1.개요

오퍼레이팅 시스템에서 핵심적인 Problem들 중 Producer – Consumer Problem, Readers- Writers Problem, Dining Philosphers Problem을 다양한 조건에서 해결하기 위하여 문제 해결방법을 설계하였다.

개발환경

개발언어: C

Os: Ubuntu 22.04.2 LTS

gccVersion: gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0

코드편집기: VsCode

Producer – Consumer Problem은 생산자와 소비자의 활동을 관리하는 문제로, 생산자는 Buffer에 생산을 하고, 소비자는 Buffer에서 생산자가 만든 것을 소비한다. 이 때 생산자는 Buffer의 Size를 고려하여, Buffer가 꽉 찼다면 더 이상 생산을 못하도록 하고, 소비자가 Buffer를 비워준다면(Consume) 다시 생산을 하도록 한다. 또한, 소비자는 Buffer가 Empty상태라면, 더 이상 소비를 하지 못하도록 하고, 생산자가 Buffer를 채워준다면 (Prodeuce)다시 소비를 하도록 한다.

Readers- Writers Problem은 Writes와 Readers의 활동을 관리하는 문제로, Writer는 특정 변수를 변경하고, Reader는 변수를 출력하도록 한다. 이 때 Writer는 Reader가 변수를 Read하거나 다른 Writer가 변수를 변경하는 도중에는 변수에 접근할 수 없으며, Reader는 Writer가 변수를 변경하는 도중에 변수에 접근하지 못하도록 한다. 하지만 Reader는 변수에 변화를 주지 않으므로 다수의 Reader 변수에 접근할 수 있도록 허용한다.

Dining Philosphers Problem은 원탁에 5명의 철학자와 5개의 포크가 있을 때 철학자가 자신의 양옆에 있는 2개의 포크를 집어야만 식사를 할 수 있다는 문제로, 5명의 철학자가 각자 포크를 하나씩만 들고 있는다면 5명의 철학자 모두 식사를 하지 못하는 상태로 멈춰있게 되고, 이 상태가 유지된다. 이 문제에서 철학자는 Process(혹은 Thread)를 나타내고, 포크는 Process가 필요로하는 공유자원을 나타내며, 5명의 철학자 그 누구도 식사를 하지 못하는 상태는 Process들의 DeadLock 상태를 나타낸다. 이 때 철학자의 행동을 제어해서 DeadLock상태에 빠지지 않도록 하는 문제이다.

2. Producer – Consumer Problem

Producer – Consumer Problem을 해결한다. 이 때 Buffer의 Size가 infinite 한 경우와 Bounded된 경우, 두 가지 조건에서 문제를 해결하였다.

```
void* thread_p(void* arg) {//producer
    int i, val;
    for (i = 0; i < ITER; i++) {
        val = x;
        printf("Poducuer %u:%d\n", (unsigned int)pthread_self(), x);
        x = val + 1;
    }

    pthread_exit(NULL);
}

void* thread_c(void* arg) {//consumer
    int i, val;
    for (i = 0; i < ITER; i++) {
        val = x;
        printf("Consumer %u: %d\n", (unsigned int)pthread_self(), val);
        x = val - 1;
    }

    pthread_exit(NULL);
}
```

아무런 조치도 취하지 않은 Producer – Consumer Problem 코드이다. Produecer는 전역변수 x값을 지역변수 val에 저장하고, x값에 val+1값을 다시 저장하는 방식으로 x값을 증가시킨다.

Consumer는 전역변수 x값을 지역변수 val에 저장하고, x값에 val-1값을 다시 저장하는 방식으로 x값을 감소시킨다.

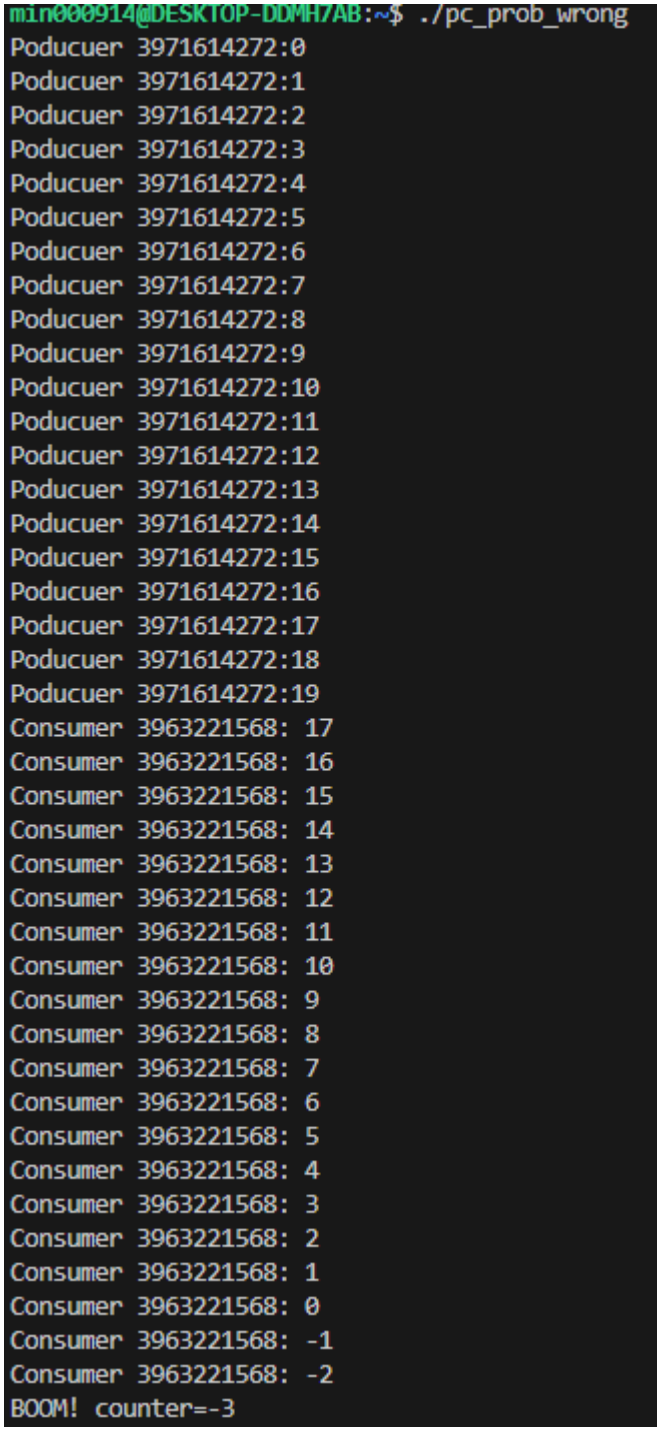
문제없이 작동할 것 같지만, 해당 코드에서 Race Condition을 고려한다면 그렇지 않다. 즉, Producer와 Consumer가 전역변수인 x를 동시에 접근한다면, x에 값이 Os스케쥴링에 따라 비결정적으로 변하게 되고, 다음 예시와 같이 예측할 수 없는 결과를 초래한다.

또한, X가 Buffer에 들어있는 생산품의 수라고 했을 때 Consumer는 Buffer가 비어있다면 소비를 멈출 수 있어야하거나, Buffer의 Size의 제한이 있을 때 Producer는 생산을 멈춰야 한다.

Producer thread Id: x값

Consumer thread Id: x값 으로 출력된다.

마지막 x값이 0이 아니라면 Boom!을 맞다면, OK counter을 출력한다.



다음과 같이, Consumer와 Producer의 실행횟수가 같음에도 0이 출력되지 않는 모습을 볼 수 있다. 이는 전역변수 접근 시 상호배제가 없어 여러 Thread에서 동시접근을 하며

일어나는 문제가 원인이다.

이를 막기 위해서는 Thread(혹은 Process)에서 전역변수(공유자원)에 접근할 경우인 Critical Section 영역에서는 Mutual Exclusion을 활용해서 공유자원에 여러 Thread에서 동시에 접근하지 못하도록 막아야한다. 또한, Buffer의 상태에 따라 Condition Variable(CV)을 활용해서 소비자와 생산자의 행동에 제약을 두어야한다.

다음은 Bounded Buffer상태를 가정한 코드이다.(Buffer의 Size는 10으로 설정하으며, Producer와 Consumer 각각 30번씩 실행되도록 하였다.)

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define ITER 30//반복횟수
#define MAX 10//BufferSize

void* thread_p(void* arg);
void* thread_c(void* arg);

int x = 0;
sem_t m;//전역변수 x에 대한 상호배제를 위한 Sem
sem_t fill;//Buffer가 비었을 경우 소비자의 소비를 막기위한 CV
sem_t empty;//Buffer가 꽉 찼을 경우 생산자의 생산을 막기위한 CV

int main() {
    pthread_t tid1, tid2;
    sem_init(&m, 0, 1);
    sem_init(&fill, 0, 0);
    sem_init(&empty, 0, MAX);
    pthread_create(&tid1, NULL, thread_p, NULL);
    pthread_create(&tid2, NULL, thread_c, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (x != 0)
        printf("BOOM! counter=%d\n", x);
    else
        printf("OK counter=%d\n", x);

    sem_destroy(&m);
    sem_destroy(&fill);
```

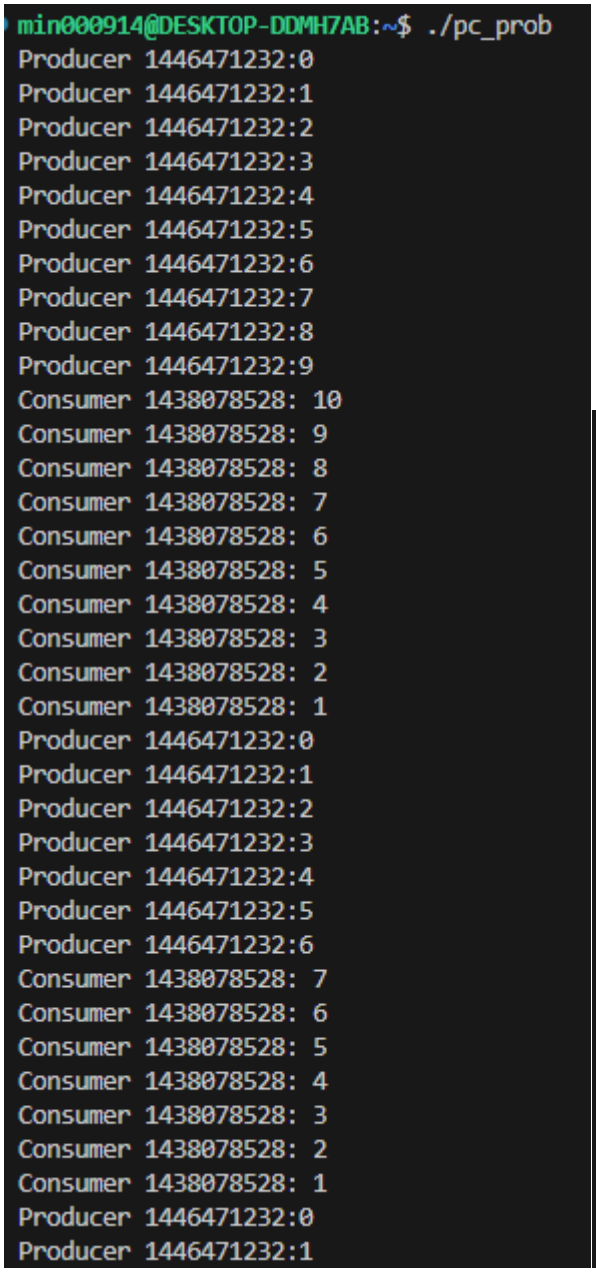
```
    sem_destroy(&empty);

    return 0;
}

void* thread_p(void* arg) {//생산자
    int i, val;
    for (i = 0; i < ITER; i++) {
        sem_wait(&empty);
        sem_wait(&m);

        val = x;
        printf("Producer %u:%d\n", (unsigned int)pthread_self(), x);
        x = val + 1;

        sem_post(&m);
        sem_post(&fill);
    }

    pthread_exit(NULL);
}

void* thread_c(void* arg) {//소비자
    int i, val;
    for (i = 0; i < ITER; i++) {
        sem_wait(&fill);
        sem_wait(&m);

        val = x;
        printf("Consumer %u: %d\n", (unsigned int)pthread_self(), val);
        x = val - 1;

        sem_post(&m);
        sem_post(&empty);
    }

    pthread_exit(NULL);
}
```

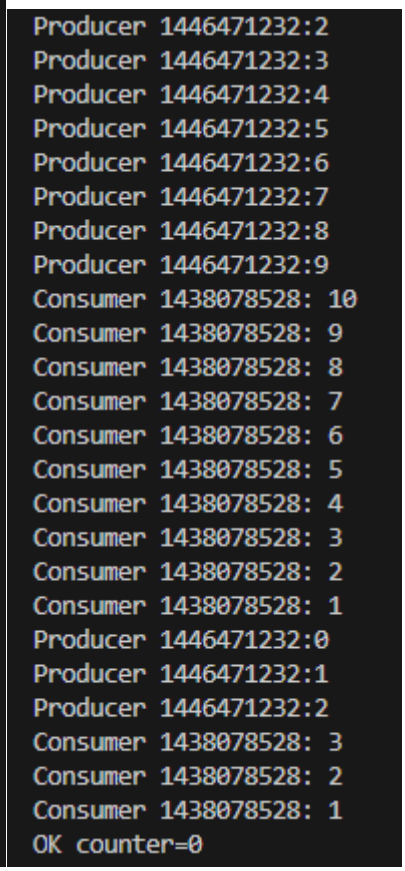
세마포어를 활용해서 Mutual Exclusion과 CV를 구현하였다. 전역변수 x에대한 동시 접근을 막기위하여 x에 접근하는 구간을 Producer와 Consume 두 곳 모두에 sem_wait(&m);과 sem_post(&m); 으로 감싸주어 Mutual Exclusion을 구현하였다.

CV를 구현하기 위하여 fill과 empty라는 세마포어를 생성하였다. Fill은 버퍼가 비었는지 확인하는 세마포어로 0부터 시작하고, Producer가 생산한다면 1씩증가하고, Consumer가 소비한다면 1씩 감소하게 하여 0보다 작아진다면 즉, Buffer가 비었다면 Consumer를

Lock하도록하였다. Empty는 버퍼가 꽉찼는지 확인하는 세마포어로 Buffer의 Size만큼 만들어 Producer가 생산할 때마다 empty값을 줄여 0보다 작다면 즉, BufferSize만큼 생산을 했다면 Lock이걸리도록 하였다.

다음은 실행 예시이다.





Producer는 Buffer의 MaxSize인 10만큼 생산했다면 Consumer가 소비하도록 하며, Consumer는 Buffer가 비었다면, Producer가 생산하도록한다. 또한, 상호배제로 인해 RaceCondition이 일어나지 않아 각각 30번씩 Producer와 Consumer가 실행되었을 때 최종 buffer에 아무것도 남지 않았음(0)을 확인할 수 있다.

다음은 Infinite Buffer상태를 가정한 코드이다.(버퍼의 Size는 무한대이고, Producer와 Consumer 각각 30번씩 수행하도록 하였다.)

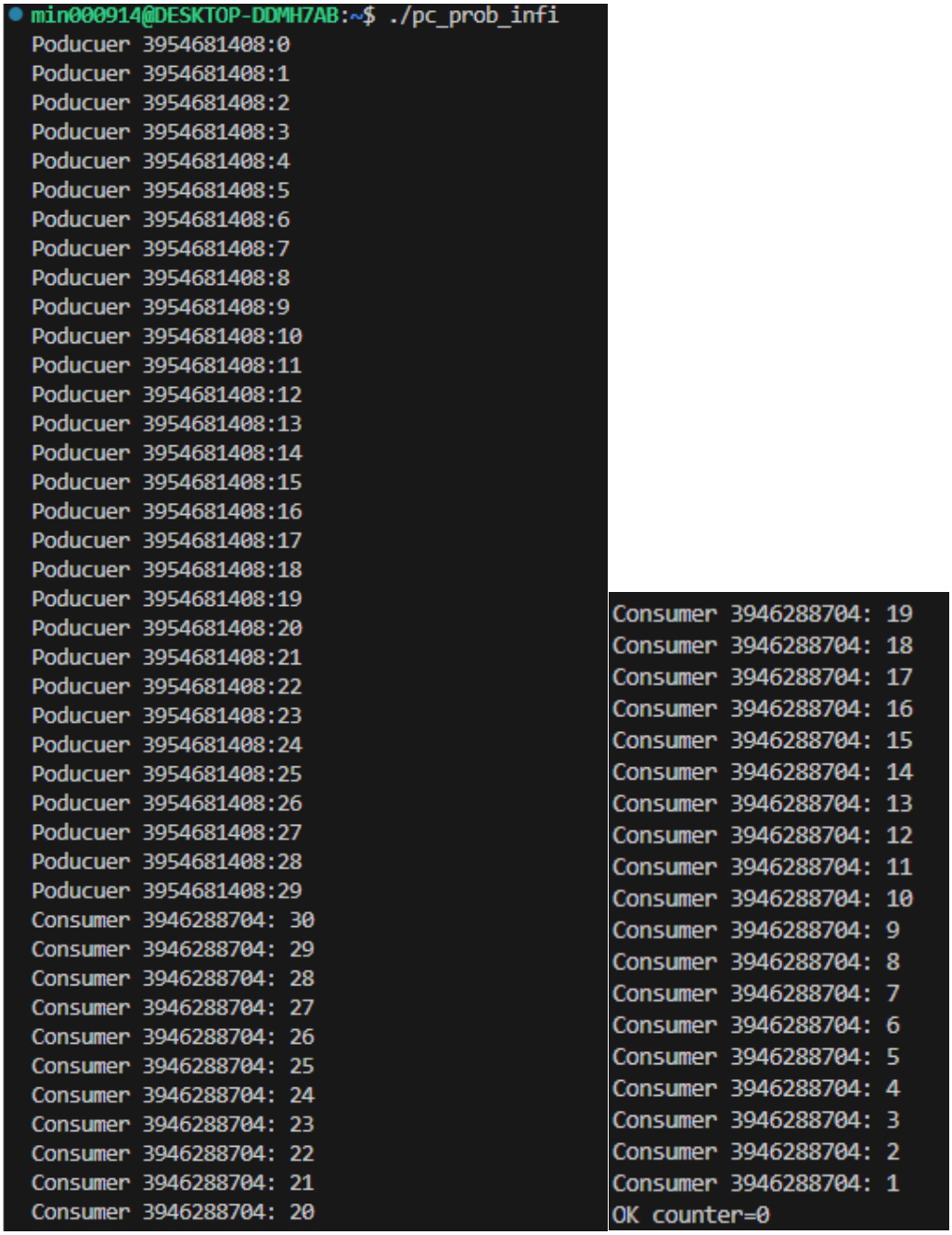
```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define ITER 30//반복횟수

void* thread_p(void* arg);
void* thread_c(void* arg);

int x = 0;
sem_t m;;//전역변수 x에 대한 상호배제를 위한 Sem
sem_t fill;;//Buffer가 비었을 경우 소비자의 소비를 막기위한 CV

int main() {
    pthread_t tid1, tid2;
    sem_init(&m, 0, 1);
    sem_init(&fill, 0, 0);

    pthread_create(&tid1, NULL, thread_p, NULL);
    pthread_create(&tid2, NULL, thread_c, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (x != 0)
        printf("BOOM! counter=%d\n", x);
    else
        printf("OK counter=%d\n", x);

    sem_destroy(&m);
    sem_destroy(&fill);

    return 0;
}

void* thread_p(void* arg) {//생산자
    int i, val;
    for (i = 0; i < ITER; i++) {
        sem_wait(&m);

        val = x;
        printf("Poducuer %u:%d\n", (unsigned int)pthread_self(), x);
        x = val + 1;
```

```
        sem_post(&m);
        sem_post(&fill);
    }

    pthread_exit(NULL);
}

void* thread_c(void* arg) {//소비자
    int i, val;
    for (i = 0; i < ITER; i++) {
        sem_wait(&fill);
        sem_wait(&m);

        val = x;
        printf("Consumer %u: %d\n", (unsigned int)pthread_self(), val);
        x = val - 1;

        sem_post(&m);
    }

    pthread_exit(NULL);
}
```

Infinite Buffer에서는 Producer가 더 이상 Buffer의 MaxSize를 고려하지 않고 무한정 생산할 수 있기 때문에 Buffer Size를 고려하는 CV를 위한 세마포어 empty를 지운 코드이다. Consumer에서는 여전히 Buffer가 비었다면 소비할 수 없다는 제약이 남아 있으므로, 세마포어 fill을 남겼다.

Producer는 x에대한 상호배제만 된다면, 무한정 생산 가능하며, 생산마다 Buffer의 재고를 나타내는 fill을 증가시킨다.

Consumer는 여전히 Buffer가 Empty라면 소비할 수 없다는 제약을 fill을 통해 처리하고, x에대한 상호배제까지 된다면, 소비할 수 있게 하였다.



Bounded Buffer일때와는 달리 Producer가 Produer의 최대 반복횟수인 30회까지 생산할 수 있는 것을 확인할 수 있다. 반면, Consumer는 여전히 제약이 걸려있어 0보다 적을 경우 더 이상 수행될 수 없다.

2.Readers - Writers Problem

Readers와 Writers는 Reader에 우선권을 주는 방식과, Writer에 우선권을 주는 방식으로 문제를 해결하였다.

다음은 Reader에 우선권이 있는 Readers – Writers Problem코드이다. (Writer가 변수(readThing)를 변경할 때는 readThing++로 처리해주었다. 각각 Writer Thread 3개, Read Thread 3개를 생성해 주었다. Readers와 Writers 각각 15번씩 실행되도록 하였다.)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define ITER 15//반복횟수
#define R_NUM 3//Reader 수
#define W_NUM 3
void *thread_writer(void *arg);
void *thread_reader(void *arg);

int readThing = 0;//Writer가 변경하고 Reader 읽을 변수
sem_t x; //Reader 에서 변경되는 readCount를 상호배제하기 위한 Sem
sem_t wsem; //Writer 혹은 Reader가 수행중일 때 다른 Writer의 접근을 막기위한 CV를 위한 Sem
int readCount = 0;//현재 Reading 중인 Reader 수

int main()
{
    pthread_t write_tid[W_NUM];
    pthread_t reader_tid[R_NUM];
    sem_init(&x, 0, 1);
    sem_init(&wsem, 0, 1);

    for (unsigned int i = 0; i < R_NUM; i++)//3개의 Read Thread를 생성한다.
    {
        pthread_create(&reader_tid[i], NULL, thread_reader, (void*)(i+1));
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
        pthread_create(&write_tid[i], NULL, thread_writer, (void*)(i+1));
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
```

```
        pthread_join(reader_tid[i], NULL);
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
        pthread_join(write_tid[i], NULL);
    }
    sem_destroy(&x);
    sem_destroy(&wsem);
    return 0;
}

void *thread_writer(void *arg) //Writer
{
    int i;
    int writeNum= (unsigned long int)arg;//자신이 몇번째 Write 인지 저장
    for (i = 0; i < ITER; i++)
    {
        sem_wait(&wsem);//다른 Writer 가 수행중이거나, Read 가 수행중이라면
Lock 이걸린다.

        printf("%d WRITE%dth  %u:%d\n",writeNum, i, (unsigned
int)pthread_self(), readThing++);

        sem_post(&wsem);
    }

    pthread_exit(NULL);
}

void *thread_reader(void *arg)
{
    int i;
    int readerNum=(unsigned long int)arg;
    for (i = 0; i < ITER; i++)
    {
        sem_wait(&x);//ReadCount 상호배제
        readCount++;
        if (readCount == 1) //ReadCount 가 1 이라면,
            sem_wait(&wsem); //Writer 가 wsem 을 잡고있다면 멈추고 아니라면
Writer 가 잡지 못하게 Lock 을 건다.
        sem_post(&x);

        printf("%d READ %dth  %u:%d\n", readerNum, i, (unsigned
int)pthread_self(), readThing);

        sem_wait(&x);
        readCount--;
        if (readCount == 0)//Read 중인 모든 Reader 가 빠져나갔다면
```

```
            sem_post(&wsem);//그 때 Writer의 Lock을 풀어준다.
        sem_post(&x);
    }

    pthread_exit(NULL);
}
```

wsem는 Writer Thread에서 Read 혹은 다른 Write가 수행될 때 Writer가 수행되지 못하도록 하고, Writer가 수행중일 때 다른 Read혹은 Writer가 접근하지 못하도록 하기 위한 세마포어 변수이다.

Reader에서 readCount는 현재 Reading중인 ReadThread의 개수를 Counting 해주는 것이고, 이는 전역변수이므로 세마포어 x로 상호배제한다. 한 개의 Read가 들어왔을 때 wsem을 잡아 Writer가 더 이상 접근하지 못하도록하며, ReadCount가 0이되었을 때 즉, Reading중이던 다수의 Reader들의 실행이 모두 종료되었을 때 비로소, wsem을 UnLock하여, Writer가 접근할 수 있도록 하여 Reader에게 Preference를 부여한다..

실행 예시는 다음과 같다.

몇번째Read인지 / Read / 몇번째수행인지 / Thread Id: readThing 값

몇번째WRTIRE인지 / WRTIRE / 몇번째수행인지 / Thread Id: readThing 값

```
min000914@DESKTOP-DDMH7AB:~$ ./rw_prob_r
1 READ 0th  2660865600:0
1 READ 1th  2660865600:0
2 READ 0th  2652472896:0
2 READ 1th  2652472896:0
2 READ 2th  2652472896:0
2 READ 3th  2652472896:0
2 READ 4th  2652472896:0
2 READ 5th  2652472896:0
2 READ 6th  2652472896:0
2 READ 7th  2652472896:0
2 READ 8th  2652472896:0
2 READ 9th  2652472896:0
2 READ 10th  2652472896:0
2 READ 11th  2652472896:0
2 READ 12th  2652472896:0
2 READ 13th  2652472896:0
2 READ 14th  2652472896:0
1 READ 2th  2660865600:0
1 READ 3th  2660865600:0
1 READ 4th  2660865600:0
1 READ 5th  2660865600:0
1 READ 6th  2660865600:0
1 READ 7th  2660865600:0
1 READ 8th  2660865600:0
1 READ 9th  2660865600:0
1 READ 10th  2660865600:0
1 READ 11th  2660865600:0
1 READ 12th  2660865600:0
1 READ 13th  2660865600:0
1 READ 14th  2660865600:0
3 READ 0th  2644080192:0
3 READ 1th  2644080192:0
3 READ 2th  2644080192:0
3 READ 3th  2644080192:0
3 READ 4th  2644080192:0
3 READ 5th  2644080192:0
1 WRITE0th  2635687488:0
1 WRITE1th  2635687488:1
1 WRITE2th  2635687488:2
1 WRITE3th  2635687488:3
3 READ 6th  2644080192:4
3 READ 7th  2644080192:4
3 READ 8th  2644080192:4
3 READ 9th  2644080192:4
3 READ 10th  2644080192:4
3 WRITE0th  2550134336:4
1 WRITE4th  2635687488:5
1 WRITE5th  2635687488:6
1 WRITE6th  2635687488:7
```

```
1 WRITE7th  2635687488:8
1 WRITE8th  2635687488:9
3 READ 11th  2644080192:10
1 WRITE9th  2635687488:10
1 WRITE10th  2635687488:11
1 WRITE11th  2635687488:12
1 WRITE12th  2635687488:13
1 WRITE13th  2635687488:14
1 WRITE14th  2635687488:15
3 WRITE1th  2550134336:16
3 WRITE2th  2550134336:17
3 WRITE3th  2550134336:18
3 WRITE4th  2550134336:19
3 WRITE5th  2550134336:20
3 WRITE6th  2550134336:21
3 WRITE7th  2550134336:22
3 WRITE8th  2550134336:23
3 WRITE9th  2550134336:24
3 WRITE10th  2550134336:25
3 WRITE11th  2550134336:26
3 WRITE12th  2550134336:27
3 READ 12th  2644080192:28
3 READ 13th  2644080192:28
3 WRITE13th  2550134336:28
3 WRITE14th  2550134336:29
3 READ 14th  2644080192:30
2 WRITE0th  2627294784:30
2 WRITE1th  2627294784:31
2 WRITE2th  2627294784:32
2 WRITE3th  2627294784:33
2 WRITE4th  2627294784:34
2 WRITE5th  2627294784:35
2 WRITE6th  2627294784:36
2 WRITE7th  2627294784:37
2 WRITE8th  2627294784:38
2 WRITE9th  2627294784:39
2 WRITE10th  2627294784:40
2 WRITE11th  2627294784:41
2 WRITE12th  2627294784:42
2 WRITE13th  2627294784:43
2 WRITE14th  2627294784:44
```

Multiple한 Read의 접근이 가능하고, Write의 접근은 후순위로 밀린 상황이다. Read Thread가 한 개 남을 때까지 Write Thread는 거의 접근할 수 없게 된다.

다음은 Writer에 우선권이 있는 Readers – Writers Problem코드이다. (Writer가 변수(readThing)를 변경할 때는 readThing++로 처리해주었다. 각각 Writer Thread 3개, Read Thread 3개를 생성해 주었다. Readers와 Writers 각각 15번씩 실행되도록 하였다.)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define ITER 15//반복횟수
#define R_NUM 3//Reader 수
#define W_NUM 3
void *thread_writer(void *arg);
void *thread_reader(void *arg);

int readThing = 0;//Writer가 변경하고 Reader 읽을 변수
sem_t x;//Reader에서 변경되는 readCount를 상호배제하기 위한 Sem
sem_t y;//Writer에서 변경되는 WriteCount를 상호배제하기 위한 Sem
sem_t z;//후술.
sem_t rsem;//대기 중이던 Writer가 모두 수행을 끝낼 때까지 Reader의 접근을 막는 Sem(WirterPreference)
sem_t wsem;//Writer 혹은 Reader가 수행중일 때 다른 Writer의 접근을 막기위한 CV를 위한 Sem
int readCount = 0;//현재 Reading 중인 Reader 수
int writeCount = 0;//현재 Write 실행을 위해 대기중인 Writer 수


int main()
{
    pthread_t write_tid[W_NUM];
    pthread_t reader_tid[R_NUM];
    sem_init(&x, 0, 1);
    sem_init(&y, 0, 1);
    sem_init(&z, 0, 1);
    sem_init(&rsem, 0, 1);
    sem_init(&wsem, 0, 1);
```

```
    for (unsigned int i = 0; i < R_NUM; i++)//3 개의 Read Thread 를 생성한다.
    {
        pthread_create(&reader_tid[i], NULL, thread_reader, (void*)(i+1));
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
        pthread_create(&write_tid[i], NULL, thread_writer, (void*)(i+1));
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
        pthread_join(reader_tid[i], NULL);
    }
    for (unsigned int i = 0; i < R_NUM; i++)
    {
        pthread_join(write_tid[i], NULL);
    }
    sem_destroy(&x);
    sem_destroy(&y);
    sem_destroy(&z);
    sem_destroy(&rsem);
    sem_destroy(&wsem);
    return 0;
}

void *thread_writer(void *arg)//Writer
{
    int i;
    int writeNum= (unsigned long int)arg;//자신이 몇번째 Write 인지 저장
    for (i = 0; i < ITER; i++) {
        sem_wait(&y);//WriterCount 를 위한 상호배제
        writeCount++;
        if (writeCount == 1)
            sem_wait(&rsem);//Writer 에서 rsem 을 잡고 있다면(대기중인 Wirter 가
있다면) Read 를 수행할 수 없게한다.
        sem_post(&y);
        sem_wait(&wsem);//다른 Writer 가 수행중이거나, Read 가 수행중이라면
Lock 이걸린다.

        printf("%d WRITE%dth  %u:%d\n",writeNum, i, (unsigned
int)pthread_self(), readThing++);

        sem_post(&wsem);
        sem_wait(&y);
        writeCount--;
        if (writeCount == 0)//Writer 가 다 빠져 나간다면, Read 가 접근할 수 있도록
rsem 을 풀어준다.
            sem_post(&rsem);
        sem_post(&y);
```

```
    }

    pthread_exit(NULL);
}

void *thread_reader(void *arg)
{
    int i;
    int readerNum= (unsigned long int)arg;
    for (i = 0; i < ITER; i++)
    {
        sem_wait(&z);//z가 없다면, rsem을 잡기위해 Writer에서는 하나의
Writer만이 접근할 수 있지만
        //Reader에서는 다수의 Reader가 rsem을 잡기위해 경쟁할 수 있다. 이를 막아
WriterPreference를 준다.
        sem_wait(&rsem);
        sem_wait(&x);
        readCount++;
        if (readCount == 1)
            sem_wait(&wsem);
        sem_post(&x);
        sem_post(&rsem);
        sem_post(&z);

        printf("%d READ %dth  %u:%d\n", readerNum, i, (unsigned
int)pthread_self(), readThing);

        sem_wait(&x);
        readCount--;
        if (readCount == 0)
            sem_post(&wsem);
        sem_post(&x);
    }

    pthread_exit(NULL);
}
}
```

WriterPreference를 위하여 writeCount 와 rsem, z를 추가하였다.

rsem을 통해 Writer가 rsem을 잡고 있다면, reader에서는 더 이상 변수에 접근할 수 없게 rsem을 통해 막힌다. 그리고 이 rsem은 대기중인 write의 수 writeCount가 0이 되었을 때만 풀린다.

또한 Reader와 Writer가 동시에 rsem을 잡기위해 대기중일 때 z가 없다면 Writer는 하나의 Thread만, Reader는 다수의 Reader가 동시에 Rsemd을 잡기위해 대기한다. 이렇게 되

면, Writer의 Preference를 해치게 되며 이를 막기위해 rsem에는 한 개의 Reader만 대기할 수 있게 z를 통해 제어하였다.

다음은 실행 예시이다.

```
min000914@DESKTOP-DDMH7AB:~$ ./rw_prob_w
 1 READ 0th  3963688512:0
 2 READ 0th  3955295808:0
 3 READ 0th  3946903104:0
 1 WRITE0th  3938510400:0
 1 WRITE1th  3938510400:1
 1 WRITE2th  3938510400:2
 3 WRITE0th  3921724992:3
 2 WRITE0th  3930117696:4
 2 WRITE1th  3930117696:5
 2 WRITE2th  3930117696:6
 2 WRITE3th  3930117696:7
 2 WRITE4th  3930117696:8
 2 WRITE5th  3930117696:9
 2 WRITE6th  3930117696:10
 2 WRITE7th  3930117696:11
 2 WRITE8th  3930117696:12
 2 WRITE9th  3930117696:13
 2 WRITE10th  3930117696:14
 1 WRITE3th  3938510400:15
 1 WRITE4th  3938510400:16
 1 WRITE5th  3938510400:17
 3 WRITE1th  3921724992:18
 3 WRITE2th  3921724992:19
 3 WRITE3th  3921724992:20
 3 WRITE4th  3921724992:21
 3 WRITE5th  3921724992:22
 3 WRITE6th  3921724992:23
 3 WRITE7th  3921724992:24
 2 WRITE11th  3930117696:25
 1 WRITE6th  3938510400:26
 1 WRITE7th  3938510400:27
 1 WRITE8th  3938510400:28
 1 WRITE9th  3938510400:29
 2 WRITE12th  3930117696:30
 2 WRITE13th  3930117696:31
 3 WRITE8th  3921724992:32
 3 WRITE9th  3921724992:33
 3 WRITE10th  3921724992:34
 3 WRITE11th  3921724992:35
 3 WRITE12th  3921724992:36
 3 WRITE13th  3921724992:37
 3 WRITE14th  3921724992:38
 2 WRITE14th  3930117696:39
```

```
1 READ 1th  3963688512:45
1 READ 2th  3963688512:45
3 READ 1th  3946903104:45
3 READ 2th  3946903104:45
3 READ 3th  3946903104:45
3 READ 4th  3946903104:45
3 READ 5th  3946903104:45
3 READ 6th  3946903104:45
3 READ 7th  3946903104:45
3 READ 8th  3946903104:45
3 READ 9th  3946903104:45
3 READ 10th  3946903104:45
3 READ 11th  3946903104:45
3 READ 12th  3946903104:45
3 READ 13th  3946903104:45
3 READ 14th  3946903104:45
1 READ 3th  3963688512:45
1 READ 4th  3963688512:45
1 READ 5th  3963688512:45
1 READ 6th  3963688512:45
1 READ 7th  3963688512:45
1 READ 8th  3963688512:45
1 READ 9th  3963688512:45
1 READ 10th  3963688512:45
1 READ 11th  3963688512:45
1 READ 12th  3963688512:45
1 READ 13th  3963688512:45
1 READ 14th  3963688512:45
2 READ 1th  3955295808:45
2 READ 2th  3955295808:45
2 READ 3th  3955295808:45
2 READ 4th  3955295808:45
2 READ 5th  3955295808:45
2 READ 6th  3955295808:45
2 READ 7th  3955295808:45
2 READ 8th  3955295808:45
2 READ 9th  3955295808:45
2 READ 10th  3955295808:45
2 READ 11th  3955295808:45
2 READ 12th  3955295808:45
2 READ 13th  3955295808:45
2 READ 14th  3955295808:45
```

다음과 같이 Reader Preference때와 비교해서 Writer의 접근이 용이해져 Writer의

Preference가 오른 것을 확인할 수 있다. 아까 Read에서와는 반대로 Writer Thread가 한 개 남을 때까지 Read가 거의 접근하지 못하는 것을 확인할 수 있다.

3.Dining Philosphers Problem

Dining Philosphers Problem은 DeadLock문제를 나타내는 전형적인 문제이다. DeadLock은 이미 Block되어 발생할 수 없는 Event를 기다리는 Process(혹은 Thread)에서 발생하는 문제로 이를 해결해주지 않는다면, 해당 Process가 더 이상 작업을 수행할 수 없게되며, 해당 Process가 소유하고 있는 공유자원마저 낭비되게 된다.

DeadLock이 발생하는 조건에는 4가지가 존재한다.

1. Mutual Exclusion(상호배제)
2. No Preemption(비선점)
3. Hold and Wait(점유와 대기): 공유자원이 추가로 필요할 때 자신이 갖고 있는 공유자원을 그대로 Hold한 상태로 Wait한다
4. Circular Wait(순환 대기): 여러 Process가 서로에게 필요한 자원을 들고 기다릴 때 발생한다.

이 4가지 조건 중 한가지만 해결해주어도 DeadLock은 발생하지 않으며 1번과 2번의 경우는 자원의 특성상 해결할 수 없는 경우가 많다.

그렇기 때문에 보통 3. Hold and Wait 혹은 4.Circular Wait을 해결하는 방식을 선택한다.

DeadLock을 Handling하는 방법에는 4가지가 존재한다.

1)DeadLock Preventation(예방)

DeadLock의 발생조건인 mutual exclusion, no preemption, hold and wait, circular wait이 일어나지 않도록 예방한다.

Cons: device utilization을 낮추거나, concurrency(동시성)을 낮추거나, 혹은 Resource사용을 어렵게 한다.

mutual exclusion, no preemption, hold and wait, circular wait 중 하나를 제약하면, DeadLock이 예방된다.

mutual exclusion, no preemption은 자원의 특성이라 제약하기 힘들다.

hold and wait를 제약: Request all resources. 프로세스가 한번에 모든 Resource를 Request하고, 모든 요청이 허용될 때까지 프로세스를 Block한다.

cons: 비효율적이고, Concurrency를 저하시킨다.

circular wait를 제약: Resource Ordering. resource에 numbering을 하고, 프로세스가 resource를 증가하는 순서로만 사용할 수 있게 한다.

cons: 모든 자원을 ordering하는데에 무리가 있고, limited incremental resource request(점진적인 자원요청에 따른 gain 포기 해야한다.)(B쓰고 A써야하는데 A쓰고 B써야하니까 효율떨어진다.)

2)DeadLock Avoidance(회피)

자원사용방법에 대한 제약이 아니라, resource allocation에 대한 시스템 State에 따라 request에 대해 승인 혹은 거절하여, DeadLock이 발생할 경우를 회피한다.

Cons: 추가적인 미래의 Resource에 대한 Request정보를 미리 알고 있어야한다.

3)DeadLock Detection and Recovery(검출 및 복구)

DeadLock 발생을 허용하며, 주기적으로 DeadLock을 Detection하고, Detect된다면, 해당 프로세스를 Recovery를 위하여 Killing 하거나, 자원을 회수한다.

Cons: 복구과정에 따른 Overhead가 발생한다.

4)Just Ignore the Problem(무시)

DeadLock 발생 가능성은 적은 데 비해 앞선 방법들의 OverHead는 높다. 그렇기 때문에 DeadLock을 무시한다. 단순한 Reboot으로 해결, 자원종류에 따라 해결한다.

다음은 **DeadLock Preventation** 방법 중 Request all resources 방식을 활용한 Code이다. 이 방식은 Hold and Wait을 막기 위한 방법으로 포크를 하나만 들고 있는 경우를 배제하기 위해 포크를 두개 다 들때까지 Scheduling되지 않도록 하기 위해 Once세마포어를 사용하였다. 5명의 철학자가 각각 Loop를 100번씩 수행하도록 하였다.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>

#define NUM 5 // 철학자 수
#define ITER 100// 한 철학자 당 Loop 횟수

pthread_mutex_t forks[NUM]; // forks
sem_t once; // 한번에 필요한 모든 fork 를 얻기 전까지 Scheduling 을 막는다.

void pickup(int philosopher_num)
{
    pthread_mutex_lock(&forks[philosopher_num % NUM]);
}

void putdown(int philosopher_num)
{
    pthread_mutex_unlock(&forks[philosopher_num % NUM]);
}

void thinking(int philosopher_num)
{
    printf("Philosopher %d is thinking\n", philosopher_num);
}

void eating(int philosopher_num)
```

```
{
    printf("Philosopher %d is eating\n", philosopher_num);
}

void *philosopher(void *arg)
{
    int philosopher_num = (unsigned long int)arg;
    int left_fork = philosopher_num;
    int right_fork = (philosopher_num + 1) % NUM;

    for (int i = 0; i < ITER; i++) {
        sem_wait(&once); // 필요한 모든 fork를 얻기 전까지 다른 철학자가 접근하지
못하도록한다.
        pickup(left_fork);
        printf("Philosopher %d picks up the fork %d.\n", philosopher_num,
left_fork);
        pickup(right_fork);
        printf("Philosopher %d picks up the fork %d.\n", philosopher_num,
right_fork);
        sem_post(&once); // 필요한 모든 fork를 얻었다면, once를 풀어준다.

        eating(philosopher_num);

        putdown(right_fork);
        printf("Philosopher %d puts down the fork %d.\n", philosopher_num,
right_fork);
        putdown(left_fork);
        printf("Philosopher %d puts down the fork %d.\n", philosopher_num,
left_fork);

        thinking(philosopher_num);
    }
    return NULL;
}


int main()
{
    pthread_t threads[NUM];
    clock_t start, end;
    double cpu_time_used;

    for (int i = 0; i < NUM; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }
    sem_init(&once, 0, 1);

    start = clock(); // 시작 시간 측정
```

```
    for (unsigned long int i = 0; i < NUM; i++) {
        pthread_create(&threads[i], NULL, philosopher, (void *)i);
    }

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

    end = clock(); // 종료 시간 측정
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC; // 수행 시간
계산

    printf("Total execution time: %f seconds\n", cpu_time_used);

    for (int i = 0; i < NUM; i++) {
        pthread_mutex_destroy(&forks[i]);
    }
    sem_destroy(&once);
    return 0;
}
```

clock함수를 통해 수행시간 또한 측정하였다.

```
min000914@DESKTOP-DDMH7AB:~$ ./dp_prob_1
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 4 picks up the fork 4.
Philosopher 4 picks up the fork 0.
Philosopher 3 puts down the fork 4.
Philosopher 1 picks up the fork 1.
Philosopher 1 picks up the fork 2.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking
Philosopher 3 picks up the fork 3.
Philosopher 4 is eating
Philosopher 4 puts down the fork 0.
Philosopher 1 is eating
Philosopher 1 puts down the fork 2.
Philosopher 1 puts down the fork 1.
Philosopher 1 is thinking
Philosopher 4 puts down the fork 4.
Philosopher 4 is thinking
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 1 picks up the fork 1.
Philosopher 1 picks up the fork 2.
Philosopher 1 is eating
Philosopher 1 puts down the fork 2.
Philosopher 1 puts down the fork 1.
Philosopher 1 is thinking
Philosopher 2 picks up the fork 2.
Philosopher 2 picks up the fork 3.
Philosopher 2 is eating
Philosopher 2 puts down the fork 3.
Philosopher 2 puts down the fork 2.
Philosopher 2 is thinking
Philosopher 4 picks up the fork 4.
Philosopher 4 picks up the fork 0.
Philosopher 4 is eating
```

```
Philosopher 4 picks up the fork 4.
Philosopher 4 picks up the fork 0.
Philosopher 4 is eating
Philosopher 4 puts down the fork 0.
Philosopher 4 puts down the fork 4.
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 0 picks up the fork 0.
Philosopher 4 is thinking
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking
Philosopher 4 picks up the fork 4.
Philosopher 4 picks up the fork 0.
Philosopher 4 is eating
Philosopher 4 puts down the fork 0.
Philosopher 4 puts down the fork 4.
Philosopher 4 is thinking
Philosopher 4 picks up the fork 4.
Philosopher 4 picks up the fork 0.
Philosopher 4 is eating
Philosopher 4 puts down the fork 0.
Philosopher 4 puts down the fork 4.
Philosopher 4 is thinking
Total execution time: 0.077236 seco
```

수행결과 전체는 아님을 밝힌다.

각 철학자들이 자신의 양쪽에 fork를 pickUp하는 상황에서는 Scheduling이 발생하지 않아, 모든 철학자들이 하나의 fork만을 잡고 대기하는 Hold and Wait상태에서 발생하는 DeadLock을 해결할 수 있다.

0.077236 Seconds로 수행되었다.

다음은 **DeadLock Avoidance** 방법 중 banker's Algorithme을 활용한 방식이다. 이 방식은 banker's Algorithme을 활용하여, thread를 실행하기 전, thread를 수행할만한 자원(fork)이 남아있는지(Safe) 확인한 후 남아있다면, thread를 수행하고 남아있지 않다면(UnSafe)다른 Thread가 자원을 반납할 때까지 대기하는 방식으로, DeadLock상황을 미리 검출하여 회피하는 방식을 통해 DeadLock을 해결한다. 5명의 철학자가 각각 Loop를 100번씩 수행하도록 하였다.

Safe상태를 확인하는 과정은 각 철학자Thread마다 실행전 Safe함수를 확인하여, 자신이 원하는 fork수에서 현재 가지고 있는 fork수를 빼준 fork수가 가용할 수 있는 fork수 보다 많다면, UnSafe상태이고, 그렇지 않다면 Safe상태이다.

UnSafe상태라면, pthread_cond_wait 함수를 이용해 Sleep상태로 들어가고, Safe상태인 Thread가 수행을 마친다면, 가용할 수 있는 자원의 수가 그만큼 증가했으므로 pthread_cond_broadcast를 통해 Sleep상태인 모든 Thread를 깨워 다시 Safe상태를 Check하도록한다.

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define NUM 5
#define ITER 100 // 한 철학자 당 Loop 횟수

sem_t forks[NUM]; // forks

int maximum[NUM][NUM] = { //각 철학자가 원하는 포크의 수
    {1, 1, 0, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 1, 1, 0},
    {0, 0, 0, 1, 1},
    {1, 0, 0, 0, 1}};
```

```
int allocated[NUM][NUM] = {0}; //각 철학자에게 할당된 포크의 수
int available[NUM] = {1, 1, 1, 1, 1}; //각 포크가 얼마나 남아있는지 확인
sem_t x; //전역변수인 allocated 와 available 변경 시 상호배제를 위한 sem
pthread_mutex_t mutex;//만약 UnSafe 상태라면, 해당 Thread 를 Block 시키고
pthread_cond_t cond;//Safe 상태인 Thread 의 수행을 마치면 BroadCast 를 통해
Block 된 모든 Thread 를 UnLock 한다.

void pickup(int philosopher_num, int fork)
{
    sem_wait(&forks[fork]);
    sem_wait(&x);//전역변수에 대한 상호배제
    allocated[philosopher_num][fork] = 1;//fork PickUp 시 본인에게 할당된
fork 수를 늘린다.
    available[fork] = 0;//남은 fork 수는 줄인다.
    sem_post(&x);
}

void putdown(int philosopher_num, int fork)
{
    sem_post(&forks[fork]);
    sem_wait(&x);//전역변수에 대한 상호배제
    allocated[philosopher_num][fork] = 0;//fork putdown 시 본인에게 할당된
fork 수를 0 으로 한다.
    available[fork] = 1;//남은 fork 수는 늘인다.
    sem_post(&x);
}

void thinking(int philosopher_num)
{
    printf("Philosopher %d is thinking.\n", philosopher_num);
    return;
}

void eating(int philosopher_num)
{
    printf("Philosopher %d is eating.\n", philosopher_num);
    return;
}

int safe(int philosopher_num)
{
    int need[NUM];//해당 철학자가 필요한 fork 수 를 저장
    int safe = 1;
    sem_wait(&x);//전역변수에 대한 상호배제
    for (int i = 0; i < NUM; i++)
    {
        //지금 필요한 fork 수 = 최대 필요한 fork 수 - 현재 가지고 있는 fork 수
        need[i] = maximum[philosopher_num][i] - allocated[philosopher_num][i];
```

```
    }
    for (int i = 0; i < NUM; i++)
    {
        if (available[i] < need[i])//가용가능한 fork 보다 많이 필요로하면
UnSafe 상태이다.
        {
            safe = 0;
            break;
        }
    }
    sem_post(&x);
    return safe;
}
void *philosopher(void *arg)
{
    int philosopher_num = (unsigned long int)arg;
    int left_fork = philosopher_num;
    int right_fork = (philosopher_num + 1) % NUM;

    for (int i = 0; i < ITER; i++)

    {
        pthread_mutex_lock(&mutex);
        while (!safe(philosopher_num))
        {
            pthread_cond_wait(&cond, &mutex);//UnSafe 한 상태라면 자러간다.
        }
        pthread_mutex_unlock(&mutex);

        pickup(philosopher_num, left_fork);
        printf("Philosopher %d picks up the fork %d.\n", philosopher_num,
left_fork);
        pickup(philosopher_num, right_fork);
        printf("Philosopher %d picks up the fork %d.\n", philosopher_num,
right_fork);

        eating(philosopher_num);

        putdown(philosopher_num, right_fork);
        printf("Philosopher %d puts down the fork %d.\n", philosopher_num,
right_fork);
        putdown(philosopher_num, left_fork);
        printf("Philosopher %d puts down the fork %d.\n", philosopher_num,
left_fork);

        thinking(philosopher_num);
```

```
        pthread_cond_broadcast(&cond);//Safe 상태가 되면, 자고 있는 모든 Thread 를
깨운다.
    }
    return NULL;

}

int main()
{
    clock_t start, end;//시간측정
    double cpu_time_used;//시간측정

    pthread_t threads[NUM];
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    for (int i = 0; i < NUM; i++)
    {
        sem_init(&forks[i], 0, 1);
    }
    sem_init(&x, 0, 1);

    start = clock(); //시간측정
    for (unsigned long int i = 0; i < NUM; i++)
    {
        pthread_create(&threads[i], NULL, philosopher, (void *)i);
    }

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

    end = clock(); // 종료 시간 측정
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC; // 수행 시간
계산

    printf("Total execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

다음은 수행예시이다.

```
min000914@DESKTOP-DDMH7AB:~$ ./dp_prob_2
Philosopher 0 picks up the fork 0.
Philosopher 1 picks up the fork 1.
Philosopher 2 picks up the fork 2.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 2 picks up the fork 3.
Philosopher 2 is eating.
Philosopher 2 puts down the fork 3.
Philosopher 2 puts down the fork 2.
Philosopher 2 is thinking.
Philosopher 1 picks up the fork 2.
Philosopher 1 is eating.
Philosopher 1 puts down the fork 2.
Philosopher 2 picks up the fork 2.
Philosopher 2 picks up the fork 3.
Philosopher 2 is eating.
Philosopher 2 puts down the fork 3.
Philosopher 2 puts down the fork 2.
Philosopher 2 is thinking.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 1 puts down the fork 1.
Philosopher 1 is thinking.
Philosopher 2 picks up the fork 2.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating.
Philosopher 0 puts down the fork 1.
Philosopher 0 puts down the fork 0.
Philosopher 0 is thinking.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 0 picks up the fork 0.
Philosopher 0 picks up the fork 1.
Philosopher 0 is eating.
```

```
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 3 picks up the fork 3.
Philosopher 3 picks up the fork 4.
Philosopher 3 is eating.
Philosopher 3 puts down the fork 4.
Philosopher 3 puts down the fork 3.
Philosopher 3 is thinking.
Philosopher 2 picks up the fork 2.
Philosopher 2 picks up the fork 3.
Philosopher 2 is eating.
Philosopher 2 puts down the fork 3.
Philosopher 2 puts down the fork 2.
Philosopher 2 is thinking.
Philosopher 2 picks up the fork 2.
Philosopher 2 picks up the fork 3.
Philosopher 2 is eating.
Philosopher 2 puts down the fork 3.
Philosopher 2 puts down the fork 2.
Philosopher 2 is thinking.
Total execution time: 0.090737 seconds
```

수행결과 전체는 아님을 밝힌다.

각 철학자들의 대한 수행을 하기 전 Safe함수를 통해 수행을 할 수 있는 상태인지 확인하여, 사전에 DeadLock을 회피함으로 DeadLock 상태를 해결하였다.

0.090737 Seconds로 수행되었다.

매 수행마다 Safe상태를 Check하는 Banker's Algorithm의 수행시간이 더 느릴것이라 예측했지고, 실제 실험에서는 평균적으로 Requst all Resources방식이 10%정도 빠르게 측정되었다.

Request All Reource 방식은 자원을 한 번에 모두 요청하고, 해당 자원을 사용할 수 있는지 여부를 확인한다. 이 방식은 Banker's와 비교하였을 때 간단하고 빠르지만, 현재의 상태를 고려하지 않기 때문에 안전성을 보장하지 않는다.

마지막으로 DeadLock Detect&Recovery를 구현해보았다. 나머지 부분은 기존 Dining Philosphers Problem와 동일하다. Dls_thread를 구현하여 일정시간마다 DeadLock을 Check한다. 이 때 pthread_mutex_trylock를 통해 모든 fork에 대한 Lock을 시도해보고 하나라도 Lock이된다면, DeadLock상태가 아닌것으로 판단하고, 모드 Lock이 될 수 없다면, 임시 DeadLock상태로 판단한다. 모두가 포크를 들고 있는 상황이 무조건 DeadLock인 것은 아니니 임시 DeadLock상태가 3번이상 검출된다면, DeadLock으로 간주하여 모든 fork를 풀어주는 방식을 채택하여 Recovery를 진행하였다.

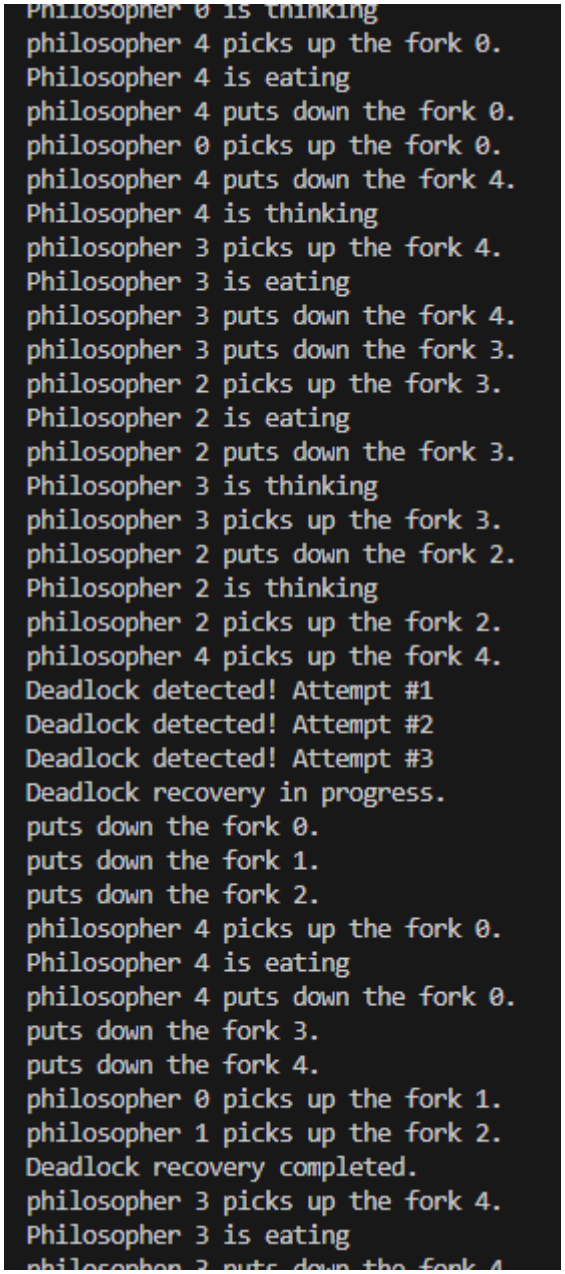
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <semaphore.h>

#define NUM 5

pthread_mutex_t forks[NUM]; // forks


void pickup(int philosopher_num)
{
    pthread_mutex_lock(&forks[philosopher_num % NUM]);
}
```

```c
void putdown(int philosopher_num)
{
    pthread_mutex_unlock(&forks[philosopher_num % NUM]);
}

void thinking(int philosopher_num)
{
    printf("Philosopher %d is thinking\n", philosopher_num);
    //usleep(rand() % 500000); // 임의의 대기 시간 추가
}

void eating(int philosopher_num)
{
    printf("Philosopher %d is eating\n", philosopher_num);
    //usleep(rand() % 500000); // 임의의 대기 시간 추가
}

void *philosopher(void *arg)
{
    int philosopher_num = (unsigned long int)arg;
    int left_fork = philosopher_num;
    int right_fork = (philosopher_num + 1) % NUM;

    while (1)
    {
        //sem_wait(&DeadLockChecking);
        pickup(left_fork);
        printf("philosopher %d picks up the fork %d.\n", philosopher_num,
left_fork);
        pickup(right_fork);
        printf("philosopher %d picks up the fork %d.\n", philosopher_num,
right_fork);
        eating(philosopher_num);
        putdown(right_fork);
        printf("philosopher %d puts down the fork %d.\n", philosopher_num,
right_fork);
        putdown(left_fork);
        printf("philosopher %d puts down the fork %d.\n", philosopher_num,
left_fork);

        thinking(philosopher_num);
        //sem_post(&DeadLockChecking);
    }
    return NULL;
}

void *dlc_thread(void *arg)//DeadLock 을 검출하는 Thread
```

```
{
    int deadlock_counter = 0;//상호배제가 제대로 이루어지지 않아 DeadLock 검출에
오류가 있다.
    //그렇기 때문에 DeadLock 검출이 3 번이상됐다면 DeadLock 으로 판단하도록한다.
    while (1)
    {
        sleep(2); // 2 초마다 Deadlock 을 체크

        // 모든 포크에 대해 시도해보고 획득할 수 없는 경우 Deadlock 으로 판단
        bool deadlock = true;
        for (int i = 0; i < NUM; i++)
        {
            while (pthread_mutex_trylock(&forks[i]) == 0)//모든 fork 의 mutex 를
Lock 해보고
            {
                pthread_mutex_unlock(&forks[i]);//하나라도 Lock 이 걸렸다면
                deadlock = false;//DeadLock 상태가 아니다.
                break;
            }
        }

        if (deadlock)
        {
            deadlock_counter++;
            printf("Deadlock detected! Attempt #%d\n", deadlock_counter);
            if (deadlock_counter >= 3)//DeadLock 이 3 번 검출됐다면
            {
                printf("Deadlock recovery in progress.\n");

                for (int i = 0; i < 5; i++)
                {
                    printf("puts down the fork %d.\n", i);//모든 fork 를
내려놓도록 한다.
                    putdown(i);
                }

                printf("Deadlock recovery completed.\n");
                deadlock_counter = 0;
            }
        }
        else
        {
            deadlock_counter = 0;
        }
    }
}

int main()
```

```
{
    pthread_t threads[NUM];
    pthread_t dlc;

    for (int i = 0; i < NUM; i++)
    {
        pthread_mutex_init(&forks[i], NULL);
    }
    for (unsigned long int i = 0; i < NUM; i++)
    {
        pthread_create(&threads[i], NULL, philosopher, (void *)i);
    }

    pthread_create(&dlc, NULL, dlc_thread, NULL);

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

    pthread_join(dlc, NULL);

    for (int i = 0; i < NUM; i++)
    {
        pthread_mutex_destroy(&forks[i]);
    }
    printf("NO DEADLOCK\n");
    return 0;
}
```

다음은 DeadLock검출 및 Recovery 예시이다.



구현이 어려울뿐더러, DeadLock을 탐지하고 회복하는 과정에서 큰 overHead가 발생할 수 있다는 것을 알았다.