Nama: Daffa Harikhsan
NIM: 23/513044/PA/21918

# Tugas 5

**Activity 7.1**

Before Modify

simple_thread.c > thread_function

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *thread_function(void *arg);
int i,j;

int main() {

    pthread_t a_thread;
    pthread_create(newthread: &a_thread, attr: NULL, start_routine: thread_function, arg: NULL);
    pthread_join(th: a_thread, thread_return: NULL);
    printf(format: "Inside Main Program\n");
    for(j=4;j>=0;j--){
        printf(format: "%d\n",j);
        sleep(seconds: 1);
    }
}
    void *thread_function(void *arg) {
        printf(format: "Inside Thread\n");
        for(i=0;i<5;i++){
            printf(format: "%d\n",i);
            sleep(seconds: 1);
        }
    }
```

```
dharikhsan@cloudshell:~$ gcc -o simple_thread simple_thread.c -lpthread
dharikhsan@cloudshell:~$ ./simple_thread
Inside Thread
0
1
2
3
4
Inside Main Program
4
3
2
1
0
dharikhsan@cloudshell:~$
```

Output menunjukkan bahwa thread berjalan terlebih dahulu (mencetak "Inside Thread" dan angka dari 0 hingga 4), kemudian setelah thread selesai, program utama mencetak "Inside Main Program" dan angka dari 4 hingga 0. Hasil ini menunjukkan bahwa pthread_join() berfungsi dengan baik karena program utama menunggu thread selesai sebelum melanjutkan eksekusi.

After Modify

C simple_thread.c > thread_function

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *thread_function(void *arg);
int i,j;

int main() {

    pthread_t a_thread;
    pthread_create(newthread: &a_thread, attr: NULL, start_routine: thread_function, arg: NULL);
    //pthread_join(a_thread, NULL);
    printf(format: "Inside Main Program\n");
    for(j=4;j>=0;j--){
        printf(format: "%d\n",j);
        sleep(seconds: 1);
    }
}
    void *thread_function(void *arg) {
        printf(format: "Inside Thread\n");
        for(i=0;i<5;i++){
            printf(format: "%d\n",i);
            sleep(seconds: 1);
        }
    }
```

```
dharikhsan@cloudshell:~$ gcc -o simple_thread simple_thread.c -lpthread
dharikhsan@cloudshell:~$ ./simple_thread
Inside Main Program
4
Inside Thread
0
3
1
2
2
1
3
0
4
dharikhsan@cloudshell:~$
```

Jika pthread_join(a_thread, NULL); kita comment (//pthread_join(a_thread, NULL);) maka akan menyebabkan thread dan main program berjalan secara paralel tanpa saling menunggu. Hasilnya, output program menjadi tidak terprediksi dan bergantung pada penjadwalan CPU oleh sistem operasi. Meski program bisa berjalan lebih cepat, sinkronisasi antara thread dan program utama hilang, yang berpotensi menyebabkan masalah logika atau output yang tidak lengkap jika program utama selesai sebelum thread menyelesaikan tugasnya. Dalam kebanyakan kasus, terutama untuk program yang membutuhkan urutan eksekusi yang jelas, pthread_join() sangat penting untuk memastikan thread menyelesaikan pekerjaannya sebelum program utama berakhir.

**Activity 7.2**

inout_thread.c > main

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
struct arg_struct
{
int a;
int b;
int sum;
};
void *addition(void *arguments){
struct arg_struct *args = arguments;
args -> sum = args -> a + args -> b;
pthread_exit(retval: NULL);
}
int main(){
pthread_t t;
struct arg_struct *args = malloc(size: sizeof *args);
args -> a = 10;
args -> b = 5;
pthread_create(newthread: &t, attr: NULL, start_routine: addition, arg: args);

pthread_join(th: t, thread_return: NULL);
printf(format: "%d + %d = %d\n", args -> a, args -> b, args -> sum);
}
```

```
dharikhsan@cloudshell:~$ gcc -o inout_thread inout_thread.c -lpthread
dharikhsan@cloudshell:~$ ./inout_thread
10 + 5 = 15
dharikhsan@cloudshell:~$
```

Program ini menggunakan thread untuk menjalankan operasi penjumlahan dua bilangan (a = 10 dan b = 5). Fungsi thread menghitung penjumlahan a + b, menyimpannya dalam variabel sum, dan setelah thread selesai, hasil penjumlahan dicetak oleh program utama. Output akhirnya adalah 10 + 5 = 15, yang mencerminkan hasil operasi tersebut. Threading memungkinkan pemisahan tugas dan dapat meningkatkan efisiensi di program yang lebih kompleks, namun sinkronisasi dengan pthread_join() diperlukan untuk memastikan urutan eksekusi yang tepat.

Intinya program ini menggunakan *thread* untuk menjalankan operasi penjumlahan dua angka, Setelah *thread* selesai menghitung, program utama mencetak hasilnya., Menggunakan *thread* memungkinkan tugas berjalan secara paralel, meskipun dalam contoh ini hanya ada satu tugas tambahan.

**Deadlock**

Before modify

deadlock.c > function1

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void *function1();
void *function2();
pthread_mutex_t res_a;
pthread_mutex_t res_b;

int main() {
pthread_mutex_init(mutex: &res_a,mutexattr: NULL);
pthread_mutex_init(mutex: &res_b,mutexattr: NULL);
pthread_t one, two;
pthread_create(newthread: &one, attr: NULL, start_routine: function1, arg: NULL); // create thread
pthread_create(newthread: &two, attr: NULL, start_routine: function2, arg: NULL);
pthread_join(th: one, thread_return: NULL);
pthread_join(th: two, thread_return: NULL);
printf(format: "Thread joined\n");

}

void *function1() {
pthread_mutex_lock(mutex: &res_a);
printf(format: "Thread ONE acquired res_a\n");
sleep(seconds: 1);
pthread_mutex_lock(mutex: &res_b);
printf(format: "Thread ONE acquired res_b\n");
pthread_mutex_unlock(mutex: &res_b);
printf(format: "Thread ONE released res_b\n");
pthread_mutex_unlock(mutex: &res_a);
printf(format: "Thread ONE released res_a\n");

}

void *function2() {
pthread_mutex_lock(mutex: &res_b);
printf(format: "Thread TWO acquired res_b\n");
sleep(seconds: 1);
pthread_mutex_lock(mutex: &res_a);
printf(format: "Thread TWO acquired res_a\n");

pthread_mutex_unlock(mutex: &res_a);
printf(format: "Thread TWO released res_a\n");
pthread_mutex_unlock(mutex: &res_b);
printf(format: "Thread TWO released res_b\n");

}
```

```
dharikhsan@cloudshell:~$ gcc -o deadlock deadlock.c -lpthread
dharikhsan@cloudshell:~$ ./deadlock
Thread ONE acquired res_a
Thread TWO acquired res_b
```

Kode tersebut terjadi deadlock karena dua thread saling menunggu untuk mendapatkan mutex yang sudah dikunci oleh thread lain. Kondisi ini menggambarkan pentingnya menggunakan urutan penguncian yang konsisten dan strategi manajemen sumber daya yang efektif dalam desain sistem multithreading untuk menghindari deadlock dan memastikan stabilitas sistem.

Terkait dengan output yang ditampilkan di kode, output tersebut secara langsung menunjukkan terjadinya deadlock:

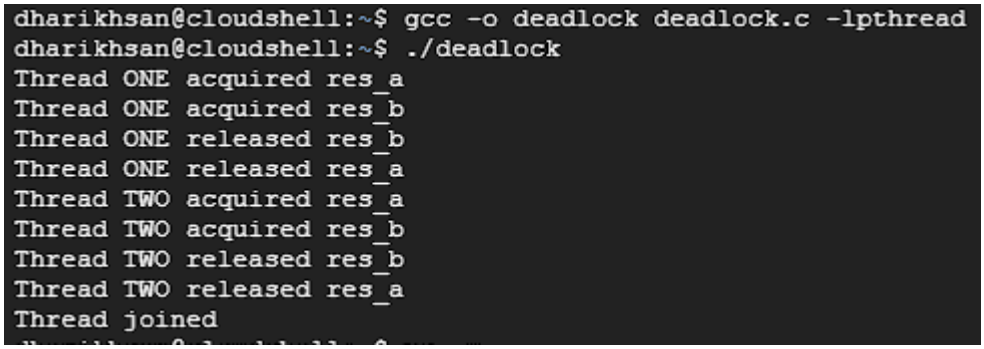
- Output "Thread ONE acquired res_a" menunjukkan bahwa Thread 1 telah berhasil mengunci sumber daya res_a.
- Output "Thread TWO acquired res_b" menunjukkan bahwa Thread 2 telah berhasil mengunci sumber daya res_b.

Kedua thread kemudian masuk dalam keadaan menunggu untuk mendapatkan mutex kedua yang sudah dikunci oleh thread lain, sehingga tidak ada lagi output yang muncul setelah itu. Ini mengindikasikan bahwa kedua thread tersebut tidak bisa melanjutkan eksekusi dan terjebak dalam deadlock, sesuai dengan keadaan yang mereka alami saat ini.

After Modify

deadlock.c > main

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void *function1();
void *function2();
pthread_mutex_t res_a;
pthread_mutex_t res_b;
int main() {
    pthread_mutex_init(mutex: &res_a,mutexattr: NULL);
    pthread_mutex_init(mutex: &res_b,mutexattr: NULL);
    pthread_t one, two;
    pthread_create(newthread: &one, attr: NULL, start_routine: function1, arg: NULL); // create thread
    pthread_create(newthread: &two, attr: NULL, start_routine: function2, arg: NULL);
    pthread_join(th: one, thread_return: NULL);
    pthread_join(th: two, thread_return: NULL);
    printf(format: "Thread joined\n");
}

void *function1() {
 pthread_mutex_lock(mutex: &res_a);
 printf(format: "Thread ONE acquired res_a\n");
 sleep(seconds: 1);

 pthread_mutex_lock(mutex: &res_b);
 printf(format: "Thread ONE acquired res_b\n");

 pthread_mutex_unlock(mutex: &res_b);
 printf(format: "Thread ONE released res_b\n");

 pthread_mutex_unlock(mutex: &res_a);
 printf(format: "Thread ONE released res_a\n");
}

void *function2() {
 pthread_mutex_lock(mutex: &res_a);
 printf(format: "Thread TWO acquired res_a\n");
 sleep(seconds: 1);

 pthread_mutex_lock(mutex: &res_b);
 printf(format: "Thread TWO acquired res_b\n");

 pthread_mutex_unlock(mutex: &res_b);
 printf(format: "Thread TWO released res_b\n");

 pthread_mutex_unlock(mutex: &res_a);
 printf(format: "Thread TWO released res_a\n");
}
```




Perubahan urutan akses pada function2 menyebabkan penguncian dan pelepasan mutex telah dikelola dengan sukses, memungkinkan kedua thread untuk menyelesaikan tugas mereka secara berurutan tanpa masuk ke dalam kondisi deadlock. Dengan menggunakan mekanisme penguncian yang konsisten dan terkoordinasi, kedua thread secara efektif menghindari situasi di mana salah satu thread terhalang oleh yang lain, memastikan eksekusi yang lancar dan sekuensial dari tugas-tugas yang bergantung pada sumber daya bersama. Hal ini menegaskan pentingnya desain dan implementasi yang tepat dalam manajemen thread dan sumber daya dalam aplikasi multithreading.