambil garpu 1
Filsuf-4 selesai makan ke-1

Filsuf-4 sedang berpikir...
Filsuf-4 mencoba mengambil garpu 0
Filsuf-2 melepaskan garpu 3
Filsuf-3 berhasil mengambil garpu 3
Filsuf-1 berhasil mengambil garpu 2
Filsuf-3 mencoba mengambil garpu 4
Filsuf-1 ✓✓✓ MAKAN untuk ke-2 kali
Filsuf-3 berhasil mengambil garpu 4
Filsuf-2 melepaskan garpu 2
Filsuf-3 ✓✓✓ MAKAN untuk ke-2 kali
Filsuf-2 selesai makan ke-1

Filsuf-2 sedang berpikir...
Filsuf-2 mencoba mengambil garpu 2
Filsuf-1 melepaskan garpu 2
Filsuf-2 berhasil mengambil garpu 2
Filsuf-2 mencoba mengambil garpu 3
Filsuf-0 berhasil mengambil garpu 1
Filsuf-1 melepaskan garpu 1
Filsuf-0 ✓✓✓ MAKAN untuk ke-1 kali
Filsuf-1 selesai makan ke-2

Filsuf-3 melepaskan garpu 4
Filsuf-3 melepaskan garpu 3
Filsuf-3 selesai makan ke-2
```

```
Filsuf-2 berhasil mengambil garpu 3
Filsuf-2 √√/ MAKAN untuk ke-2 kali
Filsuf-2 melepaskan garpu 3
Filsuf-2 melepaskan garpu 2
Filsuf-2 selesai makan ke-2

Filsuf-0 melepaskan garpu 1
Filsuf-0 melepaskan garpu 0
Filsuf-0 selesai makan ke-1

Filsuf-0 sedang berpikir...
Filsuf-4 berhasil mengambil garpu 0
Filsuf-4 mencoba mengambil garpu 4
Filsuf-4 berhasil mengambil garpu 4
Filsuf-4 √√/ MAKAN untuk ke-2 kali
Filsuf-0 mencoba mengambil garpu 0
Filsuf-4 melepaskan garpu 4
Filsuf-4 melepaskan garpu 0
Filsuf-0 berhasil mengambil garpu 0
Filsuf-4 selesai makan ke-2
Filsuf-0 mencoba mengambil garpu 1
Filsuf-0 berhasil mengambil garpu 1

Filsuf-0 √√/ MAKAN untuk ke-2 kali
Filsuf-0 melepaskan garpu 1
Filsuf-0 melepaskan garpu 0
Filsuf-0 selesai makan ke-2
```

```
=====
SEMUA FILSUF TELAH SELESAI MAKAN!
Tidak terjadi deadlock berkat Lock Ordering
=====
```

Gambar 10. Hasil tugas 2 Dining Philosophers Problem

7.2 Link Repository

<https://github.com/Rivaldi-231220056/Komter-Pertemuan-ke-6>