**mutual exclusion 구현 학습**상호배제

컴퓨터공학과

2142851 김형준

운영체제 02분반

**목차**

1. **Dekker’s algorithm 구현**
   1. 소스 코드 및 설명
   2. 결과 화면
   3. 결과 분석
2. **Peterson’s algorithm 구현**
   1. 소스 코드 및 설명
   2. 결과 화면
   3. 결과 분석
3. **Dijkstra’s algorithm 구현**
   1. 소스 코드 및 설명
   2. 결과 화면
   3. 결과 분석
4. **Semaphore 구현**
   1. 소스 코드 및 설명
   2. 결과 화면
   3. 결과 분석
5. **알고리즘 종합 코드 및 시간 분석**
   1. 소스 코드
   2. 결과 화면
   3. 시간 분석
6. **리뷰**
   1. 동시성 및 경쟁 조건
   2. 성능 평가

**참고 사항**

1. **소스 코드 설명은 주석 처리를 통해 설명하였습니다.  
   (동일한 코드의 경우 중복하여 주석을 달지는 않았습니다.)**
2. **코드 내에서 상호 배제가 정상적으로 작동하는지 확인하기 위해 cnt 변수를 추가하여, 연산이 실행될 때마다 1씩 증가시켰습니다. 올바르게 작동할 경우 최종 cnt의 값이 100이 되어야 합니다.**
3. **목차 5번의 코드를 통해 모든 알고리즘을 한 코드에서 실행할 수 있도록 하였습니다.**
4. **추가로, 5번 코드에 시간 측정 기능을 넣었습니다.**
5. **Dekker’s algorithm 구현**
6. **소스 코드 및 설명**

*#include* <stdio.h>

*#include* <pthread.h>

int flag[2] = {0, 0}; *// 각 스레드의 플래그 상태를 저장하는 배열*

int turn = 0; *// 턴을 나타내는 변수*

int n[101]; *// 1부터 100까지의 숫자를 저장하는 배열*

int cnt = 0; *// 출력된 수의 카운트*

typedef struct {

    int start;

    int end;

    int thread\_id;

} args\_t;

void dekker\_critical\_section\_enter(int *thread\_id*) {

*// thread: 크리티컬 섹션 들어갈래*

    flag[thread\_id] = 1;

*// 상대 스레드의 flag가 올려져있다면*

*while* (flag[1 - thread\_id]) {

*// 내 차례도 아니라면*

*if* (turn != thread\_id) {

*// 대기..*

            flag[thread\_id] = 0;

*while* (turn != thread\_id) {}

*// 다시 들어가기*

            flag[thread\_id] = 1;

        }

    }

}

void dekker\_critical\_section\_exit(int *thread\_id*) {

*// 내 차례 끝났을 경우 turn 양도*

    turn = 1 - thread\_id;

*// 이제 안 들어갈거야.*

    flag[thread\_id] = 0;

}

void\* dekker\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 데커 알고리즘 크리티컬 영역 입장*

        dekker\_critical\_section\_enter(thread\_id);

        printf("Thread[%2d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 데커 알고리즘 크리티컬 영역 퇴장*

        dekker\_critical\_section\_exit(thread\_id);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

int main() {

*for*(int i = 0; i < 101; i++)

        n[i] = i;

    pthread\_t threads[2];

    args\_t thread\_args[2];

*// thread 0번*

    thread\_args[0].start = 1;

    thread\_args[0].end = 50;

    thread\_args[0].thread\_id = 0;

*// thread 1번*

    thread\_args[1].start = 51;

    thread\_args[1].end = 100;

    thread\_args[1].thread\_id = 1;

*// create threads*

    pthread\_create(&threads[0], NULL, dekker\_func, &thread\_args[0]);

    pthread\_create(&threads[1], NULL, dekker\_func, &thread\_args[1]);

*// wait to end*

    pthread\_join(threads[0], NULL);

    pthread\_join(threads[1], NULL);

    printf("printf count: %d\n", cnt);

*return* 0;

}

1. **결과 화면**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **결과 분석**

Dekker의 알고리즘을 사용하여 상호 배제를 구현합니다. 2개의 thread가 각각 1~50, 51~100까지의 숫자에 곱셈 연산을 하고, 결과와 thread의 번호를 출력합니다. 이를 통해 임계 구역에 대한 접근을 제어합니다.

코드의 순서는 아래와 같습니다.

1. 숫자 배열 n을 0부터 100까지 초기화합니다. (0은 사용하지 않음)
2. 2개의 스레드가 생성되고, ‘dekker\_func’ 함수를 수행합니다.
3. 스레드가 실행되면 ‘dekker\_func’ 함수에서 ‘start’부터 ‘end’까지 반복하면서 각 숫자를 3으로 곱합니다.
4. 임계 구역 진입 시 ‘dekker\_critical\_section\_enter’, 퇴장 시 ‘dekker\_critical\_section\_exit’ 함수를 실행합니다.
5. 상호 배제를 보장하기 위해 ‘flag’와 ‘turn’ 변수를 사용합니다.
6. 임계 구역에서 나올 때, ‘turn’ 변수의 값을 변경하여 다른 스레드가 진입할 수 있도록 합니다.
7. 두 스레드가 완료되면, ‘cnt’ 변수의 값을 확인합니다. 변수는 숫자가 3으로 곱해지는 동안 카운트됩니다.
8. 모든 스레드가 종료되고, ‘cnt’ 값이 100이라면, 각 숫자가 한 번씩 정확하게 처리되었음을 알 수 있습니다.

* 코드의 결과를 보면 두 스레드가 숫자를 잘 나누어 처리하고 있습니다. 또한, 출력된 ‘cnt’ 값이 100으로 나오며, 이는 Dekker의 알고리즘이 동작하고 상호 배제를 보장하고 있음을 보여줍니다.

1. **Peterson’s algorithm 구현**
2. **소스 코드 및 설명**

*#include* <stdio.h>

*#include* <pthread.h>

int flag[2] = {0, 0};

int turn = 0;

int n[101];

int cnt = 0;

typedef struct {

    int start;

    int end;

    int thread\_id;

} args\_t;

void peterson\_critical\_section\_enter(int *thread\_id*) {

*// thread 크리티컬 섹션 진입 시도*

    flag[thread\_id] = 1;

*// 현재 thread가 진입을 시도하므로, 상대 thread의 차례로 turn을 설정*

*// 너가 먼저 들어가*

    turn = 1 - thread\_id;

*// 상대 thread가 진입 시도 & 상대가 크리티컬 섹션에 진입할 차례인 경우 -> 대기*

*while* (flag[1 - thread\_id] && turn == 1 - thread\_id) {}

}

void peterson\_critical\_section\_exit(int *thread\_id*) {

    flag[thread\_id] = 0;

}

void\* peterson\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 피터슨 알고리즘 크리티컬 섹션 진입*

        peterson\_critical\_section\_enter(thread\_id);

        printf("Thread[%2d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 피터슨 알고리즘 크리티컬 섹션 퇴장*

        peterson\_critical\_section\_exit(thread\_id);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

int main() {

*for*(int i = 0; i < 101; i++)

        n[i] = i;

    pthread\_t threads[2];

    args\_t thread\_args[2];

*// Thread 0번*

    thread\_args[0].start = 1;

    thread\_args[0].end = 50;

    thread\_args[0].thread\_id = 0;

*// Thread 1번*

    thread\_args[1].start = 51;

    thread\_args[1].end = 100;

    thread\_args[1].thread\_id = 1;

*// thread create*

    pthread\_create(&threads[0], NULL, peterson\_func, &thread\_args[0]);

    pthread\_create(&threads[1], NULL, peterson\_func, &thread\_args[1]);

*// wait to end*

    pthread\_join(threads[0], NULL);

    pthread\_join(threads[1], NULL);

    printf("printf count: %d\n", cnt);

*return* 0;

}

1. **결과 화면**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **결과 분석**

Peterson의 알고리즘을 사용하여 2개의 스레드가 상호 배제를 유지하며 임계 영역을 실행합니다. 스레드 각각의 범위에서 정수를 곱셈 연산하여 결과를 출력합니다.

* 각 스레드가 처리한 모든 숫자가 3으로 곱해져 출력됩니다.
* ‘cnt’ 변수는 모든 숫자가 처리될 때마다 1씩 증가하므로, 상호 배제가 잘 동작했다면 최종적으로 ‘cnt’ 값은 100이 되어야 합니다.
* 코드의 결과를 보면 두 스레드가 숫자를 잘 나누어 처리하고 있습니다. 또한, 출력된 ‘cnt’ 값이 100으로 나오며, 이는 Peterson의 알고리즘이 동작하고 상호 배제를 보장하고 있음을 보여줍니다.

1. **Dijkstra’s algorithm 구현**
2. **소스 코드 및 설명**

*#include* <stdio.h>

*#include* <pthread.h>

*#define* N 4 *// thread count*

*#define* RANGE 100

*// thread flag*

enum state { IDLE = 0, WANT\_IN, IN\_CS };

int flag[N] = {IDLE};

int turn = 0;

int n[101];

int cnt = 0;

typedef struct {

    int start;

    int end;

    int thread\_id;

} args\_t;

void dijkstra\_critical\_section\_enter(int *thread\_id*) {

    int j;

*do* {

*// 임계 영역 진입 시도 1단계*

        flag[thread\_id] = WANT\_IN;

*while* (turn != thread\_id) {

*if* (flag[turn] == IDLE) {

                turn = thread\_id;

            }

        }

*// 임계 영역 진입 시도 2단계*

        flag[thread\_id] = IN\_CS;

        j = 0;

*// 자신 이외에 in-CS 영역이 있는지 확인*

*while* (j < N && (j == thread\_id || flag[j] != IN\_CS)) {

            j++;

        }

    } *while* (j < N);

*// 없으면 내가 크리티컬 섹션에 들어가기*

}

void dijkstra\_critical\_section\_exit(int *thread\_id*) {

*// flag를 IDLE로 설정*

    flag[thread\_id] = IDLE;

*// 다음 턴으로*

    turn = (turn + 1) % N;

}

void\* dijkstra\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

        dijkstra\_critical\_section\_enter(thread\_id); *// Dijkstra 임계 영역 진입*

        printf("Thread[%d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

        dijkstra\_critical\_section\_exit(thread\_id); *// Dijkstra 임계 영역 종료*

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

int main() {

*// 숫자 배열 초기화*

*for* (int i = 0; i < 101; i++) {

        n[i] = i;

    }

    pthread\_t threads[N];

    args\_t thread\_args[N];

*// 각 스레드의 범위 설정*

    int step = RANGE / N;

*for* (int i = 0; i < N; i++) {

        thread\_args[i].start = i \* step + 1;

        thread\_args[i].end = (i + 1) \* step;

        thread\_args[i].thread\_id = i;

*// 스레드 생성*

        pthread\_create(&threads[i], NULL, dijkstra\_func, &thread\_args[i]);

    }

*// wait to end*

*for* (int i = 0; i < N; i++) {

        pthread\_join(threads[i], NULL);

    }

*// 최종 출력 카운트*

    printf("printf count: %d\n", cnt);

*return* 0;

}

1. **결과 화면**

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **결과 분석**

Dijkstra의 상호 배제 알고리즘을 사용하여 4개의 스레드가 상호 배제 조건을 만족하면서, 각각 주어진 범위의 정수를 곱셈 연산하여 출력합니다.

* 각 스레드는 지정된 범위의 모든 숫자를 3으로 곱하여 출력합니다.
* 연산 및 출력에 성공하면 변수 ‘cnt’의 값을 1 증가시킵니다.
* 스레드들이 상호 배제에 성공하면, ‘cnt’ 변수는 정확히 100이 됩니다.
* 코드의 결과를 보면 두 스레드가 숫자를 잘 나누어 처리하고 있습니다. 또한, 출력된 ‘cnt’ 값이 100으로 나오며, 이는 Dijkstra의 알고리즘이 동작하고 상호 배제를 보장하고 있음을 보여줍니다.

1. **Semaphore 구현**
2. **소스 코드 및 설명**

*#include* <stdio.h>

*#include* <pthread.h>

*#include* <semaphore.h>

*#define* RANGE 100

*#define* THREAD\_COUNT 4

*// 세마포 객체*

sem\_t sem;

int n[101];

int cnt = 0;

typedef struct {

    int start;

    int end;

    int thread\_id;

} args\_t;

void\* semaphore\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 세마포를 통해 크리티컬 섹션 진입 제어*

        sem\_wait(&sem);

        printf("Thread[%d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 크리티컬 섹션 종료 후 세마포 인카운트*

        sem\_post(&sem);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

int main() {

*for* (int i = 0; i < 101; i++) {

        n[i] = i;

    }

*// 세마포 초기값*

    sem\_init(&sem, 0, 1);

    pthread\_t threads[THREAD\_COUNT];

    args\_t thread\_args[THREAD\_COUNT];

*// thread 범위*

    int step = RANGE / THREAD\_COUNT;

*for* (int i = 0; i < THREAD\_COUNT; i++) {

        thread\_args[i].start = i \* step + 1;

        thread\_args[i].end = (i + 1) \* step;

        thread\_args[i].thread\_id = i;

*// thread create*

        pthread\_create(&threads[i], NULL, semaphore\_func, &thread\_args[i]);

    }

*// wait to end*

*for* (int i = 0; i < THREAD\_COUNT; i++) {

        pthread\_join(threads[i], NULL);

    }

*// 세마포 파괴!!*

    sem\_destroy(&sem);

    printf("Total count: %d\n", cnt);

*return* 0;

}

1. **결과 화면**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **결과 분석**

4개의 스레드가 각각 주어진 범위의 정수를 곱셈 연산하여 출력합니다. ‘cnt’ 변수를 통해 그 수를 카운트합니다. semaphore를 사용하여 스레드가 임계 영역에 진입할 때 동기화를 관리합니다.

* 스레드들은 semaphore를 사용하여 임계 영역에 진입합니다.
* 임계 영역에서 스레드 각각의 범위의 정수를 3으로 곱하여 출력합니다. 이때, 변수 ‘cnt’의 값을 1 증가시킵니다.
* ‘cnt’ 변수는 100이 되어야 합니다. 만약 100이라면, 각 스레드가 1부터 100까지의 모든 정수를 정확히 한 번씩 처리했음을 의미합니다.
* 코드의 결과를 보면, ‘cnt’ 값이 100으로 나오며, 이는 semaphore가 잘 작동하고 상호 배제 조건을 성공적으로 준수했다는 것을 알 수 있습니다.

1. **알고리즘 종합 코드 및 시간 분석**
2. **소스 코드**

*#include* <stdio.h>

*#include* <pthread.h>

*#include* <semaphore.h>

*#include* <stdlib.h>

*#define* RANGE 100 *// 숫자 범위*

int turn = 0;

int n[101];

int cnt = 0;

int\* flag;

int thread\_count;

typedef struct {

    int start;

    int end;

    int thread\_id;

} args\_t;

*// 1. Dekker*

void dekker\_critical\_section\_enter(int *thread\_id*);

void dekker\_critical\_section\_exit(int *thread\_id*);

void\* dekker\_func(void\* *args*);

*// 2. Peterson*

void peterson\_critical\_section\_enter(int *thread\_id*);

void peterson\_critical\_section\_exit(int *thread\_id*);

void\* peterson\_func(void\* *args*);

*// 3. Dijkstra*

void dijkstra\_critical\_section\_enter(int *thread\_id*);

void dijkstra\_critical\_section\_exit(int *thread\_id*);

void\* dijkstra\_func(void\* *args*);

enum state { IDLE = 0, WANT\_IN, IN\_CS };

*// 4. Semaphore*

void\* semaphore\_func(void\* *args*);

sem\_t sem;

char algorithms[5][30] = { "exit", "Dekker's Algorithm", "Peterson's Algorithm", "Dijkstra's Algorithm", "Semaphore" };

int main() {

*for* (int i = 0; i< 101; i++) {

        n[i] = i;

    }

*while*(1) {

        int input;

        cnt = 0;

        turn = 0;

        printf("<Select Your Algorithm>\n");

        printf("---------- Select ----------\n");

*for*(int i = 0; i < 5; i++) {

            printf("%2d. %s\n", i, algorithms[i]);

        }

        printf("\n");

        printf("input: ");

*if* (scanf("%d", &input) != 1) {

*while* (getchar() != '\n'); *// 입력 스트림을 비움*

            system("clear");

            printf("Invalid input. (input type: int)\n\n");

*continue*; *// 다시 입력 받기*

        }

*if* (input == 0) *break*;

*else* *if* (input == 1 || input == 2 || input == 3 || input == 4) {

            printf("================ %s Start ================\n", algorithms[input]);

        }

*else* {

            system("clear");

            printf("Wrong input value\n\n");

*continue*;

        }

        pthread\_t\* threads;

        args\_t\* thread\_args;

        int thread\_args\_count;

*if* (input == 1 || input == 2) {

            thread\_count = 2;

        }

*else* *if* (input == 3 || input == 4) {

            thread\_count = 4;

        }

*// semaphore first setting*

*if* (input == 4) {

            sem\_init(&sem, 0, 1);

        }

*// thread malloc*

        threads = (pthread\_t\*)malloc(sizeof(pthread\_t) \* thread\_count);

        thread\_args = (args\_t\*)malloc(sizeof(args\_t) \* thread\_count);

*// flag malloc*

        flag = (int\*)malloc(sizeof(int) \* thread\_count);

*// flag setting*

*if* (input == 1 || input == 2 || input == 3) {

*for* (int i = 0; i < thread\_count; i++) {

                flag[i] = 0;

            }

        }

*else* *if* (input == 3) {

*for* (int i = 0; i < thread\_count; i++) {

                flag[i] = IDLE;

            }

        }

        int step = RANGE / thread\_count;

        double aveTime = 0;

*for* (int repeat = 0; repeat < 10; repeat++) {

*// TIME!!*

            clock\_t startTime, endTime;

            double totalTime;

            startTime = clock();

*// thread setting*

*for* (int i = 0; i < thread\_count; i++) {

                thread\_args[i].start = i \* step + 1;

                thread\_args[i].end = (i + 1) \* step;

                thread\_args[i].thread\_id = i;

*// thread create*

*if* (input == 1) pthread\_create(&threads[i], NULL, dekker\_func, &thread\_args[i]);

*else* *if* (input == 2) pthread\_create(&threads[i], NULL, peterson\_func, &thread\_args[i]);

*else* *if* (input == 3) pthread\_create(&threads[i], NULL, dijkstra\_func, &thread\_args[i]);

*else* *if* (input == 4) pthread\_create(&threads[i], NULL, semaphore\_func, &thread\_args[i]);

            }

*/\* function \*/*

*// wait to the end*

*for* (int i = 0; i < thread\_count; i++) {

                pthread\_join(threads[i], NULL);

            }

*// destroy semaphore*

*if* (input == 4) {

                sem\_destroy(&sem);

            }

*// TIME!!*

            endTime = clock();

            totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

            printf("\n");

            printf("================ Result ================\n");

            printf("Algorithm           | %s\n", algorithms[input]);

            printf("Mutual exclusion    | %s (cnt: %d)\n", ((cnt == 100) ? "succeeded!" : "failed."), cnt);

            printf("Thread count        | %d\n", thread\_count);

            printf("Total Time          | %f seconds\n", totalTime);

            printf("================= End =================\n");

            printf("\n");

            aveTime += totalTime;

        }

        printf("[%lf]\n", aveTime/10);

        free(flag);

        free(threads);

        free(thread\_args);

*// system("pause");*

        printf("Press Enter to continue :)");

        getchar(); *// Wait to enter 'Enter' :)*

        getchar();

        system("clear");

    }

*return* 0;

}

*/\* 1. Dekker \*/*

void dekker\_critical\_section\_enter(int *thread\_id*) {

*// thread: 크리티컬 섹션 들어갈래*

    flag[thread\_id] = 1;

*// 상대 스레드의 flag가 올려져있다면*

*while* (flag[1 - thread\_id]) {

*// 내 차례도 아니라면*

*if* (turn != thread\_id) {

*// 대기..*

            flag[thread\_id] = 0;

*while* (turn != thread\_id) {}

*// 다시 들어가기*

            flag[thread\_id] = 1;

        }

    }

}

void dekker\_critical\_section\_exit(int *thread\_id*) {

*// 내 차례 끝났을 경우 turn 양도*

    turn = 1 - thread\_id;

*// 이제 안 들어갈거야.*

    flag[thread\_id] = 0;

}

void\* dekker\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 데커 알고리즘 크리티컬 영역 입장*

        dekker\_critical\_section\_enter(thread\_id);

        printf("Thread[%2d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 데커 알고리즘 크리티컬 영역 퇴장*

        dekker\_critical\_section\_exit(thread\_id);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

*/\* 2. Peterson \*/*

void peterson\_critical\_section\_enter(int *thread\_id*) {

*// thread 크리티컬 섹션 진입 시도*

    flag[thread\_id] = 1;

*// 현재 thread가 진입을 시도하므로, 상대 thread의 차례로 turn을 설정*

*// 너가 먼저 들어가*

    turn = 1 - thread\_id;

*// 상대 thread가 진입 시도 & 상대가 크리티컬 섹션에 진입할 차례인 경우 -> 대기*

*while* (flag[1 - thread\_id] && turn == 1 - thread\_id) {}

}

void peterson\_critical\_section\_exit(int *thread\_id*) {

    flag[thread\_id] = 0;

}

void\* peterson\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 피터슨 알고리즘 크리티컬 섹션 진입*

        peterson\_critical\_section\_enter(thread\_id);

        printf("Thread[%2d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 피터슨 알고리즘 크리티컬 섹션 퇴장*

        peterson\_critical\_section\_exit(thread\_id);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

*/\* 3. Dijkstra \*/*

void dijkstra\_critical\_section\_enter(int *thread\_id*) {

    int j;

*do* {

*// 임계 영역 진입 시도 1단계*

        flag[thread\_id] = WANT\_IN;

*while* (turn != thread\_id) {

*if* (flag[turn] == IDLE) {

                turn = thread\_id;

            }

        }

*// 임계 영역 진입 시도 2단계*

        flag[thread\_id] = IN\_CS;

        j = 0;

*// 자신 이외에 in-CS 영역이 있는지 확인*

*while* (j < thread\_count && (j == thread\_id || flag[j] != IN\_CS)) {

            j++;

        }

    } *while* (j < thread\_count);

*// 없으면 내가 크리티컬 섹션에 들어가기*

}

void dijkstra\_critical\_section\_exit(int *thread\_id*) {

*// flag를 IDLE로 설정*

    flag[thread\_id] = IDLE;

*// 다음 턴으로*

    turn = (turn + 1) % thread\_count;

}

void\* dijkstra\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

        dijkstra\_critical\_section\_enter(thread\_id); *// Dijkstra 임계 영역 진입*

        printf("Thread[%d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

        dijkstra\_critical\_section\_exit(thread\_id); *// Dijkstra 임계 영역 종료*

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

*/\* 4. Semaphore \*/*

void\* semaphore\_func(void\* *args*) {

    args\_t\* thread\_args = (args\_t\*)args;

    int start = thread\_args->start;

    int end = thread\_args->end;

    int thread\_id = thread\_args->thread\_id;

*for* (int iter = start; iter <= end; iter++) {

*// 세마포를 통해 크리티컬 섹션 진입 제어*

        sem\_wait(&sem);

        printf("Thread[%d]: %3d \* 3 = %3d\n", thread\_id, n[iter], n[iter] \* 3);

        cnt++;

*// 크리티컬 섹션 종료 후 세마포 인카운트*

        sem\_post(&sem);

    }

    printf("=== thread[%2d] algorithm end ===\n", thread\_id);

*return* NULL;

}

1. **결과 화면**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

input: 3인 경우 (다익스트라 알고리즘)

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **시간 분석**

\* 시간 측정 및 분석을 진행하였으나, 실행 간 편차가 커서 정확한 측정이 불가했습니다.

- Dekker’s Algorithm

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- Peterson’s Algorithm

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- Dijkstra’s Algorithm

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- Semaphore

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1회차 평균 (10회)

|  |  |  |  |
| --- | --- | --- | --- |
| Dekker | Peterson | Dijkstra | Semaphore |
| 0.000304 | 0.000264 | 0.001456 | 0.000583 |

2회차 평균 (10회)

|  |  |  |  |
| --- | --- | --- | --- |
| Dekker | Peterson | Dijkstra | Semaphore |
| 0.000510 | 0.000242 | 0.079997 | 0.000622 |

시간 측정 결과,

Peterson < Dekker < Semaphore < Dijkstra

순서임을 알 수 있었습니다.

다익스트라 알고리즘의 경우, 시간의 편차가 너무 크다는 문제가 있습니다.

예측할 수 있는 문제점으로는 아래와 같습니다.

* 공유 자원에 여러 스레드가 접근할 때 동기화가 제대로 이루어지지 않으면, 스레드가 제때 임계 영역에 접근하지 못합니다.
* 함수 호출, 반복문, 자료 구조, 알고리즘의 복잡도 등을 추가 검토할 필요가 있습니다.

이에 코드의 추가 개선이 필요함을 느꼈습니다.

1. **리뷰**
2. **동시성 및 경쟁 조건**

동시성: 여러 스레드가 동시에 실행되어 자원을 공유하거나 작업을 병렬로 수행하는 것

경쟁 조건은 데이터 손상, 부정확한 결과, 교착 상태와 같은 문제가 발생시킵니다.

코드에서 발생할 수 있는 경쟁 조건으로는 ‘cnt’ 변수의 증가가 있습니다.

만약 동시성이 적절히 제어되지 않았다면 잘못된 결과가 나왔을 것입니다.

이러한 동시성을 관리하기 위한 방법으로는

* + 뮤텍스
  + 세마포어
  + 락
  + 상호 배제 알고리즘

등이 있습니다.

작성한 코드에서는 상호 배제와 semaphore를 통해 동시성을 처리하고 있습니다.

개선 사항으로는 더욱 세밀한 동기화, 최신 동시성 및 병렬 프로그래밍 기법에 대한 학습을 하고, 적용을 통해 코드의 안정성과 효율성을 높이는 것입니다.

1. **성능 평가**

성능을 평가하기 위해서는 실행 효율과 동작 품질을 측정해야 합니다. 이를 위해서는 다음과 같은 방법을 사용할 수 있습니다.

* + 실행 시간을 측정하고 최적화되었는지 확인합니다. 추가로 병목 현상을 식별합니다.
  + CPU, 메모리, 입출력 자원 등의 사용량을 모니터링합니다.
  + 각 스레드의 동작을 분석하여 비효율적인 대기 시간을 확인합니다.
  + 코드에 다양한 부하를 주어 프로그램이 안정적으로 동작하는지 테스트합니다. 예를 들어, 스레드 수를 늘리거나 입력 데이터의 크기를 크게 증가시켜 프로그램의 동작을 평가합니다.
  + 다른 유사한 코드와 비교하여 상대적인 성능을 평가합니다.
  + 최적화의 가능성을 파악합니다. 예를 들어, 동기화의 개선, 데이터 접근 패턴의 최적화 등이 있습니다.