Assignment4 : deadlock hunters

[AI기반시스템프로그래밍]

작성일 : 2025.11.25.

학번 : 23013393

작성자 : 최지혁

Assignment4 : deadlock hunters

> 이 과제는 **"Google Gemini Pro : 빠른 모드"**를 참고하여 작성하였습니다.

코드

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

// --- 상수 및 전역 변수 설정 ---

#define NUM_THREADS 2
#define T1_INDEX 0
#define T2_INDEX 1

// 데드락 감지 시간 관련 상수
#define CHECK_INTERVAL_MS 100 // 0.1초마다 검사
#define DEADLOCK_WAIT_TIME 5 // 총 5초 대기
```

```
#define DEADLOCK_THRESHOLD (DEADLOCK_WAIT_TIME * 1000 / CHECK
_INTERVAL_MS) // 5초 = 50회 검사

// 스레드 정보 구조체
typedef struct {
    int index;
    char *name;
    char *pattern;
} ThreadInfo;

// 자원(뮤텍스) 정보 구조체
typedef struct {
    pthread_mutex_t lock;
    int owner_index; // 현재 락을 보유한 스레드의 인덱스 (T1_INDEX, T2_INDEX
등, -1: 미보유)
    char *name; // 자원 이름 (A 또는 B)
} Resource;

// --- Wait-for Graph (WFG) 관련 전역 변수 ---
// wait_for_graph[i][j] = 1 : 스레드 i가 스레드 j가 보유한 자원을 기다림
int wait_for_graph[NUM_THREADS][NUM_THREADS] = {0};
pthread_mutex_t graph_mutex; // WFG 접근 보호를 위한 뮤텍스
pthread_mutex_t print_mutex; // 출력 보호를 위한 뮤텍스 (로그 메시지 보호)

// 자원 A와 B
Resource R[NUM_THREADS];
ThreadInfo Threads[NUM_THREADS];

// --- DFS를 위한 상태 정의 ---
#define WHITE 0
#define GRAY 1
#define BLACK 2

// --- 함수 선언 ---
void initialize_system();
void cleanup_system();
int dfs_cycle(int u, int color[]);
int check_deadlock();
```

```
void *monitor_thread_func(void *arg);
void *thread_func_t1(void *arg);
void *thread_func_t2(void *arg);
int custom_lock(Resource *res, int waiter_index);
void custom_unlock(Resource *res, int releaser_index);

// --- 1, 2번 항목: WFG 및 DFS 기반 사이클 감지 구현 ---

/**
 * @brief DFS를 사용하여 그래프에서 사이클을 탐지합니다.
 */
int dfs_cycle(int u, int color[]) {
    color[u] = GRAY;

    for (int v = 0; v < NUM_THREADS; v++) {
        if (wait_for_graph[u][v] == 1) { // u가 v를 기다림
            if (color[v] == GRAY) {
                return 1; // 사이클 발견 (Gray → Gray)
            }
            if (color[v] == WHITE) {
                if (dfs_cycle(v, color)) {
                    return 1;
                }
            }
        }
    }

    color[u] = BLACK;
    return 0;
}

/**
 * @brief 전체 WFG에 대해 데드락(사이클)이 있는지 확인합니다.
 */
int check_deadlock() {
    int color[NUM_THREADS];
    memset(color, WHITE, sizeof(color));
```

```
    pthread_mutex_lock(&graph_mutex);
    for (int i = 0; i < NUM_THREADS; i++) {
        if (color[i] == WHITE) {
            if (dfs_cycle(i, color)) {
                pthread_mutex_unlock(&graph_mutex);
                return 1; // 데드락 감지
            }
        }
    }
    pthread_mutex_unlock(&graph_mutex);
    return 0; // 데드락 없음
}


/**
 * @brief 락 요청을 처리하고 WFG에 간선을 추가합니다.
 */
int custom_lock(Resource *res, int waiter_index) {
    pthread_mutex_lock(&print_mutex);
    printf(" [스레드 %s] 자원 %s 잠금 시도\n",
           Threads[waiter_index].pattern, res->name);
    pthread_mutex_unlock(&print_mutex);

    // 락 시도
    if (pthread_mutex_trylock(&res->lock) == 0) {
        // 잠금 성공
        res->owner_index = waiter_index;

        pthread_mutex_lock(&print_mutex);
        printf(" [스레드 %s] 자원 %s 잠금 성공\n",
               Threads[waiter_index].pattern, res->name);
        pthread_mutex_unlock(&print_mutex);

        return 0;
    } else {
        // 잠금 실패 (자원이 점유 중)
        int owner_index = res->owner_index;
```

```
        // WFG에 간선 추가: waiter_index가 owner_index를 기다림
        pthread_mutex_lock(&graph_mutex);
        wait_for_graph[waiter_index][owner_index] = 1;
        pthread_mutex_unlock(&graph_mutex);

        // 4번 항목: 출력 형식 준수
        pthread_mutex_lock(&print_mutex);
        printf(" [감시기] 스레드 %s → 스레드 %s를 기다림 (자원 %s)\n",
            Threads[waiter_index].pattern, Threads[owner_index].pattern, res
->name);
        pthread_mutex_unlock(&print_mutex);

        // 실제 잠금을 기다림 (여기서 영원히 대기하게 됨)
        pthread_mutex_lock(&res->lock);

        // 잠금 성공 후 (데드락 해소 후 락 획득): WFG에서 간선 제거
        pthread_mutex_lock(&graph_mutex);
        wait_for_graph[waiter_index][owner_index] = 0;
        pthread_mutex_unlock(&graph_mutex);

        // 소유자 업데이트
        res->owner_index = waiter_index;

        return -1;
    }
}

/**
 * @brief 락 해제를 처리하고 소유자를 초기화합니다.
 */
void custom_unlock(Resource *res, int releaser_index) {
    // 락 해제 시에도 WFG 간선 제거 로직이 필요할 수 있지만,
    // 이 과제는 데드락 감지 후 종료가 목표이므로 단순화
    res->owner_index = -1;
    pthread_mutex_unlock(&res->lock);
}
```

```
// --- 수정된 2-2번 항목: 감시기 스레드 구현 (5초 동안 유지 확인) ---

/**
 * @brief 데드락 감시 스레드 함수 (5초 동안 데드락 유지 확인 후 종료)
 */
void *monitor_thread_func(void *arg) {
    int deadlock_counter = 0;

    while (1) {
        usleep(CHECK_INTERVAL_MS * 1000); // 0.1초 대기

        if (check_deadlock()) {
            deadlock_counter++;

            if (deadlock_counter >= DEADLOCK_THRESHOLD) {
                // 5초 동안(50회) 데드락이 해소되지 않고 유지됨

                // 4번 항목: 출력 형식 준수
                pthread_mutex_lock(&print_mutex);
                printf("\n === 데드락 감지됨! (%d초 경과) ===\n", DEADLOCK_WAIT
_TIME);
                printf(" 감지된 스레드들 :\n");
                printf(" - 스레드 %s\n", Threads[T1_INDEX].pattern);
                printf(" - 스레드 %s\n", Threads[T2_INDEX].pattern);
                printf("\n 프로그램을 종료합니다.\n");
                pthread_mutex_unlock(&print_mutex);

                cleanup_system();
                exit(0);
            }
        } else {
            // 데드락이 해소되었으므로 카운터 초기화
            deadlock_counter = 0;
        }
    }
    return NULL;
}
```

```
// --- 3번 항목: 데드락 재현 ---

/**
 * @brief 스레드 T1의 작업 함수 (A→B 순서)
 */
void *thread_func_t1(void *arg) {
    // 1. 자원 A 잠금 시도 (R[0])
    custom_lock(&R[0], T1_INDEX);

    // T2가 B 락을 성공할 시간을 부여하여 교착 상태를 유도
    usleep(500000);

    // 2. 자원 B 잠금 시도 (R[1]) - T2가 B를 가지고 있어 대기
    custom_lock(&R[1], T1_INDEX);

    // *이 코드는 실행되지 않음 (데드락 감지 후 종료)*
    custom_unlock(&R[1], T1_INDEX);
    custom_unlock(&R[0], T1_INDEX);
    return NULL;
}

/**
 * @brief 스레드 T2의 작업 함수 (B→A 순서)
 */
void *thread_func_t2(void *arg) {
    // 1. 자원 B 잠금 시도 (R[1])
    custom_lock(&R[1], T2_INDEX);

    // T1이 A 락을 성공할 시간을 부여하여 교착 상태를 유도
    usleep(500000);

    // 2. 자원 A 잠금 시도 (R[0]) - T1이 A를 가지고 있어 대기
    custom_lock(&R[0], T2_INDEX);

    // *이 코드는 실행되지 않음 (데드락 감지 후 종료)*
    custom_unlock(&R[0], T2_INDEX);
    custom_unlock(&R[1], T2_INDEX);
    return NULL;
```

```
}


// --- 초기화 및 실행 ---

void initialize_system() {
    // 1. 뮤텍스 및 자원 초기화
    pthread_mutex_init(&graph_mutex, NULL);
    pthread_mutex_init(&print_mutex, NULL);

    pthread_mutex_init(&R[0].lock, NULL);
    R[0].owner_index = -1;
    R[0].name = strdup("A");

    pthread_mutex_init(&R[1].lock, NULL);
    R[1].owner_index = -1;
    R[1].name = strdup("B");

    // 2. 스레드 정보 초기화
    Threads[T1_INDEX].index = T1_INDEX;
    Threads[T1_INDEX].name = strdup("T1");
    // LaTeX 출력을 위해 \rightarrow를 사용함
    Threads[T1_INDEX].pattern = strdup("T1(A→B)");

    Threads[T2_INDEX].index = T2_INDEX;
    Threads[T2_INDEX].name = strdup("T2");
    Threads[T2_INDEX].pattern = strdup("T2(B→A)");
}

void cleanup_system() {
    // 메모리 해제
    free(R[0].name);
    free(R[1].name);
    free(Threads[T1_INDEX].name);
    free(Threads[T1_INDEX].pattern);
    free(Threads[T2_INDEX].name);
    free(Threads[T2_INDEX].pattern);
```

```
    // 뮤텍스 파괴
    pthread_mutex_destroy(&R[0].lock);
    pthread_mutex_destroy(&R[1].lock);
    pthread_mutex_destroy(&graph_mutex);
    pthread_mutex_destroy(&print_mutex);
}

int main() {
    pthread_t tids[NUM_THREADS];
    pthread_t monitor_tid;

    initialize_system();

    // 1. 감시 스레드 시작
    if (pthread_create(&monitor_tid, NULL, monitor_thread_func, NULL) != 0)
{
        perror("Monitor thread creation failed");
        cleanup_system();
        return 1;
    }

    // 2. T1, T2 스레드 시작
    if (pthread_create(&tids[T1_INDEX], NULL, thread_func_t1, NULL) != 0) {
        perror("T1 thread creation failed");
        cleanup_system();
        return 1;
    }

    if (pthread_create(&tids[T2_INDEX], NULL, thread_func_t2, NULL) != 0) {
        perror("T2 thread creation failed");
        cleanup_system();
        return 1;
    }

    // 메인 스레드는 스레드가 종료되기를 기다림 (데드락 감지 후 monitor_thread가
exit(0)로 종료시킬 것임)
    pthread_join(tids[T1_INDEX], NULL);
    pthread_join(tids[T2_INDEX], NULL);
```

```
    pthread_join(monitor_tid, NULL);

    cleanup_system();
    return 0;
}
```

코드 구현

1. Wait-for Graph 구현

```
// --- Wait-for Graph (WFG) 관련 전역 변수 ---
// wait_for_graph[i][j] = 1 : 스레드 i가 스레드 j가 보유한 자원을 기다림
int wait_for_graph[NUM_THREADS][NUM_THREADS] = {0};
pthread_mutex_t graph_mutex; // WFG 접근 보호를 위한 뮤텍스
pthread_mutex_t print_mutex; // 출력 보호를 위한 뮤텍스 (로그 메시지 보호)
```

- `int wait_for_graph[NUM_THREADS][NUM_THREADS]` 배열에서 값이 `1`인 경우, 행 인덱스에 해당하는 스레드가 열 인덱스에 해당하는 스레드를 기다리고 있음(간선 존재)을 의미합니다. 초기값은 모두 `0`입니다.

```
/**
 * @brief 락 요청을 처리하고 WFG에 간선을 추가합니다.
 */
int custom_lock(Resource *res, int waiter_index) {
    pthread_mutex_lock(&print_mutex);
    printf(" [스레드 %s] 자원 %s 잠금 시도\n",
           Threads[waiter_index].pattern, res->name);
    pthread_mutex_unlock(&print_mutex);

    // 락 시도
    if (pthread_mutex_trylock(&res->lock) == 0) {
        // 잠금 성공
        res->owner_index = waiter_index;
```

```
        pthread_mutex_lock(&print_mutex);
        printf(" [스레드 %s] 자원 %s 잠금 성공\n",
            Threads[waiter_index].pattern, res→name);
        pthread_mutex_unlock(&print_mutex);

        return 0;
    } else {
        // 잠금 실패 (자원이 점유 중)
        int owner_index = res→owner_index;

        // WFG에 간선 추가: waiter_index가 owner_index를 기다림
        pthread_mutex_lock(&graph_mutex);
        wait_for_graph[waiter_index][owner_index] = 1;
        pthread_mutex_unlock(&graph_mutex);

        // 4번 항목: 출력 형식 준수
        pthread_mutex_lock(&print_mutex);
        printf(" [감시기] 스레드 %s → 스레드 %s를 기다림 (자원 %s)\n",
            Threads[waiter_index].pattern, Threads[owner_index].pattern, res
→name);
        pthread_mutex_unlock(&print_mutex);

        // 실제 잠금을 기다림 (여기서 영원히 대기하게 됨)
        pthread_mutex_lock(&res→lock);

        // 잠금 성공 후 (데드락 해소 후 락 획득): WFG에서 간선 제거
        pthread_mutex_lock(&graph_mutex);
        wait_for_graph[waiter_index][owner_index] = 0;
        pthread_mutex_unlock(&graph_mutex);

        // 소유자 업데이트
        res→owner_index = waiter_index;

        return -1;
    }
}
```

- 간선 추가는 `custom_lock` 함수 내에서 발생

- **잠금 시도 및 실패 확인:** 스레드(예: T1)가 `custom_lock`을 호출할 때, 내부적으로 `pthread_mutex_trylock`을 시도하여 즉시 락을 획득할 수 있는지 확인합니다.
- **대기 상태 진입:** `trylock`에 실패하면, 자원은 이미 다른 스레드(예: T2)에 의해 점유 중임을 확인합니다. T1은 **대기 스레드**, T2는 **보유 스레드**가 됩니다.
- **간선 추가:** `graph_mutex`를 잠근 상태에서, WFG 행렬에 **T1 → T2** 방향으로 간선을 `1`로 설정합니다.
  - `wait_for_graph[T1_INDEX][T2_INDEX] = 1;`
- **감시기 출력:** 간선 추가 후 "`[감시기] 스레드 T1(A$\rightarrow$B) $\rightarrow$ 스레드 T2(B$\rightarrow$A)를 기다림 (자원 B)`" 형식의 로그 메시지를 출력합니다.

2. DFS 기반 사이클 탐지

```
/**
 * @brief DFS를 사용하여 그래프에서 사이클을 탐지합니다.
 */
int dfs_cycle(int u, int color[]) {
    color[u] = GRAY;

    for (int v = 0; v < NUM_THREADS; v++) {
        if (wait_for_graph[u][v] == 1) { // u가 v를 기다림
            if (color[v] == GRAY) {
                return 1; // 사이클 발견 (Gray → Gray)
            }
            if (color[v] == WHITE) {
                if (dfs_cycle(v, color)) {
                    return 1;
                }
            }
        }
    }

    color[u] = BLACK;
    return 0;
}
```

```
/**
 * @brief 전체 WFG에 대해 데드락(사이클)이 있는지 확인합니다.
 */
int check_deadlock() {
    int color[NUM_THREADS];
    memset(color, WHITE, sizeof(color));

    pthread_mutex_lock(&graph_mutex);
    for (int i = 0; i < NUM_THREADS; i++) {
        if (color[i] == WHITE) {
            if (dfs_cycle(i, color)) {
                pthread_mutex_unlock(&graph_mutex);
                return 1; // 데드락 감지
            }
        }
    }
    pthread_mutex_unlock(&graph_mutex);
    return 0; // 데드락 없음
}
```

- Wait-for Graph(WFG)에 사이클(교착 상태)이 발생했는지 확인하고, 해당 상태가 5초 동안 지속될 경우 경고를 출력하는 기능을 구현합니다.
    - **원리:** 사이클 탐지를 위해 **깊이 우선 탐색(DFS)** 알고리즘을 사용하며, 노드(스레드)의 상태를 추적하기 위해 세 가지 색깔(상태) 배열을 사용합니다.
        - **White (0):** 아직 방문하지 않은 노드
        - **Gray (1):** 현재 DFS 경로(재귀 스택)에 있는 노드
        - **Black (2):** 탐색이 완료되어 사이클이 없음을 확인한 노드
    - **사이클 탐지:** `dfs_cycle` 함수는 현재 노드를 **Gray**로 표시하고 이웃 노드를 재귀적으로 탐색합니다. 탐색 중 인접한 이웃 노드(`v`)가 이미 **Gray** 상태인 경우 , 이는 **현재 경로로 되돌아오는 간선**이 발견되었음을 의미하므로, **교착 상태(Deadlock)를 나타내는 사이클**이 존재한다고 즉시 판단하고 `1`을 반환합니다.
    - **전체 확인:** `check_deadlock` 함수는 WFG의 모든 노드에서 DFS 탐색을 시작하여, 그래프 전체에 걸쳐 사이클이 있는지 확인합니다.

```
void *monitor_thread_func(void *arg) {
    int deadlock_counter = 0;

    while (1) {
        usleep(CHECK_INTERVAL_MS * 1000); // 0.1초 대기

        if (check_deadlock()) {
            deadlock_counter++;

            if (deadlock_counter >= DEADLOCK_THRESHOLD) {
                // 5초 동안(50회) 데드락이 해소되지 않고 유지됨

                // 4번 항목: 출력 형식 준수
                pthread_mutex_lock(&print_mutex);
                printf("\n === 데드락 감지됨! (%d초 경과) ===\n", DEADLOCK_WAIT
_TIME);
                printf(" 감지된 스레드들 :\n");
                printf(" - 스레드 %s\n", Threads[T1_INDEX].pattern);
                printf(" - 스레드 %s\n", Threads[T2_INDEX].pattern);
                printf("\n 프로그램을 종료합니다.\n");
                pthread_mutex_unlock(&print_mutex);

                cleanup_system();
                exit(0);
            }
        } else {
            // 데드락이 해소되었으므로 카운터 초기화
            deadlock_counter = 0;
        }
    }
    return NULL;
}
```

- **5초 유지 확인 및 종료 로직**
  - **감시 스레드:** 별도의 `monitor_thread_func`가 백그라운드에서 데드락을 감시합니다.
  - **반복 검사:** 스레드는 **0.1초** 간격(`CHECK_INTERVAL_MS`)으로 `check_deadlock()` 함수를 반복 호출합니다.

- **카운터:** 데드락이 감지되면 내부 카운터(`deadlock_counter`)를 증가시키고, 데드락이 해소되면 카운터를 `0` 으로 초기화합니다.
- **5초 유지 조건:** 카운터가 **50회**(`DEADLOCK_THRESHOLD` , 5초에 해당)에 도달하면, 이는 교착 상태가 5초 동안 지속되었음을 의미합니다.
- **경고 출력 및 종료:** 5초 조건이 충족되면, " `=== 데드락 감지됨! (5초 경과) ===` " 메시지를 출력하고 , `exit(0)` 를 호출하여 프로그램을 안전하게 종료합니다.

3. 데드락 재현

```
/**
 * @brief 스레드 T1의 작업 함수 (A→B 순서)
 */
void *thread_func_t1(void *arg) {
    // 1. 자원 A 잠금 시도 (R[0])
    custom_lock(&R[0], T1_INDEX);

    // T2가 B 락을 성공할 시간을 부여하여 교착 상태를 유도
    usleep(500000);

    // 2. 자원 B 잠금 시도 (R[1]) - T2가 B를 가지고 있어 대기
    custom_lock(&R[1], T1_INDEX);

    // *이 코드는 실행되지 않음 (데드락 감지 후 종료)*
    custom_unlock(&R[1], T1_INDEX);
    custom_unlock(&R[0], T1_INDEX);
    return NULL;
}

/**
 * @brief 스레드 T2의 작업 함수 (B→A 순서)
 */
void *thread_func_t2(void *arg) {
    // 1. 자원 B 잠금 시도 (R[1])
    custom_lock(&R[1], T2_INDEX);

    // T1이 A 락을 성공할 시간을 부여하여 교착 상태를 유도
```

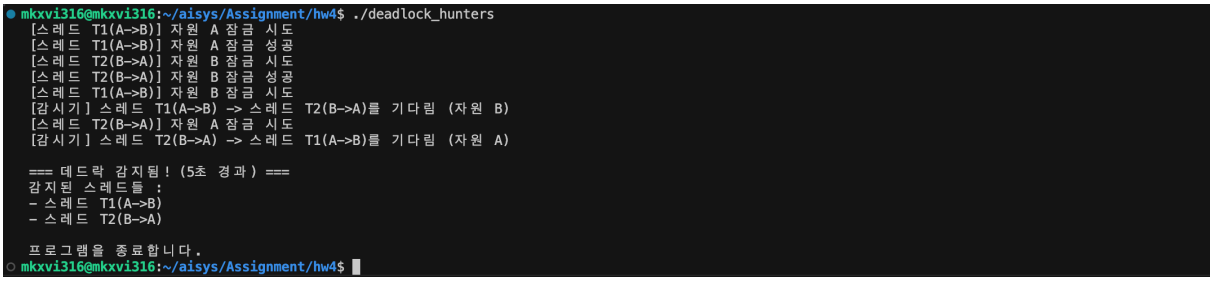
```
    usleep(500000);

    // 2. 자원 A 잠금 시도 (R[0]) - T1이 A를 가지고 있어 대기
    custom_lock(&R[0], T2_INDEX);

    // *이 코드는 실행되지 않음 (데드락 감지 후 종료)*
    custom_unlock(&R[0], T2_INDEX);
    custom_unlock(&R[1], T2_INDEX);
    return NULL;
}
```

- 이 항목은 두 스레드(`T1`, `T2`)가 자원 A와 B를 사용하여 **교착 상태(Deadlock)**를 의도적으로 발생시키는 시나리오를 구현합니다.
  - **스레드 T1 (A → B):**
    - T1은 첫 번째 자원인 **A**를 성공적으로 잠급니다.
    - 이후 T2가 B를 잠글 충분한 시간을 확보한 뒤, T1은 두 번째 자원인 **B**를 잠그려고 시도합니다. (T1은 A를 점유하고 B를 대기)
  - **스레드 T2 (B → A):**
    - T2는 첫 번째 자원인 **B**를 성공적으로 잠급니다.
    - 이후 T1이 A를 잠글 시간을 확보한 뒤, T2는 두 번째 자원인 **A**를 잠그려고 시도합니다. (T2는 B를 점유하고 A를 대기)
  - **결과:** T1은 **T2**가 보유한 B를 기다리고, T2는 **T1**이 보유한 A를 기다리는 **순환 대기** 상태가 형성되어 교착 상태가 재현됩니다. 이는 WFG에서 T1 → T2 →T1의 사이클로 나타나게 됩니다.

실행 결과


mkxvi316@mkxvi316:~/aisys/Assignment/hw4$ ./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.
mkxvi316@mkxvi316:~/aisys/Assignment/hw4$

✅ 성공!