과목:	AI기반시스템프로그래밍
교수님:	최창희 교수님
학과:	정보보호학과
학번:	20011705
이름:	이지섭

제목 :
Assignment_#4_Deadlock_Hunters

목차

I. Wait-for Graph 구현

Wait-for graph는 다음과 같이 **인접행렬로 구현**되었고 **전역**으로 선언하였습니다. 그리고 인접행렬에 **간선을 추가**하는 `add_edge` 함수도 구현되어 있습니다.

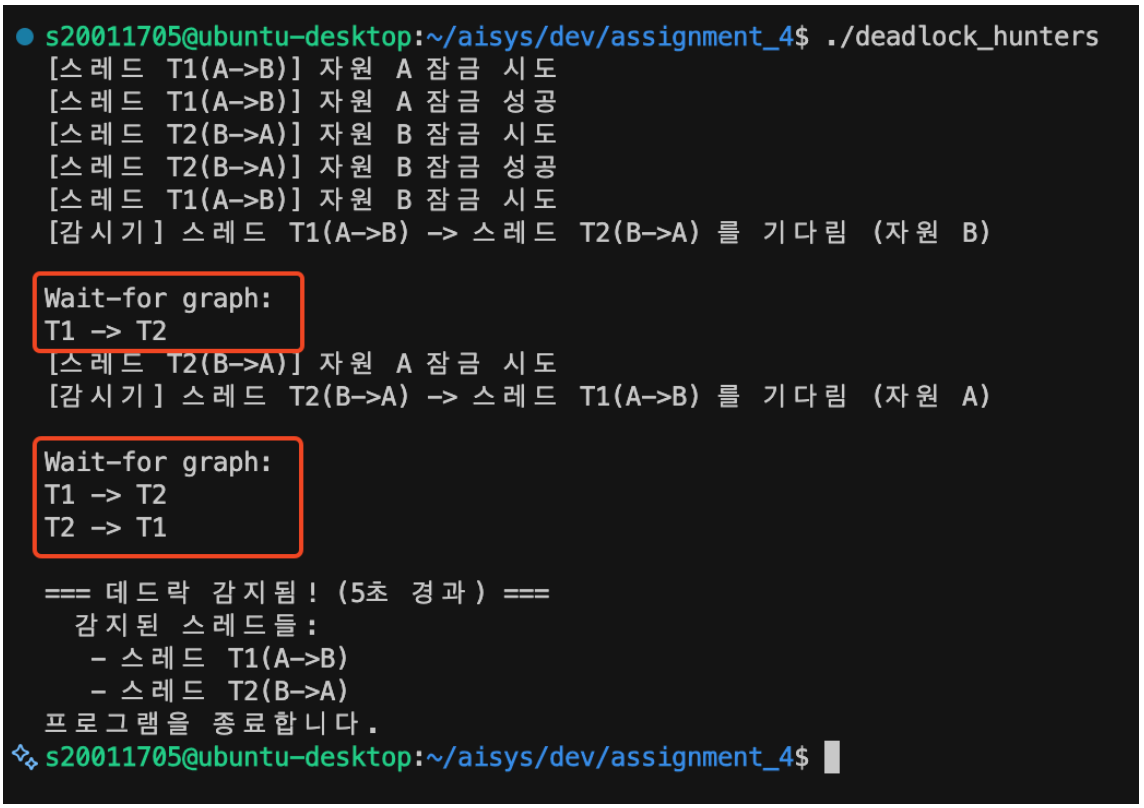
```
int wait_for_graph[NUM_THREADS][NUM_THREADS];

void add_edge(int from, int to) { wait_for_graph[from][to] = 1; }
```

스레드가 다른 스레드를 기다릴 때 `add_edge` 함수를 사용하여 해당 행렬에 1을 설정합니다.

그리고 런타임에 Wait-for graph 현황을 확인하기 위해 다음과 같이 `print_graph` 함수를 추가하였습니다. (과제 제출시 주석 처리함.)

```
// Print current wait-for graph
void print_graph(void) {
    printf("\nWait-for graph:\n");
    bool has_edges = false;
    for (int i = 0; i < MAX_THREAD; i++) {
        for (int j = 0; j < MAX_THREAD; j++) {
            if (wait_for_graph[i][j]) {
                printf("T%d -> T%d\n", i + 1, j + 1);
                has_edges = true;
            }
        }
    }
    if (!has_edges) printf("  (no edges)\n");
}
```



위 이미지와 같이 스레드간 대기 상태가 잘 저장된 것을 확인할 수 있습니다.

II. DFS 기반 사이클 탐지

```
// visited: node was ever visited; in_stack: node is on current recursion
stack.
static bool dfs_visit_bool(int node, bool visited[], bool in_stack[]) {
    visited[node] = true;
    in_stack[node] = true;

    for (int next = 0; next < MAX_THREAD; next++) {
        if (!wait_for_graph[node][next]) continue;

        if (!visited[next]) {
            if (dfs_visit_bool(next, visited, in_stack)) return true;
        } else if (in_stack[next]) {
            // back-edge -> cycle; record nodes in current stack and build
lines
            dstate.count = 0;
            for (int i = 0; i < MAX_THREAD; i++) {
                if (in_stack[i]) {
                    /* include newline so each entry prints on its own line
*/
                    snprintf(dstate.lines[dstate.count], MAX_STR_LEN,
                             "   - %s\n", threads_info[i].name);
                    dstate.count++;
                }
```

```
            }
            return true;
        }
    }

    in_stack[node] = false;
    return false;
}

bool dfs_cycle(void) {
    bool visited[MAX_THREAD];
    bool in_stack[MAX_THREAD];

    for (int i = 0; i < MAX_THREAD; i++) {
        visited[i] = false;
        in_stack[i] = false;
    }

    for (int i = 0; i < MAX_THREAD; i++) {
        if (!visited[i]) {
            if (dfs_visit_bool(i, visited, in_stack)) return true;
        }
    }
    return false;
}
```

위와 같이 `dfs_cycle`과 `dfs_visit_bool` 함수를 작성하였습니다. 실질적으로 dfs 알고리즘을 수행하는 함수는 `dfs_cycle` 입니다.
인자로 `node`를 받는데, 이는 dfs 알고리즘에서 시작 노드를 의미합니다.

`dfs_visit_bool` 함수의 주요 코드별로 설명하겠습니다.

```
if (!wait_for_graph[node][next]) continue;
```

두 노드간에 간선이 없다면 넘어갑니다.

```
if (!visited[next]) {
      if (dfs_visit_bool(next, visited, in_stack)) return true;
}
```

방문하지 않았던 인접 노드라면 재귀적으로 `dfs_visit_bool`을 호출해 방문 처리합니다.

```
else if (in_stack[next]) {
            // back-edge -> cycle; record nodes in current stack and build
lines
            dstate.count = 0;
            for (int i = 0; i < MAX_THREAD; i++) {
                if (in_stack[i]) {
                    /* include newline so each entry prints on its own line
```

```
*/
                    snprintf(dstate.lines[dstate.count], MAX_STR_LEN,
                             "    - %s\n", threads_info[i].name);
                    dstate.count++;
                }
            }
            return true;
        }
```

in_stack 배열은 현재 **재귀 경로에 있는 노드를 저장**하고 있습니다. 따라서 현재 탐색 중인 경로 상의 노드가 이미 in_stack 배열에 있었다면 **사이클로 인해 다시 되돌아온 것**입니다. 즉, **사이클(데드락)이 발생**했다는 뜻입니다.
추후 데드락 정보를 출력하기 위해서 관련된 정보를 dstate 구조체 변수에 저장합니다.

```
in_stack[node] = false;
```

인접 노드들을 모두 순회하고 나서 사이클이 없었다면, 현재 시작 노드인 node를 in_stack 배열에서 해제합니다. (false 처리)

```
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$ ./deadlock_hunters
[스레드 T1(A->B)] 자원 A 잠금 시도
[스레드 T1(A->B)] 자원 A 잠금 성공
[스레드 T2(B->A)] 자원 B 잠금 시도
[스레드 T2(B->A)] 자원 B 잠금 성공
[스레드 T1(A->B)] 자원 B 잠금 시도
[감시기] 스레드 T1(A->B) -> 스레드 T2(B->A) 를 기다림 (자원 B)
[스레드 T2(B->A)] 자원 A 잠금 시도
[감시기] 스레드 T2(B->A) -> 스레드 T1(A->B) 를 기다림 (자원 A)

=== 데드락 감지됨! (5초 경과) ===
  감지된 스레드들:
   - 스레드 T1(A->B)
   - 스레드 T2(B->A)
프로그램을 종료합니다.
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$
```

위 이미지와 같이 데드락 감지시 감지된 스레드 목록들이 정상적으로 출력되는 것을 확인할 수 있습니다. 이는 위 dfs_visit_bool 함수에서 데드락 스레드들을 dstate 변수에 정상적으로 저장한 결과입니다.

III. 데드락 재현

```
// 스레드 정보 구조체
typedef struct {
    int tid;                    // 0 → T1, 1 → T2
    int res_count;              // 사용하려는 자원 개수
    int res_order[MAX_RESOURCE]; // 잠금 순서 (예: {0,1} → A,B)
    char name[MAX_NAME_LEN];    // 스레드 이름 캐싱: 스레드 T1(A->B)
} thread_info;
thread_info threads_info[MAX_THREAD];
```

위와 같이 스레드 정보를 저장하는 구조체를 정의하여 `threads_info` 배열을 전역으로 선언하였습니다.

```
int main(void) {
    . . .

    // thread locking patterns (전역 tinfo에 설정)
    threads_info[0].tid = 0;
    threads_info[0].res_count = 2;
    threads_info[0].res_order[0] = 0;  // A
    threads_info[0].res_order[1] = 1;  // B

    threads_info[1].tid = 1;
    threads_info[1].res_count = 2;
    threads_info[1].res_order[0] = 1;  // B
    threads_info[1].res_order[1] = 0;  // A

    // 스레드 이름 먼저 생성
    for (int i = 0; i < MAX_THREAD; i++) {
        build_thread_name(&threads_info[i]);
    }

    // 스레드 실행
    for (int i = 0; i < MAX_THREAD; i++) {
        pthread_create(&threads[i], NULL, worker_thread, &threads_info[i]);
    }
    . . .

    return 0;
}
```

그리고 `main` 함수 내에서 각 스레드별 정보를 저장합니다. 위 코드에서 볼 수 있듯이, 첫 번째 스레드에서는 A->B 순서로 자원을 잠그고, 두 번째 스레드에서는 B->A 순서로 자원을 잠급니다. 그리고 첫 번째 스레드부터 순서대로 실행합니다.

```
// 스레드 실행 함수
void* worker_thread(void* arg) {
    thread_info* info = (thread_info*)arg;

    lock_resources(info);
    release_resources(info);

    return NULL;
}
```

스레드 실행 함수입니다. 자원들을 잠그고 해제하는 역할을 합니다.

```
void lock_resources(thread_info* info) {
    int tid = info->tid;
```

```
    for (int i = 0; i < info->res_count; i++) {
        int target_resource = info->res_order[i];
        printf("[%s] 자원 %c 잠금 시도\n", info->name, 'A' +
target_resource);

        if (is_locked_by_other(&resource_locks[target_resource])) {
            int owner = get_resource_owner(target_resource);
            if (owner < 0) {
                fprintf(stderr,
                        "error: get_owner returned -1 for resource %c\n",
                        'A' + target_resource);
                exit(1);
            }

            printf("[감시기] %s -> %s 를 기다림 (자원 %c)\n", info->name,
                   threads_info[owner].name, 'A' + target_resource);

            // wait-for edge 추가; dfs_cycle은 add_edge 실행 이후 호출
            add_edge(tid, owner);
            // print_graph();
            if (dfs_cycle()) {
                start_deadlock_alarm();
            }
        }

        pthread_mutex_lock(&resource_locks[target_resource]);
        thread_owns[tid][target_resource] = true;
        printf("[%s] 자원 %c 잠금 성공\n", info->name, 'A' +
target_resource);

        sleep(1);
    }
}
```

자원을 잠그는 함수입니다. 스레드 정보에 저장되어 있었던 **자원 잠금 순서**를 참조하여 잠급니다.
먼저 다른 스레드에 의해 자원이 잠겼는지 확인합니다.
이미 잠겨있다면 먼저 잠그고 있던 스레드에 대해 Wait-for graph **간선을 추가**합니다. 그리고 매번 간선을 추가할 때마다 dfs 알고리즘을 수행하여 사이클(데드락)이 존재하는지 확인합니다. 만약 존재한다면, 데드락 알람 함수인 `start_deadlock_alarm`을 호출합니다.
그리고 `pthread_mutex_lock` 함수를 호출합니다. 만약 이미 자원을 잠그고 있던 상황이었다면 대기상태에 빠지게 될 것입니다. 잠그고 있는 스레드가 없었다면 자원을 잠그는 데 성공하게 됩니다.

그리고 마지막으로 `sleep(1);` 함수를 호출하여 1초 대기하게 됩니다.
따라서 일반적으로는 다음과 같은 순서로 프로그램이 진행될 것입니다.

1. 스레드 T1이 자원 A를 잠금
2. 스레드 T1 sleep
3. 스레드 T2가 자원 B를 잠금
4. 스레드 T2 sleep

5. 스레드 T1이 자원 B를 잠그려고 했으나 이미 스레드 T2가 소유 중. 따라서 T2를 기다리게 됩니다.
6. 스레드 T2가 자원 A를 잠그려고 했으나 이미 스레드 T1이 소유 중. 따라서 T1을 기다리게 됩니다.
7. `dfs_cycle`에 의해 사이클이 발견됩니다.
8. 5초 후 데드락 감지 로그가 출력됩니다.

그러나 스레드 실행 순서는 무조건 예측할 수 있는 것이 아니기 때문에 **가끔씩 실행 순서가 달라질 때도 있습니다.**

```
static void alarm_handler(int signo) {
    const char deadlock_alert_intro_message[] =
        "\n=== 데드락 감지됨! (5초 경과) ===\n  감지된 스레드들:\n";
    const char deadlock_alert_outro_message[] = "프로그램을 종료합니다.\n";

    bool confirmed_deadlock = dfs_cycle();
    if (confirmed_deadlock) {
        write(STDOUT_FILENO, deadlock_alert_intro_message,
              sizeof(deadlock_alert_intro_message) - 1);

        for (int i = 0; i < dstate.count; i++) {
            const char* p = dstate.lines[i];
            size_t len = strlen(p);
            if (len > 0) {
                write(STDOUT_FILENO, p, len);
            }
        }
        write(STDOUT_FILENO, deadlock_alert_outro_message,
              sizeof(deadlock_alert_outro_message) - 1);
        _exit(0);
    }
    // deadlock이 아니면 프로그램 계속 실행
}

void start_deadlock_alarm(void) {
    // 여러 스레드가 동시에 호출할 수 있으므로 한 번만 실행되도록 함
    if (dstate.alarm_started) return;

    dstate.alarm_started = true;
    signal(SIGALRM, alarm_handler);
    alarm(5);
}
```

데드락 알람은 위와 같이 **시그널을 통해 구현**된 것을 확인할 수 있습니다.

```
signal(SIGALRM, alarm_handler);
alarm(5);
```

`signal` 함수를 통해 `SIGALRM` 시그널에 대해 시그널 핸들러 함수를 등록합니다. `alarm(5)` 코드에 의해 5초 후 `SIGALRM` 시그널이 발생합니다.

이후 `alarm_handler` 함수가 호출되고, **마지막으로 데드락 여부를 검사**하여 데드락 스레드들을 출력하게 됩니다.

```
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$ ./deadlock_hunters
[스레드 T1(A->B)] 자원 A 잠금 시도
[스레드 T1(A->B)] 자원 A 잠금 성공
[스레드 T2(B->A)] 자원 B 잠금 시도
[스레드 T2(B->A)] 자원 B 잠금 성공
[스레드 T1(A->B)] 자원 B 잠금 시도
[감시기] 스레드 T1(A->B) -> 스레드 T2(B->A) 를 기다림 (자원 B)
[스레드 T2(B->A)] 자원 A 잠금 시도
[감시기] 스레드 T2(B->A) -> 스레드 T1(A->B) 를 기다림 (자원 A)

=== 데드락 감지됨! (5초 경과) ===
  감지된 스레드들:
   - 스레드 T1(A->B)
   - 스레드 T2(B->A)
프로그램을 종료합니다.
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$
```

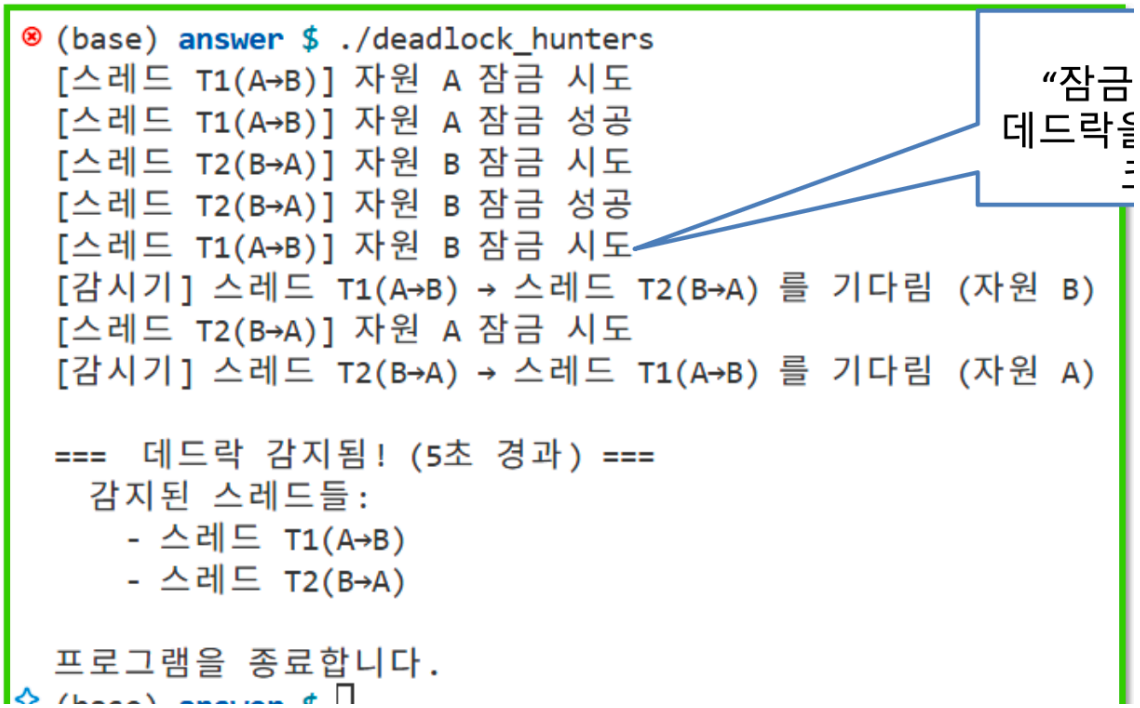
위 이미지와 같이 예상한 결과가 나왔습니다. T1과 T2 두 스레드 사이에 데드락이 발생하였습니다.

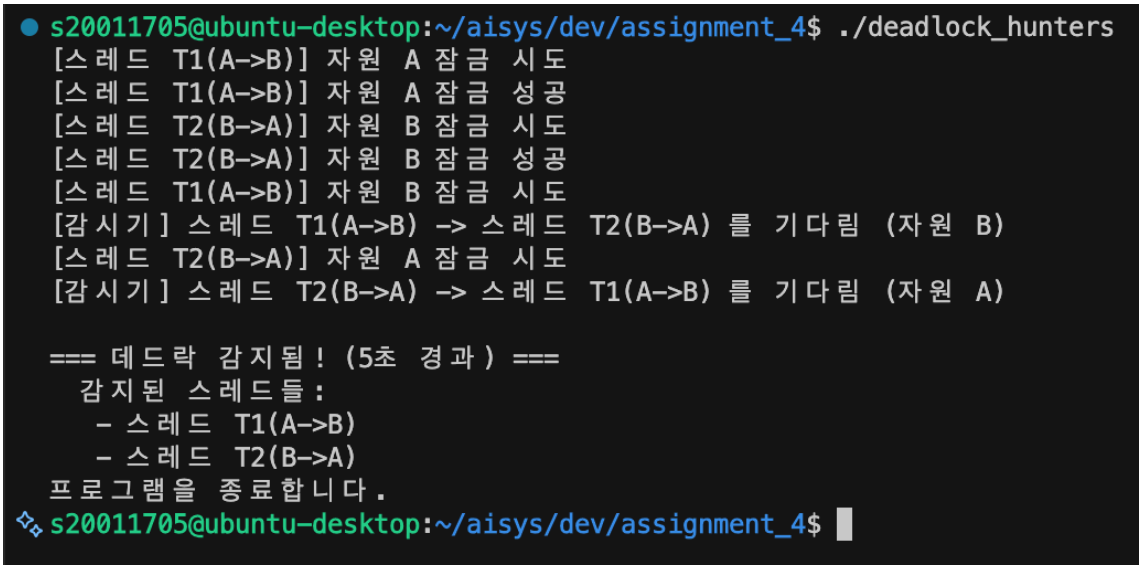
```
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$ ./deadlock_hunters
[스레드 T1(A->B)] 자원 A 잠금 시도
[스레드 T1(A->B)] 자원 A 잠금 성공
[스레드 T2(B->A)] 자원 B 잠금 시도
[스레드 T2(B->A)] 자원 B 잠금 성공
[스레드 T2(B->A)] 자원 A 잠금 시도
[감시기] 스레드 T2(B->A) -> 스레드 T1(A->B) 를 기다림 (자원 A)
[스레드 T1(A->B)] 자원 B 잠금 시도
[감시기] 스레드 T1(A->B) -> 스레드 T2(B->A) 를 기다림 (자원 B)

=== 데드락 감지됨! (5초 경과) ===
  감지된 스레드들:
   - 스레드 T1(A->B)
   - 스레드 T2(B->A)
프로그램을 종료합니다.
s20011705@ubuntu-desktop:~/aisys/dev/assignment_4$
```

가끔씩 위 이미지와 같이 순서가 달라지는 경우도 있습니다. 그러나 결국은 데드락까지 발생하여 정상적으로 감지되는 것을 확인할 수 있습니다.

IV. 출력 형식 준수 및 로그 메시지





위와 같이 제 프로그램의 출력문과 과제의 요구사항을 비교했을 때, **출력 형식이 준수**된 것을 확인할 수 있습니다.

V. Reference

- https://chatgpt.com/
- https://copilot.microsoft.com/
- https://gemini.google.com/
- https://www.geeksforgeeks.org/computer-networks/wait-for-graph-deadlock-detection-in-distributed-system/