OS-Lab

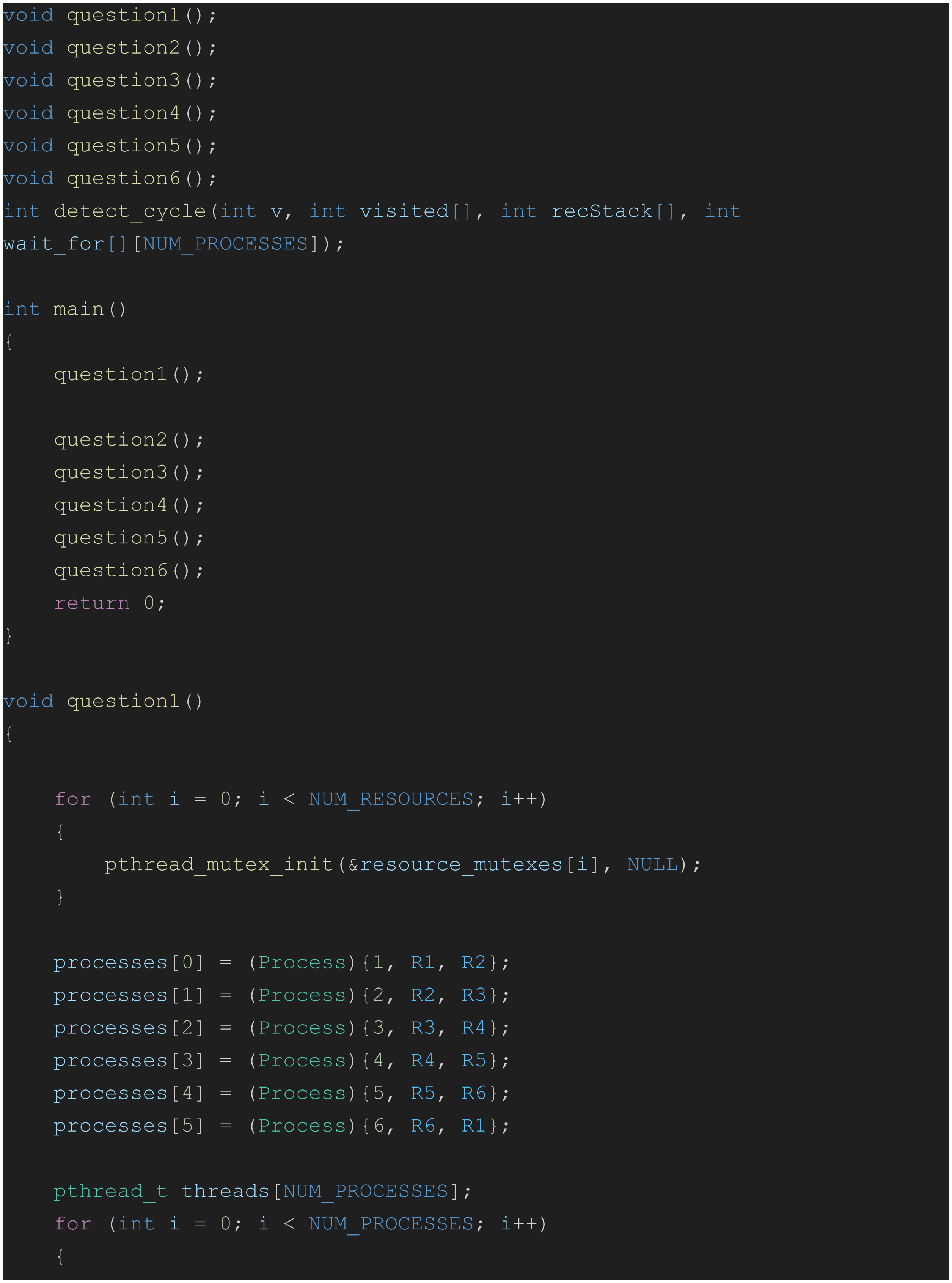
Lab#13

Name: Anas-Altaf

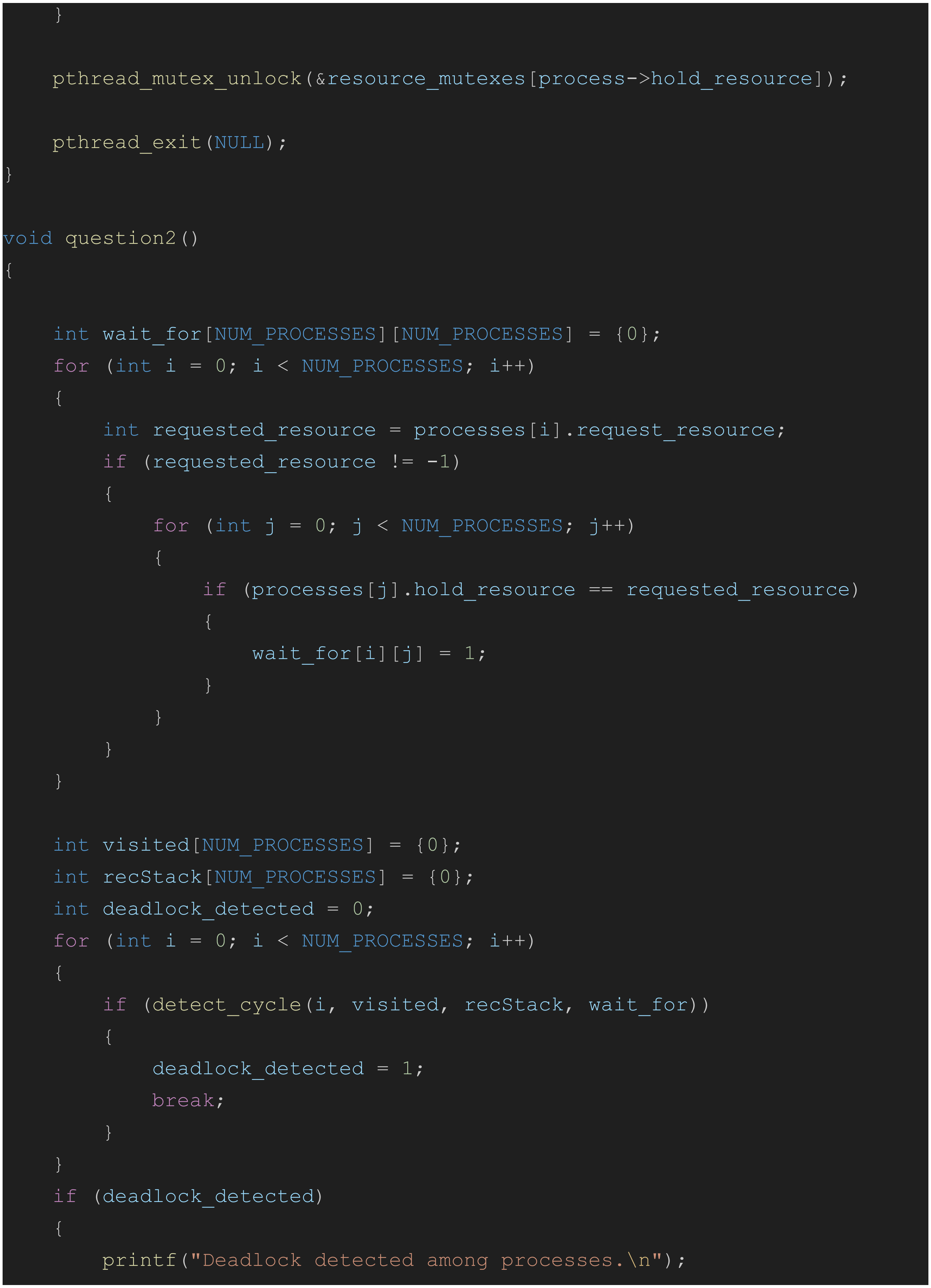
Roll.No: 22F-3639

Task-1 : Deadlock Simulation

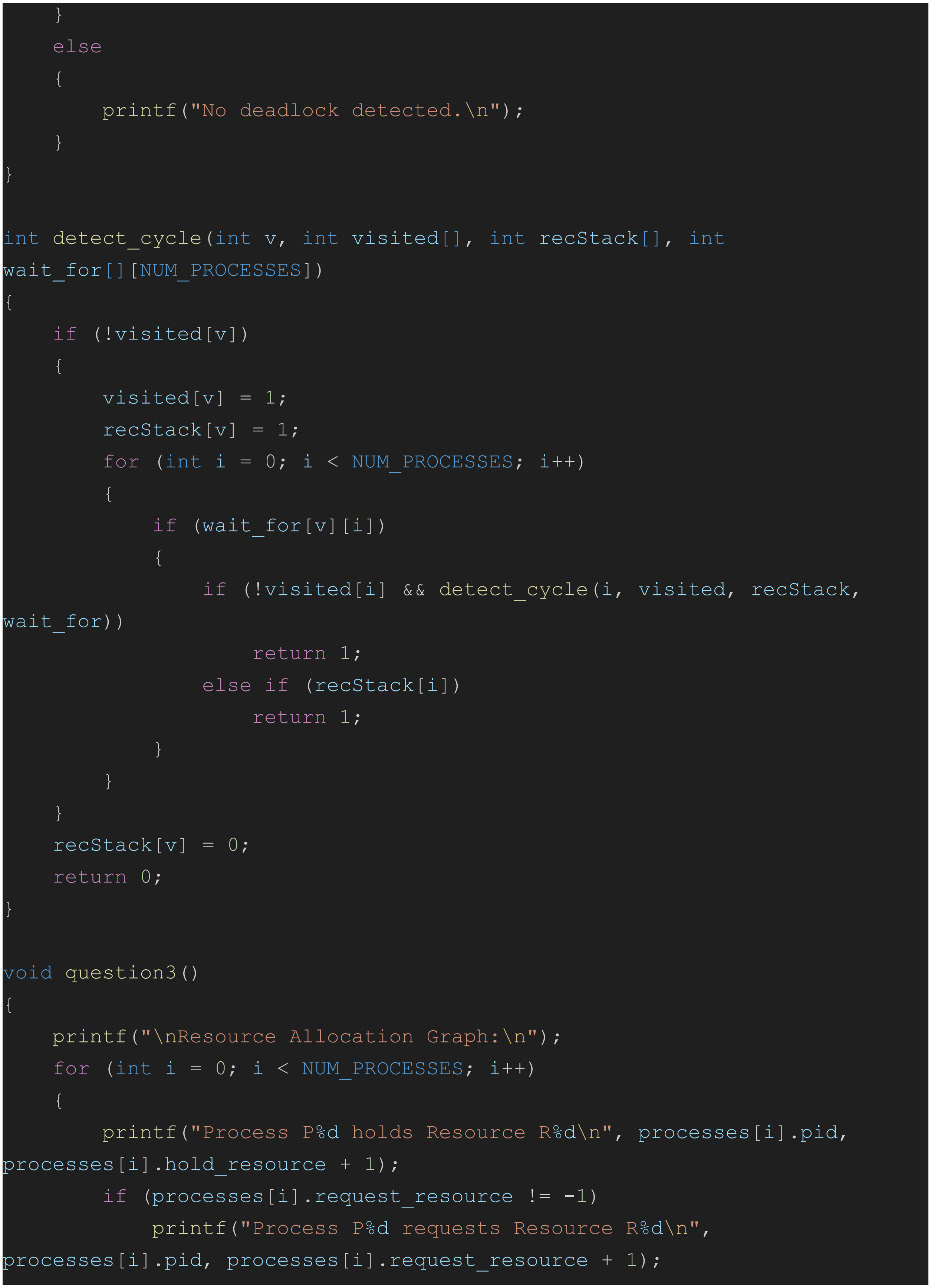
```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_PROCESSES 6
#define NUM_RESOURCES 6

enum
{
    R1,
    R2,
    R3,
    R4,
    R5,
    R6
};

pthread_mutex_t resource_mutexes[NUM_RESOURCES];

typedef struct
{
    int pid;
    int hold_resource;
    int request_resource;
} Process;

Process processes[NUM_PROCESSES];

void *process_function(void *arg);
```



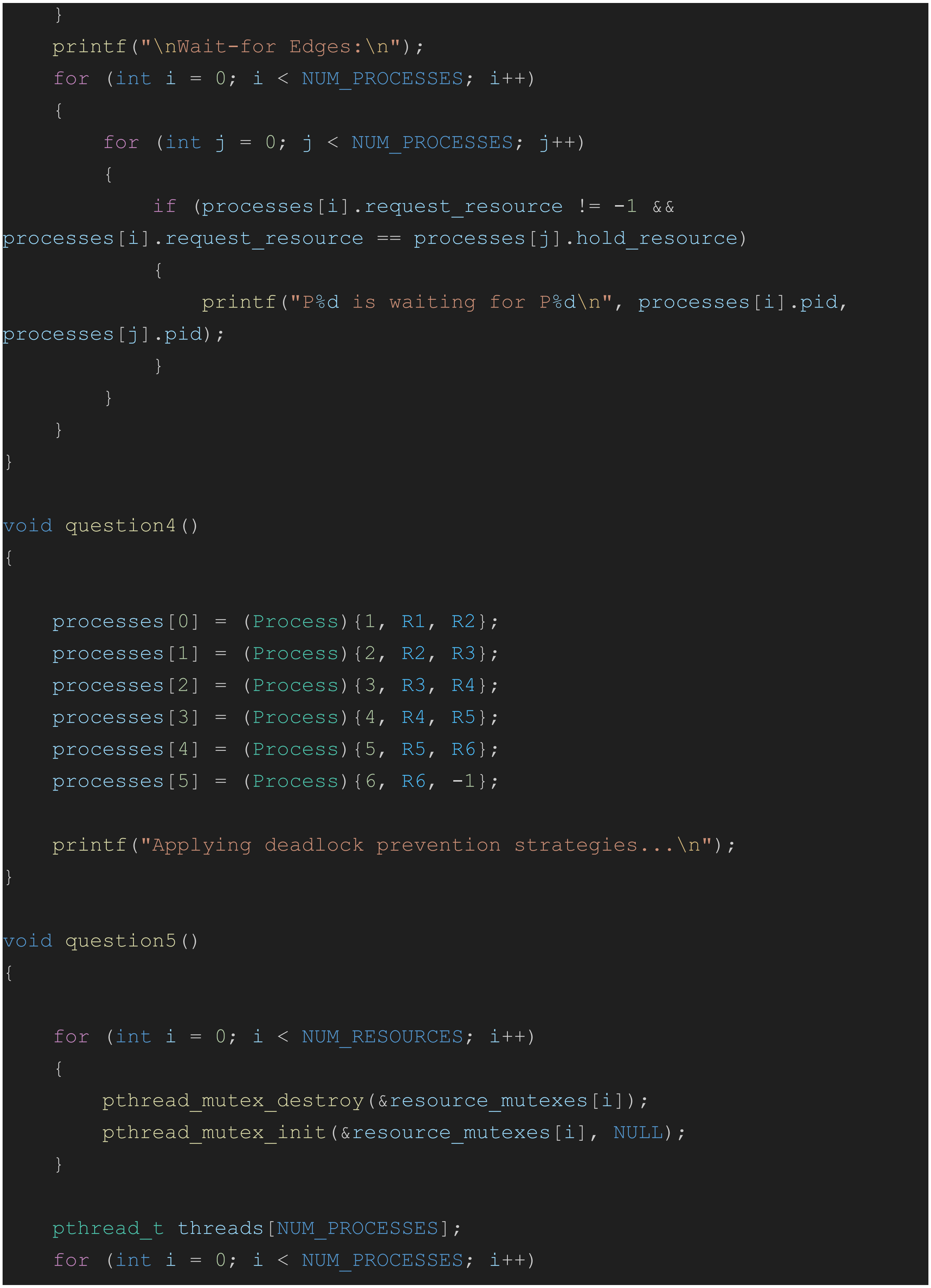
```
void question1();
void question2();
void question3();
void question4();
void question5();
void question6();
int detect_cycle(int v, int visited[], int recStack[], int
wait_for[][NUM_PROCESSES]);

int main()
{
    question1();

    question2();
    question3();
    question4();
    question5();
    question6();
    return 0;
}

void question1()
{

    for (int i = 0; i < NUM_RESOURCES; i++)
    {
        pthread_mutex_init(&resource_mutexes[i], NULL);
    }

    processes[0] = (Process){1, R1, R2};
    processes[1] = (Process){2, R2, R3};
    processes[2] = (Process){3, R3, R4};
    processes[3] = (Process){4, R4, R5};
    processes[4] = (Process){5, R5, R6};
    processes[5] = (Process){6, R6, R1};

    pthread_t threads[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        pthread_create(&threads[i], NULL, process_function, (void
*)&processes[i]);
        sleep(1);
    }

    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        pthread_join(threads[i], NULL);
    }
}

void *process_function(void *arg)
{
    Process *process = (Process *)arg;

    pthread_mutex_lock(&resource_mutexes[process->hold_resource]);
    printf("Process P%d holds R%d\n", process->pid, process->hold_resource
+ 1);

    sleep(1);

    if (process->request_resource != -1)
    {

        if (process->hold_resource < process->request_resource)
        {
            printf("Process P%d requests R%d\n", process->pid,
process->request_resource + 1);

pthread_mutex_lock(&resource_mutexes[process->request_resource]);
            printf("Process P%d acquired R%d\n", process->pid,
process->request_resource + 1);

pthread_mutex_unlock(&resource_mutexes[process->request_resource]);
        }
        else
        {
            printf("Process P%d cannot request R%d due to resource
ordering\n", process->pid, process->request_resource + 1);
        }
```



```
    }

    pthread_mutex_unlock(&resource_mutexes[process->hold_resource]);

    pthread_exit(NULL);
}

void question2()
{

    int wait_for[NUM_PROCESSES][NUM_PROCESSES] = {0};
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        int requested_resource = processes[i].request_resource;
        if (requested_resource != -1)
        {
            for (int j = 0; j < NUM_PROCESSES; j++)
            {
                if (processes[j].hold_resource == requested_resource)
                {
                    wait_for[i][j] = 1;
                }
            }
        }
    }

    int visited[NUM_PROCESSES] = {0};
    int recStack[NUM_PROCESSES] = {0};
    int deadlock_detected = 0;
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        if (detect_cycle(i, visited, recStack, wait_for))
        {
            deadlock_detected = 1;
            break;
        }
    }
    if (deadlock_detected)
    {
        printf("Deadlock detected among processes.\n");
```



```c
    }
    else
    {
        printf("No deadlock detected.\n");
    }
}

int detect_cycle(int v, int visited[], int recStack[], int
wait_for[][NUM_PROCESSES])
{
    if (!visited[v])
    {
        visited[v] = 1;
        recStack[v] = 1;
        for (int i = 0; i < NUM_PROCESSES; i++)
        {
            if (wait_for[v][i])
            {
                if (!visited[i] && detect_cycle(i, visited, recStack,
wait_for))
                    return 1;
                else if (recStack[i])
                    return 1;
            }
        }
    }
    recStack[v] = 0;
    return 0;
}

void question3()
{
    printf("\nResource Allocation Graph:\n");
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        printf("Process P%d holds Resource R%d\n", processes[i].pid,
processes[i].hold_resource + 1);
        if (processes[i].request_resource != -1)
            printf("Process P%d requests Resource R%d\n",
processes[i].pid, processes[i].request_resource + 1);
```



```
    }
    printf("\nWait-for Edges:\n");
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        for (int j = 0; j < NUM_PROCESSES; j++)
        {
            if (processes[i].request_resource != -1 &&
processes[i].request_resource == processes[j].hold_resource)
            {
                printf("P%d is waiting for P%d\n", processes[i].pid,
processes[j].pid);
            }
        }
    }
}

void question4()
{

    processes[0] = (Process){1, R1, R2};
    processes[1] = (Process){2, R2, R3};
    processes[2] = (Process){3, R3, R4};
    processes[3] = (Process){4, R4, R5};
    processes[4] = (Process){5, R5, R6};
    processes[5] = (Process){6, R6, -1};

    printf("Applying deadlock prevention strategies...\n");
}

void question5()
{

    for (int i = 0; i < NUM_RESOURCES; i++)
    {
        pthread_mutex_destroy(&resource_mutexes[i]);
        pthread_mutex_init(&resource_mutexes[i], NULL);
    }

    pthread_t threads[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        pthread_create(&threads[i], NULL, process_function, (void
*)&processes[i]);
        sleep(1);
    }

    for (int i = 0; i < NUM_PROCESSES; i++)
    {
        pthread_join(threads[i], NULL);
    }

    printf("System state after applying prevention strategies:\n");
    question2();
}

void question6()
{
    printf("Applying deadlock resolution strategies...\n");
    printf("Terminating Process P6 to break the deadlock.\n");

    processes[5].pid = 0;
    processes[5].hold_resource = -1;
    processes[5].request_resource = -1;

    for (int i = 0; i < NUM_RESOURCES; i++)
    {
        pthread_mutex_destroy(&resource_mutexes[i]);
        pthread_mutex_init(&resource_mutexes[i], NULL);
    }

    pthread_t threads[NUM_PROCESSES - 1];
    for (int i = 0; i < NUM_PROCESSES - 1; i++)
    {
        pthread_create(&threads[i], NULL, process_function, (void
*)&processes[i]);
        sleep(1);
    }

    for (int i = 0; i < NUM_PROCESSES - 1; i++)
    {
        pthread_join(threads[i], NULL);
    }

    printf("System state after applying resolution strategies:\n");
    question2();
}
```

Output:

```
PS F:\University Tasks\FAST-BSE-5B\OS Lab\Lab_13> .\deadlock_simulation.exe
Process P1 holds R1
Process P2 holds R2
Process P1 requests R2
Process P3 holds R3
Process P2 requests R3
Process P4 holds R4
Process P3 requests R4
Process P5 holds R5
Process P4 requests R5
Process P6 holds R6
Process P5 requests R6
Process P6 cannot request R1 due to resource ordering
Process P5 acquired R6
Process P4 acquired R5
Process P3 acquired R4
Process P2 acquired R3
Process P1 acquired R2
Deadlock detected among processes.

Resource Allocation Graph:
Process P1 holds Resource R1
Process P1 requests Resource R2
Process P2 holds Resource R2
Process P2 requests Resource R3
Process P3 holds Resource R3
Process P3 requests Resource R4
Process P4 holds Resource R4
Process P4 requests Resource R5
Process P5 holds Resource R5
Process P5 requests Resource R6
Process P6 holds Resource R6
Process P6 requests Resource R1
```

```
Wait-for Edges:
P1 is waiting for P2
P2 is waiting for P3
P3 is waiting for P4
P4 is waiting for P5
P5 is waiting for P6
P6 is waiting for P1
Applying deadlock prevention strategies...
Process P1 holds R1
Process P2 holds R2
Process P1 requests R2
Process P3 holds R3
Process P2 requests R3
Process P3 requests R4
Process P3 acquired R4
Process P2 acquired R3
Process P4 holds R4
Process P1 acquired R2
Process P5 holds R5
Process P4 requests R5
Process P6 holds R6
Process P5 requests R6
Process P5 acquired R6
Process P4 acquired R5
System state after applying prevention strategies:
No deadlock detected.
Applying deadlock resolution strategies...
Terminating Process P6 to break the deadlock.
Process P1 holds R1
Process P2 holds R2
Process P1 requests R2
Process P3 holds R3
Process P2 requests R3
Process P3 requests R4
Process P3 acquired R4
Process P2 acquired R3
Process P4 holds R4
Process P1 acquired R2
Process P5 holds R5
Process P4 requests R5
Process P5 requests R6
Process P5 acquired R6
Process P4 acquired R5
```

System state after applying resolution strategies:
No deadlock detected.