

Tugas Praktikum SKJ

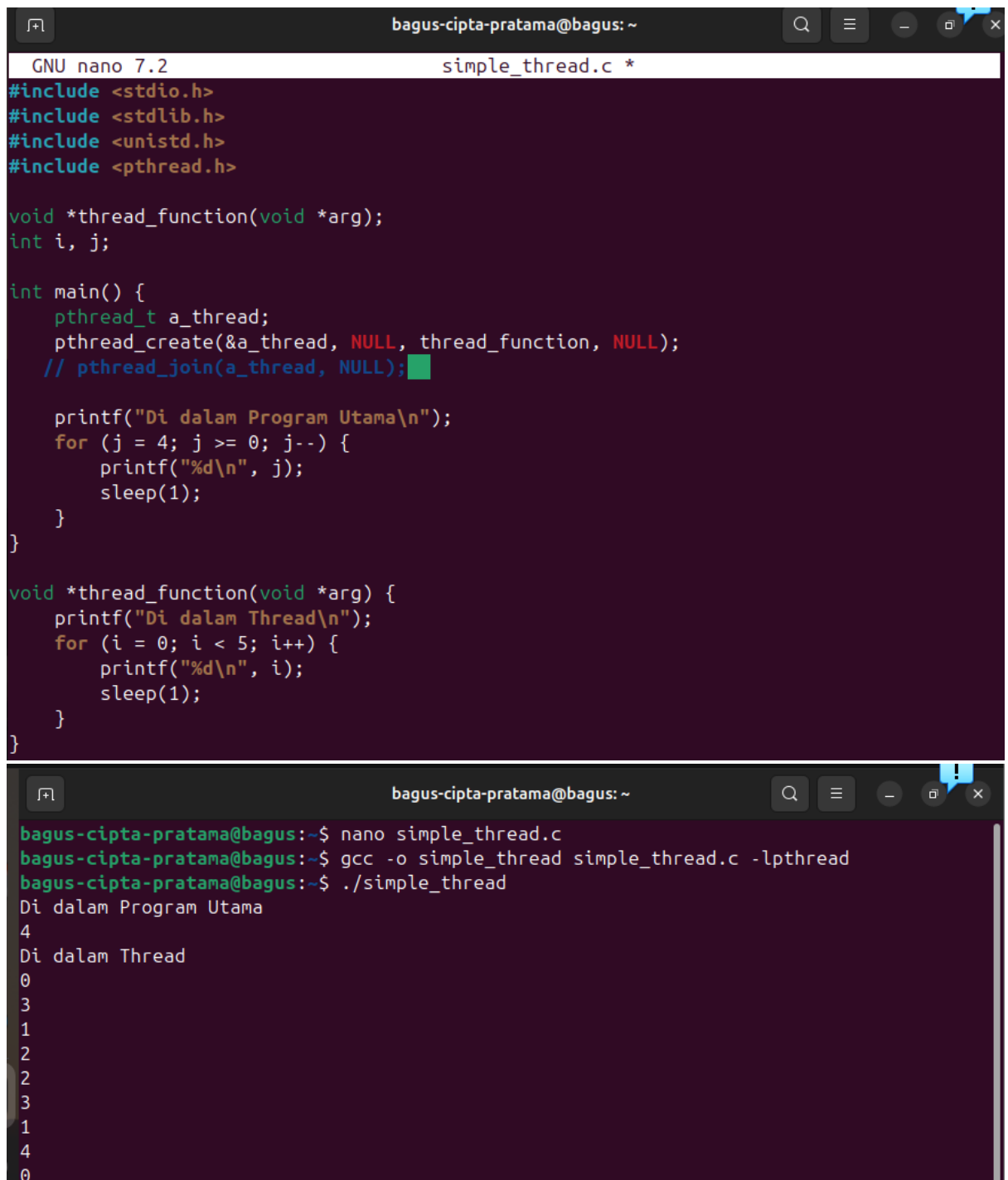
Ke – 5

Nama : Bagus Cipta Pratama

NIM : 23/516539/PA/22097

Kelas : KOMC

1. Activity 7.1 : Modifikasi *pthread_join*



```
GNU nano 7.2 simple_thread.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(&a_thread, NULL, thread_function, NULL);
    // pthread_join(a_thread, NULL);

    printf("Di dalam Program Utama\n");
    for (j = 4; j >= 0; j--) {
        printf("%d\n", j);
        sleep(1);
    }
}

void *thread_function(void *arg) {
    printf("Di dalam Thread\n");
    for (i = 0; i < 5; i++) {
        printf("%d\n", i);
        sleep(1);
    }
}

bagus-cipta-pratama@bagus: ~
bagus-cipta-pratama@bagus:~$ nano simple_thread.c
bagus-cipta-pratama@bagus:~$ gcc -o simple_thread simple_thread.c -lpthread
bagus-cipta-pratama@bagus:~$ ./simple_thread
Di dalam Program Utama
4
Di dalam Thread
0
3
1
2
2
3
1
4
0
```

Pada aktivitas ini ada dua tujuan utama yang ingin dicapai . yang pertama adalah memahami bagaimana main thread dan child thread bekerja secara simultan tanpa menunggu satu

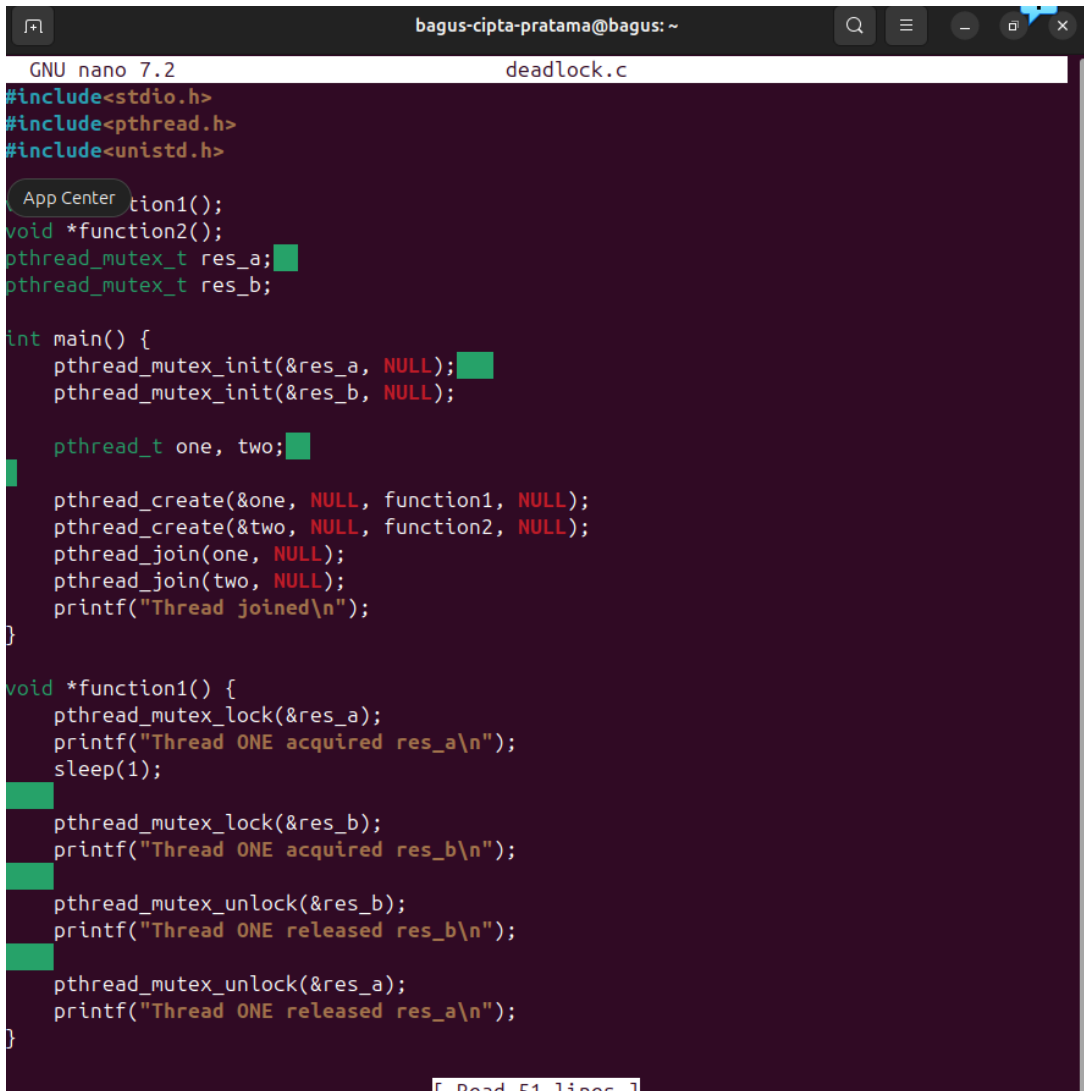
sama lain . yang kedua adalah mengamati efek tidak menggunakan fungsi `pthread_join` .

Tanpa `pthread_join` , program utama dan child thread berjalan secara independen. Program utama tidak akan menunggu hingga thread selesai menjalankan fungsinya. Ini menyebabkan hasil yang terlihat campur aduk (interleaved) antara output dari thread utama dan thread baru, karena keduanya berjalan bersamaan tanpa kontrol siapa yang harus selesai duluan.

Penjelasan output :

Program utama dimulai dengan mencetak "Di dalam Program Utama" dan angka 4, 3, 2, 1, 0. Sementara itu, *child thread* mencetak "Di dalam Thread" diikuti dengan angka 0, 1, 2, 3, 4. Karena tidak ada `pthread_join` untuk menunggu, kedua thread ini berjalan secara bersamaan, sehingga hasil cetakannya muncul bergantian (interleaved). Ini menyebabkan angka dari program utama dan *child thread* tercampur dalam output. Selanjutnya , dapat dilihat bahwa angka dari thread utama dan *child thread* muncul tidak berurutan, misalnya program utama mencetak 4, lalu *child thread* mencetak 0, dan seterusnya. Hal ini disebabkan oleh fakta bahwa kedua thread bekerja pada waktu yang sama dan sistem operasi membagi waktu proses antara kedua thread, bergantung pada jadwal thread yang diatur oleh CPU . Karena kedua thread tidak saling menunggu, sistem operasi secara dinamis mengatur eksekusi mereka, yang menghasilkan hasil yang campur aduk.

2. Activity 7.2 : menjalankan program dengan passing nilai ke thread



```
GNU nano 7.2 deadlock.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(&res_a, NULL);
    pthread_mutex_init(&res_b, NULL);

    pthread_t one, two;

    pthread_create(&one, NULL, function1, NULL);
    pthread_create(&two, NULL, function2, NULL);
    pthread_join(one, NULL);
    pthread_join(two, NULL);
    printf("Thread joined\n");
}

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

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

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

    pthread_mutex_unlock(&res_a);
    printf("Thread ONE released res_a\n");
}
```

[Read 51 lines]

```
GNU nano 7.2                                deadlock.c
pthread_create(&one, NULL, function1, NULL);
pthread_create(&two, NULL, function2, NULL);
pthread_join(one, NULL);
pthread_join(two, NULL);
printf("Thread joined\n");
}

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

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

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

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

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

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

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

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

```
bagus-cipta-pratama@bagus: ~
bagus-cipta-pratama@bagus:~$ nano deadlock.c
bagus-cipta-pratama@bagus:~$ gcc -o deadlock.out deadlock.c -lpthread
bagus-cipta-pratama@bagus:~$ ./deadlock.out
Thread TWO acquired res_a
Thread TWO acquired res_b
Thread TWO released res_b
Thread TWO released res_a
Thread ONE acquired res_a
Thread ONE acquired res_b
Thread ONE released res_b
Thread ONE released res_a
Thread joined
bagus-cipta-pratama@bagus:~$
```

Dalam activity 7.2 ini kita akan mendemonstrasikan proses deadlock . deadlock adalah situasi dalam pemrograman multithreading Dimana dua atau lebih thread saling

menunggu satu sama lain untuk melepaskan resource yang mereka butuhkan , sehingga tidak ada yang bisa melanjutkan eksekusi . dalam kasus ini , ada dua thread yang mencoba mengakses dua resource yang dikunci menggunakan mutex locks (res_a dan res_b).

Sebelum melakukan modifikasi pada program , program asli memiliki dua thread yang mengunci resource dalam urutan yang berbeda . inilah urutan pengunciannya

Thread 1 : mengunci res_a -> mengunci res_b -> melepaskan res_b -> melepaskan res_a

Thread 2 : mengunci res_b -> mengunci res_a -> melepaskan res_a -> melepaskan res_b

Dengan urutan ini , jika Thread 1 mengunci res_a dan Thread 2 mengunci res_b pada waktu yang hampir bersamaan, kedua thread akan saling menunggu untuk resource lain yang sudah dikunci oleh thread yang lain. Hal ini menciptakan deadlock, di mana kedua thread tidak dapat melanjutkan eksekusi.

Setelah itu saya memodifikasi thread 2 sehingga sama dengan thread 1 .

Apa yang terjadi setelah perubahan ?

Setelah modifikasi, tidak ada lagi potensi deadlock. Alasannya adalah karena kedua thread sekarang mencoba mengunci resource dalam urutan yang sama (res_a diikuti oleh res_b). Salah satu thread akan memperoleh akses ke kedua resource terlebih dahulu, sementara thread lainnya akan menunggu hingga resource tersebut dilepaskan. Begitu resource dilepaskan, thread kedua bisa mengunci resource dan melanjutkan eksekusinya.

Penjelasan output :

1. Output dimulai dengan thread dua mendapatkan res_a terlebih dahulu .

Ini terjadi karena thread dua berhasil lebih dahulu mendapatkan res_a . pada saat ini , thread satu belum mendapatkan kesempatan untuk mengunci res_a

2. Thread dua mendapatkan res_b setelah res _a .

Setelah thread dua berhasil mengunci res_a, ia kemudian menunggu sejenak (sesuai sleep(1)) dan berhasil mengunci res_b. Karena tidak ada persaingan dari thread satu pada saat itu, thread dua bisa langsung mengunci kedua resource (res_a dan res_b).

3. Thread dua melepaskan res_b dan res _a

Ini menunjukkan bahwa thread dua selesai bekerja dengan kedua resource dan sekarang telah membebaskan resource tersebut sehingga bisa digunakan oleh thread lain.

4. Selanjutnya thread satu melakukan hal yang sama
5. Terakhir ada thread joined .

Ini berarti kedua thread telah selesai dan sudah bergabung kembali ke thread utama (*main thread*). Eksekusi program pun selesai.

Penjelasan mengapa output seperti ini :

- A. Tidak terjadi deadlock , Karena kedua thread mengikuti urutan yang sama dalam mengunci resource, tidak ada situasi di mana satu thread memegang satu resource dan menunggu resource lain yang sudah dikunci oleh thread lainnya. Hal ini mencegah deadlock terjadi.
- B. Thread berjalan bergantian , Walaupun ada kemungkinan kedua thread berjalan hampir bersamaan, resource yang diakses bersifat eksklusif (karena mutex locks). Ini menyebabkan thread yang kedua harus menunggu resource yang dikunci oleh thread pertama, sehingga mereka berjalan bergantian.
- C. Sleep selama satu detik memberi jeda antara proses penguncian resources di masing masing thread sehingga kita bisa melihat bahwa urutan dalam output terlihat jelas .

Thank You