
Assignment #2.

Synchronization and Memory



과 목 명 : 운영체제

담 당 교 수 : 박태준 교수님

제 출 일 : 2025. 06. 20.

학 과 : 지역·바이오시스템공학과

학 번 : 184128

이 름 : 박 지 환

#01. 생산자-소비자로 구성된 응용프로그램 만들기

- ❖ 기대 결과물 : 내용이 반영된 procon.c, 이에 대한 설명 또는 레포트
- ❖ 검색 키워드 : 생산자-소비자 문제

Step 1.

07_synchronization 강의자료에 소개된 생산자-소비자 문제 내용을 다시 한번 확인한다.

1-1. 생산자-소비자 문제(Producer-Consumer Problem)

- 공유 버퍼를 사이에 두고, 공유 버퍼에 데이터를 공급하는 생산자들과 데이터를 읽고 소비하는 소비자들이 공유 버퍼를 문제 없이 사용하도록 생산자와 소비자를 동기화시키는 문제입니다.
- 생산자는 수를 증가시켜가며 물건을 채우고 소비자는 생산자를 쫓아가며 물건을 소비합니다.
- 이때 생산자 코드와 소비자 코드가 동시에 실행되면 문제가 발생하며, 실행 순서에 따른 결과 차이 또한 존재합니다.

1-2. 생산자-소비자에서 고려해야 할 3가지 문제점

- 상호 배제 해결: 생산자들과 소비자들의 공유 버퍼에 대한 상호 배제로, 큐가 오염되서는 안된다는 의미입니다.
 - 비어 있는 공유 버퍼 문제: 비어 있는 공유 버퍼를 소비자가 읽으면 안됩니다.
 - 꽉 찬 공유 버퍼 문제: 꽉 찬 공유 버퍼에 생산자가 더 이상 데이터를 입력하면 안됩니다.
- ✓ 생산자(Producer)는 큐를 채우는 사람, 소비자(Consumer)는 큐를 소비하는 사람으로, 큐가 꽉 찼는데 생산해서도 안되고, 비어있는데 읽어들이면 안됩니다.

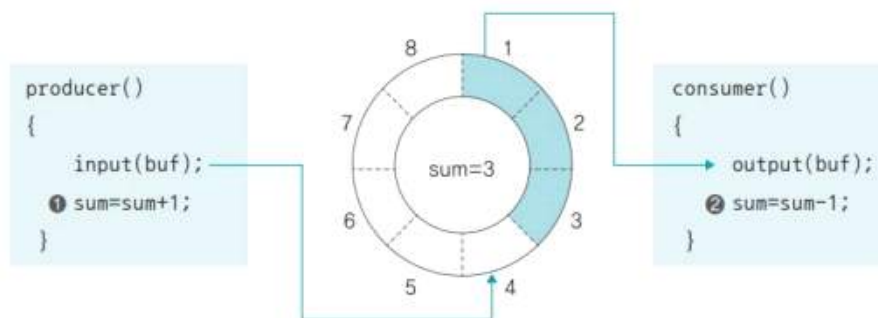


그림 1. 생산자-소비자 문제

1-3. 세마포어 2개를 이용하여 해결하기

- 읽기용 세마포어: 읽기 가능한 버퍼 개수를 확인하여 비어있는 버퍼 문제를 해결합니다.

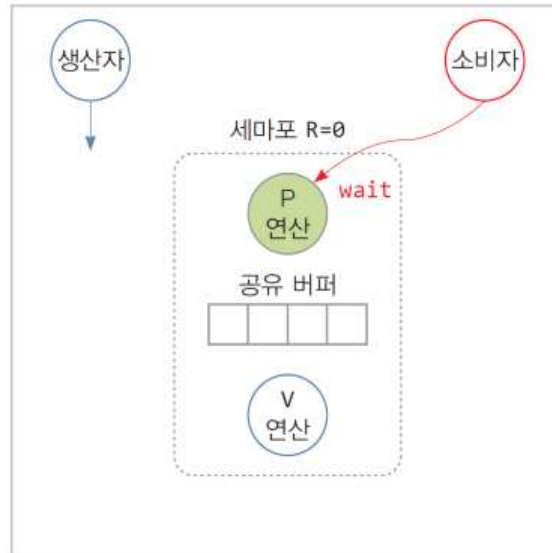


그림 2. 버퍼가 비어 있는 상태에서 소비자가 읽으려고 할 때

- ① 소비자가 버퍼에서 읽기 전 P 연산을 실행합니다.
- ② P 연산은 버퍼가 비어있는 경우($R=0$), 소비자가 읽을게 없기 때문에 소비자를 Sleep 상태로 Wait 시킵니다.

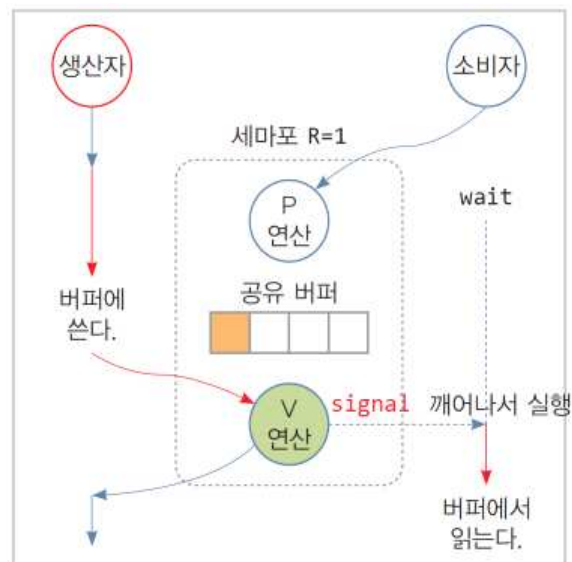


그림 3. 빈 버퍼에 생산자가 쓸 때

- ③ 생산자는 버퍼에 데이터를 기록하고 V 연산을 실행합니다.
- ④ V 연산은 세마포어 변수 R을 1 증가시키고($R=1$), 대기 중인 소비자에 Signal을 주어 깨웁니다.
- ⑤ 소비자가 깨어나면 P 연산을 마치고 공유 버퍼에서 읽고 P 연산에서 세마포어 변수 R을 1 감소 시킵니다($R=0$).

- 쓰기용 세마포어: 쓰기 가능한 버퍼 개수를 확인하여 가득 찬 버퍼 문제를 해결합니다.

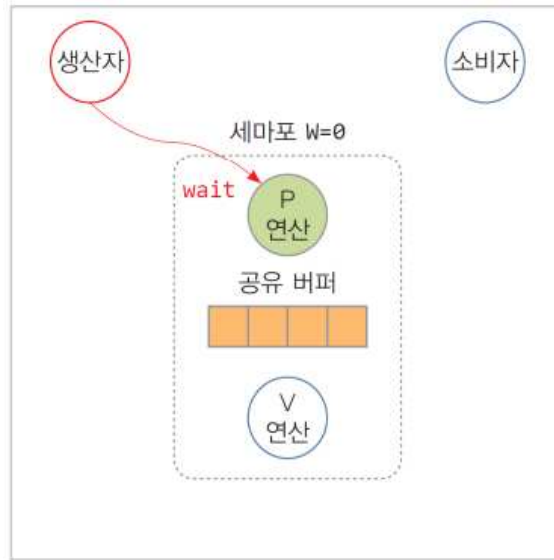


그림 4. 공유 버퍼가 찬 상태에서 생산자가 쓰려고 할 때

- ① 생산자가 버퍼에 쓰기 전 P 연산을 실행합니다.
- ② P 연산은 버퍼가 꽉 찼을 경우($W=0$), 생산자가 만들게 없기 때문에 생산자를 Sleep 상태로 Wait 시킵니다.

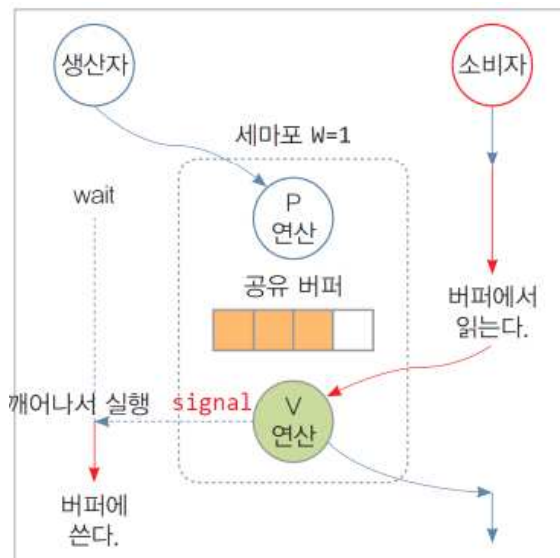


그림 5. 버퍼에 데이터가 있는 경우 소비자가 읽을 때

- ③ 소비자는 버퍼에서 데이터를 읽은 후 V 연산을 실행합니다.
- ④ V 연산은 세마포어 변수 W 를 1 증가시키고($W=1$), 대기 중인 생산자에 Signal을 주어 깨웁니다.
- ⑤ 생산자가 깨어나면 P 연산을 마치고 공유 버퍼에 쓰고 P 연산에서 세마포어 변수 W 를 1 감소 시킵니다($W=0$).

1-4. 알고리즘으로 표현하기

```

Producer { // 생산자 스레드
while(true) {
    P(W); // 세마포 W에 P/wait 연산을 수행하여
        // 버퍼가 꽉 차 있으면(쓰기 가능한 버퍼 수=0) 대기

    뮉텍스(M)를 잠근다.
    공유버퍼에 데이터를 저장한다. // 임계구역 코드
    뮉텍스(M)를 연다.

    V(R); // 세마포 R에 대해 V/signal 연산을 수행하여
        // 버퍼에 데이터가 저장되기를 기다리는 Consumer를 깨
    움.
    }
}
    
```

그림 6. 생산자 스레드

```

Consumer { // 소비자 스레드
while(true) {
    P(R); // 세마포 R에 P/wait 연산을 수행하여
        // 버퍼가 비어 있으면(읽기 가능한 버퍼 수=0) 대기한
    다.

    뮉텍스(M)를 잠근다.
    공유버퍼에서 데이터를 읽는다. // 임계구역 코드
    뮉텍스(M)를 연다.

    V(W); // 세마포 W에 대해 V/signal 연산을 수행하여
        // 버퍼가 비기를 기다리는 Producer를 깨움
    }
}
    
```

그림 7. 소비자 스레드

- R : 버퍼에 읽기 가능한 버퍼의 개수로, 비어 있는 경우(R=0) 대기합니다.
- W : 버퍼에 있는 쓰기 가능한 버퍼의 개수로, 꽉 차있는 경우 대기합니다.
- M : 생산자와 소비자 모두 사용하는 뮉텍스입니다.

Step 2.

주어진 조건에 맞는 간단한 생성자-소비자 문제 애플리케이션을 만들어본다.

목표 : 1개의 생산자 쓰레드와 1개의 소비자 쓰레드로 구성된 애플리케이션 작성

기존 procon.c 파일의 [Write here] 부분을 수정하여 목표에 해당하는 생성자-소비자 문제 애플리케이션을 완성하였습니다. 전체 코드에 대해서는 Step 3에서 다룰 예정입니다.

2-1. 생산자 쓰레드 조건

- 0부터 9까지 10개의 정수를 랜덤한 시간 간격으로 공유 버퍼에 씁니다.
- 공유 버퍼에 실제로 쓰는 부분은 mywrite() 함수로 세마포어 및 뮷텍스를 활용해 버퍼에 데이터를 기록합니다.

```
// producer thread function
void* producer(void* arg) {
    for(int i=0; i<10; i++) {
        mywrite(i); // write i into the shared memory
        printf("producer : wrote %d\n", i);

        // sleep m milliseconds
        int m = rand() % 10;
        usleep(MILLI * m * 10); // m*10 milliseconds
    }
    return NULL;
}
```

그림 8. procon.c의 producer() 함수

- ✓ for(int i = 0; i < 10; i++) : 0부터 9까지의 총 10개의 정수를 생성합니다.
- ✓ mywrite(i); : 생성한 값을 mywrite() 함수를 통해 공유 버퍼에 저장합니다.
- ✓ int m = rand() % 10; 및 usleep(MILLI * m * 10); : 0~90ms 사이에서 랜덤으로 대기한다. 이때 usleep()의 단위는 마이크로초로 지정되었습니다.
- ✓ 생산자 쓰레드가 0~9의 10개의 정수를 랜덤한 시간 간격으로 공유 버퍼에 쓰는 로직은 producer() 함수 내 루프 → mywrite() 호출 → 랜덤 Sleep 와 같이 구현됩니다.

2-2. 소비자 쓰레드 조건

- 공유 버퍼로부터 랜덤한 시간 간격으로 10개의 정수를 읽어 출력합니다.
- myread() 함수 내에서 공유 버퍼로부터 값을 읽어옵니다.

```
// consumer thread function
void* consumer(void* arg) {
    for(int i=0; i<10; i++) {
        int n = myread(); // read a value from the shared memory
        printf("\tconsumer : read %d\n", n);

        // sleep m milliseconds
        int m = rand() % 10;
        usleep(MILLI * m * 10); // m*10 milliseconds
    }
    return NULL;
}
```

그림 9. procon.c의 consumer() 함수

- ✓ int n = myread(); : 공유 버퍼로부터 1개의 값을 읽어옵니다.
- ✓ printf("consumer : read %d\n", n); : 읽은 값을 출력합니다.
- ✓ usleep(MILLI * m * 10); : 0~90ms 사이에서 랜덤으로 대기합니다.
- ✓ for (int i = 0; i < 10; i++) : 총 10번의 루프를 돌면서 10개의 정수를 읽습니다.
- ✓ 소비자 쓰레드가 공유 버퍼에서 10개의 정수를 랜덤한 시간 간격으로 읽고 출력하는 로직은 consumer() 함수와 함수 내에서 호출되는 myread() 함수에 구현되어 있습니다.

2-3. 공유 버퍼 조건

- 4개의 정수를 저장하는 큐로 작성되었고, 해당 큐는 배열로 작성되어 있습니다.

```
#define N_COUNTER 4 // the size of a shared buffer
#define MILLI 1000 // time scale

void mywrite(int n);
int myread();

pthread_mutex_t critical_section; // POSIX mutex
sem_t semWrite, semRead; // POSIX semaphore
int queue[N_COUNTER]; // shared buffer
int wptr = 0; // write pointer for queue[]
int rptr = 0; // read pointer for queue[]
```

그림 10. 4개의 정수를 저장하고 배열로 정의한 공유 버퍼
및 생산자/소비자가 데이터를 읽고 쓰는 위치(포인터)

- ✓ queue[N_COUNTER] : 길이가 4인 정수 배열로 공유 버퍼 역할을 합니다.
- ✓ wptr : 생산자가 데이터를 쓸 위치를 나타냅니다.
- ✓ rptr : 소비자가 데이터를 읽을 위치를 나타냅니다.

- 큐는 마지막 요소를 넘어가지 않고 0번으로 되돌아가는 원형 구조로 되어있습니다.

```
queue[wptr] = n;
wptr = (wptr + 1) % N_COUNTER;
```

그림 11. mywrite() 함수에서 생산자가 버퍼에 쓰는 위치

```
int n = queue[rptr];
rptr = (rptr + 1) % N_COUNTER;
```

그림 12. myread() 함수에서 소비자가 버퍼에서 읽는 위치

- ✓ wptr = (wptr + 1) % N_COUNTER; 및 rptr = (rptr + 1) % N_COUNTER; : 배열 인덱스를 원형처럼 순환 처리합니다.
- ✓ 공유 버퍼는 전역 배열 queue[4]로 선언되어 있고, wptr 및 rptr에 % N_COUNTER 연산을 적용함으로써 원형 큐 구조를 만족하고 있습니다.

2-4. 2개의 세마포어 사용

- 2개의 세마포어(semWrite, semRead)를 사용하여 공유 버퍼에 쓰기 가능한 공간 및 읽기 가능한 공간의 개수를 나타냅니다.

```
sem_t semWrite, semRead; // POSIX semaphore
```

그림 13. 전역 변수로 두 개의 세마포어 선언

```
// init semaphore
sem_init(&semWrite, 0, N_COUNTER); // initially all buffer slots are writable
sem_init(&semRead, 0, 0);           // initially no readable data
```

그림 14. main() 함수에서 세마포어 초기화

- ✓ semWrite : 공유 버퍼에 쓰기 가능한 공간을 의미하며, 버퍼 크기(4)만큼 초기화됩니다.
- ✓ semRead: 공유 버퍼에 읽을 수 있는 데이터 수를 의미하며, 처음에는 생산된 데이터가 없기 때문에 초기값이 0으로 지정됩니다.

- 세마포어를 사용하는 위치(mywrite() 및 myread() 내부)는 다음과 같습니다.

```
// write n into the shared memory
void mywrite(int n) {
    sem_wait(&semWrite); // wait for available space
    pthread_mutex_lock(&critical_section); // enter critical section

    queue[wptr] = n;
    wptr = (wptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semRead); // signal that new data is available
}
```

그림 15. sem_wait(&semWrite)를 통해 쓰기 가능한 공간이 있을 때까지 대기 후 sem_post(&semRead)를 통해 데이터를 썼으니 읽을 수 있는 공간이 증가하도록 구현

```
// read a value from the shared memory
int myread() {
    sem_wait(&semRead); // wait for available data
    pthread_mutex_lock(&critical_section); // enter critical section

    int n = queue[rptr];
    rptr = (rptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semWrite); // signal that space is available

    return n;
}
```

그림 16. sem_wait(&semRead)를 통해 읽기 가능한 공간이 있을 때까지 대기 후 sem_post(&semWrite)를 통해 데이터를 읽었으니 버퍼 공간이 비어있게 되어 쓰기를 허용하도록 구현

2-5. 1개의 뮤텝스 사용

- 공유 버퍼에서 읽는 코드와 쓰는 코드를 **임계구역**으로 설정하고, 뮤텝스를 이용해 **상호배제**를 구현하는 코드 영역은 다음과 같습니다.

```
pthread_mutex_t critical_section; // POSIX mutex
```

그림 17. 하나의 뮤텝스를 전역 변수로 선언

```
int main() {
    pthread_t t[2]; // thread structure
    srand(time(NULL));

    pthread_mutex_init(&critical_section, NULL); // init mutex

    // init semaphore
    sem_init(&semWrite, 0, N_COUNTER); // initially all buffer slots are writable
    sem_init(&semRead, 0, 0);           // initially no readable data

    // create the threads for the producer and consumer
    pthread_create(&t[0], NULL, producer, NULL);
    pthread_create(&t[1], NULL, consumer, NULL);

    for(int i=0; i<2; i++)
        pthread_join(t[i], NULL); // wait for the threads

    // destroy the semaphores
    sem_destroy(&semWrite);
    sem_destroy(&semRead);

    pthread_mutex_destroy(&critical_section); // destroy mutex
    return 0;
}
```

그림 18. 뮤텝스 초기화 및 제거

```
// write n into the shared memory
void mywrite(int n) {
    sem_wait(&semWrite); // wait for available space
    pthread_mutex_lock(&critical_section); // enter critical section

    queue[wptr] = n;
    wptr = (wptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semRead); // signal that new data is available
}
```

그림 19. procon.c의 mywrite() 함수 내부

```
// read a value from the shared memory
int myread() {
    sem_wait(&semRead); // wait for available data
    pthread_mutex_lock(&critical_section); // enter critical section

    int n = queue[rptr];
    rptr = (rptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semWrite); // signal that space is available

    return n;
}
```

그림 20. procon.c의 myread() 함수 내부

- ✓ 공유 버퍼(Queue)에 접근하는 임계구역에는 pthread_mutex_lock()과 pthread_mutex_unlock을 통해 동시에 하나의 스레드만 진입할 수 있으며, 이로써 생산자와 소비자가 동시에 버퍼를 수정하는 경쟁 상태(Race Condition)를 방지합니다.

Step 3.

함께 제공된 procon.c 파일은 본 문제의 스켈레톤 코드이며,
[Write here] 부분을 중심으로 작성한다.

※ 자료로 제공된 스켈레톤 코드 procon.c의 [Write here] 부분에 작성한 코드입니다.

```
// write n into the shared memory
void mywrite(int n) {
    /* [Write here] */
}

// write a value from the shared memory
int myread() {
    /* [Write here] */
    return n;
}
```

그림 21. 수정 전 procon.c의 mywrite() 및 myread() 함수

```
// write n into the shared memory
void mywrite(int n) {
    sem_wait(&semWrite); // wait for available space
    pthread_mutex_lock(&critical_section); // enter critical section

    queue[wptr] = n;
    wptr = (wptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semRead); // signal that new data is available
}

// read a value from the shared memory
int myread() {
    sem_wait(&semRead); // wait for available data
    pthread_mutex_lock(&critical_section); // enter critical section

    int n = queue[rptr];
    rptr = (rptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semWrite); // signal that space is available

    return n;
}
```

그림 22. 수정 후 procon.c의 mywrite() 및 myread() 함수


```

int main() {
    pthread_t t[2]; // thread structure
    srand(time(NULL));

    pthread_mutex_init(&critical_section, NULL); // init mutex

    // init semaphore
    /* [Write here] */

    // create the threads for the producer and consumer
    pthread_create(&t[0], NULL, producer, NULL);
    pthread_create(&t[1], NULL, consumer, NULL);

    for(int i=0; i<2; i++)
        pthread_join(t[i], NULL); // wait for the threads

    //destroy the semaphores
    /* [Write here] */

    pthread_mutex_destroy(&critical_section); // destroy mutex
    return 0;
}

```

그림 23. 수정 전 procon.c의 main() 함수

```

int main() {
    pthread_t t[2]; // thread structure
    srand(time(NULL));

    pthread_mutex_init(&critical_section, NULL); // init mutex

    // init semaphore
    sem_init(&semWrite, 0, N_COUNTER); // initially all buffer slots
    sem_init(&semRead, 0, 0);          // initially no readable data

    // create the threads for the producer and consumer
    pthread_create(&t[0], NULL, producer, NULL);
    pthread_create(&t[1], NULL, consumer, NULL);

    for(int i=0; i<2; i++)
        pthread_join(t[i], NULL); // wait for the threads

    // destroy the semaphores
    sem_destroy(&semWrite);
    sem_destroy(&semRead);

    pthread_mutex_destroy(&critical_section); // destroy mutex
    return 0;
}

```

그림 24. 수정 후 procon.c의 main() 함수

```

int rptr = 0; // read pointer for queue[]

// producer thread function
void* producer(void* arg) {
    for(int i=0; i<10; i++) {
        mywrite(i); // write i into the shared memory
        printf("producer : wrote %d\n", i);

        // sleep n milliseconds
        int n = rand() % 10;
        usleep(MILLI * n * 10); // n*10 milliseconds
    }
    return NULL;
}

// consumer thread function
void* consumer(void* arg) {
    for(int i=0; i<10; i++) {
        int n = myread(); // read a value from the shared memory
        printf("\tconsumer : read %d\n", n);

        // sleep n milliseconds
        int n = rand() % 10;
        usleep(MILLI * n * 10); // n*10 milliseconds
    }
    return NULL;
}

// write n into the shared memory
void mywrite(int n) {
    sem_wait(&semWrite); // wait for available space
    pthread_mutex_lock(&critical_section); // enter critical section

    queue[wptr] = n;
    wptr = (wptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semRead); // signal that new data is available
}

// read a value from the shared memory
int myread() {
    sem_wait(&semRead); // wait for available data
    pthread_mutex_lock(&critical_section); // enter critical section

    int n = queue[rptr];
    rptr = (rptr + 1) % N_COUNTER;

    pthread_mutex_unlock(&critical_section); // leave critical section
    sem_post(&semWrite); // signal that space is available

    return n;
}

int main() {
    pthread_t t[2]; // thread structure
    srand(time(NULL));

    pthread_mutex_init(&critical_section, NULL); // init mutex

    // init semaphore
    sem_init(&semWrite, 0, N_COUNTER); // initially all buffer slots are writable
    sem_init(&semRead, 0, 0); // initially no readable data

    // create the threads for the producer and consumer
    pthread_create(&t[0], NULL, producer, NULL);
    pthread_create(&t[1], NULL, consumer, NULL);

    for(int i=0; i<2; i++)
        pthread_join(t[i], NULL); // wait for the threads

    // destroy the semaphores
    sem_destroy(&semWrite);
    sem_destroy(&semRead);

    pthread_mutex_destroy(&critical_section); // destroy mutex
    return 0;
}

```

그림 25. 완성된 procon.c의 전체 코드

※ 완성된 procon.c의 실행 결과입니다.

```
os@test03:~/Desktop/os2$ gcc -pthread procon.c -o procon
os@test03:~/Desktop/os2$ ./procon
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
        consumer : read 3
producer : wrote 4
        consumer : read 4
producer : wrote 5
        consumer : read 5
producer : wrote 6
        consumer : read 6
producer : wrote 7
        consumer : read 7
producer : wrote 8
        consumer : read 8
producer : wrote 9
        consumer : read 9
```

그림 26. gcc를 통해 procon.c 컴파일 후 실행한 결과

```
os@test03:~/Desktop/os2$ ./procon
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
        consumer : read 3
producer : wrote 4
        consumer : read 4
producer : wrote 5
        consumer : read 5
producer : wrote 6
        consumer : read 6
producer : wrote 7
        consumer : read 7
producer : wrote 8
        consumer : read 8
producer : wrote 9
        consumer : read 9
```

그림 27. 두 번째 실행 결과

```
os@test03:~/Desktop/os2$ ./procon
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
producer : wrote 3
producer : wrote 4
        consumer : read 2
producer : wrote 5
producer : wrote 6
        consumer : read 3
producer : wrote 7
        consumer : read 4
producer : wrote 8
        consumer : read 5
        consumer : read 6
producer : wrote 9
        consumer : read 7
        consumer : read 8
        consumer : read 9
```

그림 28. 세 번째 실행 결과

procon.c의 실행 결과와 같이 랜덤한 시간으로 작동되기 때문에 결과가 매 실행마다 생산 및 소비 순서가 조금씩 다른 모습을 보여주었습니다.

#02. 소프트웨어로 문을 만드는 방법

- ❖ 기대 결과물 : 소프트웨어 기반 동기화에 관한 내용과 내용이 반영된 procon2.c, 이에 대한 설명 또는 레포트
- ❖ 검색 키워드 : 운영체제 동기화 알고리즘 예제

Step 1.

소프트웨어 기반 동기화 방식(알고리즘)들에 대해 간단한 조사를 하고,
그에 대한 설명을 작성한다.

※ 참고 자료

1. Alagarsamy, K. "[Some myths about famous mutual exclusion algorithms.](#)" ACM SIGACT News 34, no. 3 (2003): 94-103.
2. Scherer, B. [Mutual Exclusion: Classical Algorithms for Locks.](#) Comp, 422, 1-59.
3. 쉽게 배우는 운영체제 (조성호 | 한빛아카데미)
4. [리눅스로 공부하는 운영체제](#) (이완주, 김경중 | 위키독스)

※ 소프트웨어적으로 동기화를 구현하는 방법(알고리즘) 비교

항목	Dekker	Peterson	Backery	Filter Lock
지원 프로세스 수	2	2	$N (N \geq 2)$	$N (N \geq 2)$
사용 변수	flag[2], turn	flag[2], turn	choosing[N], number[N]	level[N], victim[N-1]
상호 배제 보장	O	O	O	O
공정성(Fairness)	낮음	보장됨	보장됨	보장됨
기아 방지	조건부 가능	O	O	O
Busy-Wating	O	O	O	O
하드웨어 의존성	X	X	X	X
복잡도	보통	낮음	보통	보통~높음
특징	최초의 이론적 SW 상호 배제 방법	간결하고 직관적	번호표 기반 공정성 보장	Peterson을 다단계 구조로 일반화

1-1. 데커 알고리즘(Decker's Algorithm)

두 개의 쓰레드가 임계 영역을 안전하게 접근할 수 있도록 하는 상호 배제(Mutual Exclusion) 알고리즘입니다. 이 알고리즘은 각 쓰레드가 다른 쓰레드의 실행 상태를 확인하고 필요한 경우 다른 쓰레드가 임계 영역을 먼저 실행할 수 있도록 기다리게 하는 방식으로 동작합니다.

데커 알고리즘을 구현하기 위해 각 쓰레드는 다음 두 가지 정보를 유지합니다.

1. flag[] : 각 쓰레드가 임계 영역에 진입하고자 할 때, true로 설정되는 플래그 배열입니다. 쓰레드가 임계 영역에서 나올 때는 false로 설정됩니다.
2. turn : 어느 쓰레드의 차례인지를 나타내는 변수입니다. 이 변수는 어느 한 쓰레드가 임계 영역에 진입하는 것을 허용하고, 다른 쓰레드가 기다리게 합니다.

● 데커 알고리즘 기본 원리

1. 쓰레드는 임계 영역에 진입하기 전에 자신의 플래그를 true로 설정합니다.
2. 쓰레드는 turn 변수와 다른 쓰레드의 플래그 상태를 확인합니다. 만약 다른 쓰레드도 임계 영역에 진입하고자 하고 turn이 다른 쓰레드를 가리키고 있다면 현재 쓰레드는 대기합니다.
3. 임계 영역을 실행한 후, 쓰레드는 자신의 플래그를 false로 설정하여 다른 쓰레드가 진입할 수 있음을 알립니다.

```

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

#define TRUE 1
#define FALSE 0
#define N 2 // 스레드의 수

int flag[N];
int turn;
int counter = 0; // 공유 변수

void enter_critical_section(int thread_id) {
    flag[thread_id] = TRUE;
    turn = 1 - thread_id;
    while (flag[1 - thread_id] == TRUE && turn == 1 - thread_id);
}

void leave_critical_section(int thread_id) {
    flag[thread_id] = FALSE;
}

void* increment_counter(void* id) {
    int thread_id = *(int*)id;

    for (int i = 0; i < 1000000; i++) {
        enter_critical_section(thread_id);
        counter++;
        leave_critical_section(thread_id);
    }
    return NULL;
}

int main() {
    pthread_t threads[N];
    int thread_ids[N] = {0, 1};

    for (int i = 0; i < N; i++) {
        pthread_create(&threads[i], NULL, increment_counter, &thread_ids[i]);
    }

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

    printf("Final counter value: %d\n", counter);
    return 0;
}

```

그림 29. 데커 알고리즘을 사용한 예제

해당 예제에서는 두 스레드가 counter 변수를 각각 1,000,000번 증가시키려고 시도합니다.

데커 알고리즘을 사용하여 임계 영역에 안전하게 접근함으로써, 최종적으로 counter의 값이 예상대로 2,000,000이 되도록 합니다.

enter_critical_section과 leave_critical_section 함수는 각각 임계 영역에 진입하고 빠져나오는 로직을 구현합니다.

1-2. 피터슨 알고리즘(Peterson's Algorithm)

피터슨 알고리즘은 두 프로세스 또는 스레드가 임계 영역에 대한 상호 배제를 달성하기 위해 사용되는 소프트웨어 기반의 동기화 매커니즘입니다. 이 알고리즘은 두 개의 프로세스가 공유 자원을 안전하게 접근할 수 있도록 보장하며, 데커 알고리즘을 개선한 형태입니다.

피터슨 알고리즘은 다음과 같은 두 가지 변수를 사용합니다.

1. `flag[]` : 각 프로세스 또는 스레드가 임계 영역에 진입하고자 할 때 `true`로 설정하는 Boolean 배열입니다. 프로세스별 하나씩 총 두 개의 플래그가 있습니다.
2. `turn` : 어떤 프로세스의 차례인지를 나타내는 변수입니다. 이 변수는 두 프로세스 중 어느 한 쪽이 임계 영역에 진입할 권리가 있음을 나타냅니다.

피터슨 알고리즘은 다음 단계로 구성됩니다.

1. 프로세스는 자신의 플래그를 `true`로 설정하여 임계 영역에 진입하고자 함을 나타냅니다.
2. 프로세스는 `turn`을 상대방에게 넘겨 임계 영역에 진입할 준비가 되었으니 다른 이가 먼저 진입할 일이 있다면 하도록 합니다.
3. 프로세스는 다음 두 조건(상대방의 플래그가 `false`이거나, `turn`이 자신에게 있을 때)이 만족할 때까지 대기합니다.
4. 임계 영역에 진입한 후 프로세스는 작업을 수행합니다.
5. 임계 영역에서 작업을 마친 프로세스는 자신의 플래그를 `false`로 설정하여 임계 영역에서 나갔음을 나타냅니다.

```

#include <stdio.h>
#include <pthread.h>
#include <stdbool.h>

#define NUM_ITERATIONS 1000000

int counter = 0;
bool flag[2] = {false, false};
int turn;

void* incrementCounter(void* id) {
    int threadId = *(int*)id;

    for (int i = 0; i < NUM_ITERATIONS; i++) {
        // 임계 영역 진입 전
        flag[threadId] = true;
        turn = 1 - threadId;
        while (flag[1 - threadId] && turn == 1 - threadId) {
            // 대기
        }

        // 임계 영역
        counter++;

        // 임계 영역 탈출
        flag[threadId] = false;
    }

    return NULL;
}

int main() {
    pthread_t threads[2];
    int threadIds[2] = {0, 1};

    // 스레드 생성
    pthread_create(&threads[0], NULL, incrementCounter, &threadIds[0]);
    pthread_create(&threads[1], NULL, incrementCounter, &threadIds[1]);

    // 스레드가 종료될 때까지 대기
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);

    printf("Final counter value: %d\n", counter);

    return 0;
}

```

그림 30. 피터슨 알고리즘을 사용한 예제

해당 예제는 counter 변수를 두 스레드가 각각 1,000,000번씩 증가시키려고 시도합니다. 피터슨 알고리즘을 사용하여 두 스레드 사이의 상호 배제를 보장함으로써 counter의 최종 값이 정확히 2,000,000이 되도록 합니다.

1-3. 베이커리 알고리즘(Bakery Algorithm)

베이커리 알고리즘은 여러 프로세스가 임계 영역에 대한 접근을 제어하기 위한 알고리즘으로, 램포트의 빵집 알고리즘으로 알려져 있습니다. 이 알고리즘은 실제 베이커리에서 번호표를 뽑아 차례를 기다리는 방식을 모델로 합니다.

프로세스 또는 스레드가 임계 영역에 진입하려고 할 때 고유한 번호표를 받아 순서대로 접근할 수 있도록 합니다.

각 프로세스는 두 가지 주요 배열을 사용합니다.

1. `choosing[]` : 프로세스가 번호를 선택 중인지를 나타내는 배열입니다. 프로세스가 번호를 받으려고 할 때 이 배열을 사용하여 다른 프로세스가 알 수 있도록 합니다.
2. `number[]` : 각 프로세스에 할당된 고유 번호(순서)를 저장하는 배열입니다. 이 번호는 임계 영역에 진입할 순서를 결정하는 데 사용됩니다.

베이커리 알고리즘 작동 원리는 다음과 같습니다.

1. 프로세스가 임계 영역에 진입하려고 하면 먼저 `choosing[i]`를 `true`로 설정하여 번호를 선택하고 있음을 나타냅니다.
2. `number[i]`에 현재 대기 중인 프로세스 중 가장 높은 번호보다 1이 큰 값을 할당받습니다.
3. `choosing[i]`를 `false`로 설정하여 번호 선택을 완료했음을 나타냅니다.
4. 다른 모든 프로세스에 대해 `choosing[j]`가 `false`가 될 때까지 기다린 후, `number[j]`를 확인하여 자신의 차례가 될 때까지 대기합니다.
5. 임계 영역에 진입한 후 작업을 완료하면 `number[i]`를 0으로 초기화하여 임계 영역에서 나갔음을 나타냅니다.

Assignment #2. Synchronization and Memory

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

#define NUM_THREADS 2
#define NUM_ITERATIONS 1000

bool choosing[NUM_THREADS] = {false};
int number[NUM_THREADS] = {0};
int resource = 0; // 공유 자원

void enter_critical_section(int i) {
    choosing[i] = true;
    int max = 0;
    for (int j = 0; j < NUM_THREADS; j++) {
        if (number[j] > max) {
            max = number[j];
        }
    }
    number[i] = max + 1;
    choosing[i] = false;

    for (int j = 0; j < NUM_THREADS; j++) {
        while (choosing[j]);
        while (number[j] != 0 && (number[j] < number[i] || (number[j] == number[i] && j < i)));
    }
}

void leave_critical_section(int i) {
    number[i] = 0;
}

void* thread_function(void* arg) {
    int i = *(int*)arg;
    for (int k = 0; k < NUM_ITERATIONS; k++) {
        enter_critical_section(i);
        // 임계 영역 시작
        resource++;
        // 임계 영역 끝
        leave_critical_section(i);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS] = {0, 1};

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

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

    printf("Expected resource value: %d, Actual resource value: %d\n", NUM_THREADS * NUM_ITERATIONS, resource);
    return 0;
}
```

그림 31. 베이커리 알고리즘을 사용한 예제 1

해당 예제는 각 스레드가 공유 자원 resource에 NUM_ITERATIONS 만큼 접근하여 값을 증가시키는 과정을 보여줍니다. 베이커리 알고리즘을 사용하여 이러한 접근이 상호 배제되도록 함으로써 경쟁 조건 없이 안전하게 resource 값을 증가시킵니다.

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>

#define NUM_THREADS 2
#define NUM_ITERATIONS 1000000

// 베이커리 알고리즘을 위한 변수
bool entering[NUM_THREADS] = {false};
int ticket[NUM_THREADS] = {0};

int counter = 0; // 공유 변수

void lock(int thread_id) {
    // 번호표 발급 시작
    entering[thread_id] = true;
    int max_ticket = 0;
    for (int i = 0; i < NUM_THREADS; i++) {
        max_ticket = ticket[i] > max_ticket ? ticket[i] : max_ticket;
    }
    ticket[thread_id] = max_ticket + 1;
    entering[thread_id] = false;

    // 다른 모든 스레드를 확인
    for (int i = 0; i < NUM_THREADS; i++) {
        if (i == thread_id) continue;
        // 다른 스레드가 번호표를 받을 때까지 대기
        while (entering[i]) { /* busy wait */ }
        // 다른 스레드의 번호표가 현재 스레드보다 우선순위가 높은지 확인
        while (ticket[i] != 0 && (ticket[i] < ticket[thread_id] || (ticket[i] == ticket[thread_id] && i < thread_id))) {
            /* busy wait */
        }
    }
}

void unlock(int thread_id) {
    ticket[thread_id] = 0;
}

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        lock(thread_id);
        counter++;
        unlock(thread_id);
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS] = {0, 1};

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

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

    printf("Expected counter value: %d, Actual counter value: %d\n", NUM_ITERATIONS * NUM_THREADS, counter);
    return 0;
}

```

그림 32. 베이커리 알고리즘을 사용한 예제 2

해당 예제는 lock 함수에서 베이커리 알고리즘을 구현하여 임계 영역에 대한 상호 배제를 보장합니다. 각 스레드는 공유 변수 counter에 안전하게 접근하여 값을 증가시키며, unlock 함수를 통해 자신의 턴을 종료합니다. 이를 통해 모든 스레드가 공정하게 임계 영역에 접근할 수 있도록 하며 경쟁 조건 없이 예상대로 counter가 2,000,000으로 증가하는 것을 보장합니다.

1-4. 필터 락 알고리즘(Filter Lock Algorithm)

피터슨 알고리즘에서의 락은 두 개의 스레드 간 상호 배제만을 제공합니다. 하지만 필터 락 알고리즘은 락을 직접적으로 N개 이상의 스레드에 대해 상호 배제를 보장하기 위해 N-Way로 일반화한 방식입니다.

Filter Lock

```
class Filter: public Lock {
private:
    volatile int level[N]; volatile int victim[N-1];
public:
    void acquire() {
        for (int j = 1; j < N; j++) {
            level[self_threadid] = j;
            victim[j] = self_threadid;
            // wait while conflicts exist
            while (sameOrHigher(self_threadid, j) &&
                    victim[j] == self_threadid);
        }
    }
    bool sameOrHigher(int i, int j) {
        for(int k = 0; k < N; k++)
            if (k != i && level[k] >= j) return true;
        return false;
    }
    void release() {
        level[self_threadid] = 0;
    }
}
```

그림 33. 필터 락 알고리즘을 사용한 예제

피터슨 알고리즘에서의 락은 두 개의 Boolean 배열 flag[2]를 사용했지만, 필터 락은 이를 확장하여 N개의 정수 배열 level[N]을 사용합니다. 이때 level[k] 값은 스레드 k가 진입하려는 최고 수준(level)을 나타내며, 각 스레드는 N-1개의 수준을 통과해야 임계 영역에 진입할 수 있습니다.

각 레벨에는 victim 이라는 변수가 있어, 그 수준에서 1개의 스레드를 제외(Filter)시켜 다음 수준으로 가지 못하도록 하며 이는 피터슨 알고리즘의 victim 변수 개념을 자연스럽게 일반화한 것입니다.

필터 락 알고리즘에서의 수준(level)

1. 수준 k에 진입하려는 스레드가 하나 이상 있다면, 최소한 하나의 스레드는 진입에 성공합니다.
 2. 수준 k에 여러 스레드가 진입하려 한다면, 최소한 하나의 스레드는 차단(Blocked)됩니다.
- 이를 통해 상호 배제가 보장됩니다.

Step 2.

조사한 동기화 방식들 중 마음에 드는 한 가지를 골라 직접 구현해본다.

이때 고른 이유도 간단하게 설명한다.

조사한 여러 동기화 알고리즘 중 **필터 락 알고리즘**을 구현해 보았습니다. 이유는 다음과 같습니다.

1. 피터슨 알고리즘의 확장형 구조

피터슨 알고리즘은 오직 두 개의 쓰레드만을 동기화할 수 있고 그렇기 때문에 N개의 쓰레드가 동시에 임계 영역으로 접근하려는 상황에서는 사용할 수 없습니다. 하지만 필터 락 알고리즘은 피터슨 알고리즘의 구조와 아이디어를 다수의 쓰레드(N개)로 일반화한 방식으로, 여러 쓰레드가 동시에 임계 영역으로 들어가려는 요청을 하는 상황에서도 적용할 수 있는 다단계 경쟁 구조로 확장되어 있습니다.

2. N-Way 상호 배제를 보장하는 소프트웨어 동기화 방식

피터슨 알고리즘은 두 쓰레드 간에는 동기화되지만, 두 개 이상의 쓰레드가 경쟁하는 환경에서는 사용할 수 없습니다. 이에 비해 필터 락 알고리즘은 N개의 쓰레드가 순차적으로 상호 배제를 보장받으며 임계 영역에 진입할 수 있도록 설계되어 보다 일반적인 상황에서도 동작할 수 있는 구조라는 점에서 실용성과 확장성을 함께 지니고 있다고 판단하여 선택했습니다.

Step 3.

Step 2에서 구현한 알고리즘을 #1 문제에서 pthread_mutex 대신 활용해본다.

필터 락 알고리즘을 기반으로 세마포어나 뮤텍스를 사용하지 않고 오직 소프트웨어적인 필터 락 알고리즘만을 사용하여 1개의 생산자와 1개의 소비자가 공유 버퍼를 통해 데이터를 주고받는 프로그램인 procon2.c를 구현해 보았습니다.

3-1. 변수

```
volatile int level[N_THREADS];
volatile int victim[N_THREADS - 1];
```

그림 34. procon2.c의 전역 변수

- ✓ level[i] : i번 쓰레드가 진입을 시도 중인 경쟁 단계를 나타냅니다.
- ✓ victim[i-1] : 각 level에서 경쟁 중 누가 희생자(victim)로 지정되었는지 표시합니다.

3-2. 필터 락 함수 - lock

- 쓰레드가 level 1 → level 2 → .. 순으로 올라가며 임계 영역 진입을 시도합니다.
- 각 level에서는 한 명만 통과가 가능하며, victim이 지정되고 다른 쓰레드가 동시에 진입을 시도 중이면 희생자가 양보해야 합니다.

```
// filter lock function
void filter_lock(int thread_id) {
    for (int l = 1; l < N_THREADS; l++) {
        level[thread_id] = l;
        victim[l - 1] = thread_id;
        int other;
        do {
            int conflict = 0;
            for (other = 0; other < N_THREADS; other++) {
                if (other == thread_id) continue;
                if (level[other] >= l && victim[l - 1] == thread_id) {
                    conflict = 1;
                    break;
                }
            }
            if (!conflict) break;
        } while (1);
    }
}
```

그림 35. procon2.c의 필터 락 함수 - lock

3-3. 필터 락 함수 - unlock

- 필터 락 함수 내에 존재하며, 임계 영역을 빠져나갈 때 자신의 level을 초기화합니다.

```
void filter_unlock(int thread_id) {
    level[thread_id] = 0;
```

그림 36. procon2.c의 필터 락 함수 - unlock

3-4. 공유 버퍼

- 길이가 4의 배열형 원형 큐 형태입니다.

```
#define N_THREADS 2
#define N_COUNTER 4
#define MILLI 1000

// filter Lock variables
volatile int level[N_THREADS];
volatile int victim[N_THREADS - 1];

int queue[N_COUNTER];
int wptr = 0, rptr = 0;
```

그림 37. procon2.c의 공유 버퍼

- ✓ 이때, 유효 데이터 개수 관리는 포함되어 있지 않습니다. 이로 인해 아래와 같은 문제가 발생하게 됩니다.

```
os@test03:~/Desktop/os2$ ./procon2
producer: wrote 0
    consumer: read 0
    consumer: read 0
producer: wrote 1
producer: wrote 2
    consumer: read 2
    consumer: read 0
producer: wrote 3
producer: wrote 4
    consumer: read 4
    consumer: read 1
producer: wrote 5
producer: wrote 6
    consumer: read 6
producer: wrote 7
    consumer: read 7
producer: wrote 8
producer: wrote 9
    consumer: read 8
    consumer: read 9
```

그림 38. procon2.c 첫 번째 실행 결과

```
os@test03:~/Desktop/os2$ ./procon2
producer: wrote 0
    consumer: read 0
producer: wrote 1
    consumer: read 1
    consumer: read 0
    consumer: read 0
producer: wrote 2
producer: wrote 3
    consumer: read 0
    consumer: read 1
    consumer: read 2
producer: wrote 4
    consumer: read 3
producer: wrote 5
    consumer: read 4
producer: wrote 6
    consumer: read 5
producer: wrote 7
producer: wrote 8
producer: wrote 9
```

그림 39. procon2.c 두 번째 실행 결과

위와 같은 문제는 공유 버퍼(큐)에서 데이터를 정확히 동기화하지 못해 같은 데이터가 여러번 읽히거나 읽기/쓰기 순서가 뒤트리는 동기화 오류가 발생하고 있다는 것을 의미합니다.

필터 락 알고리즘 자체는 임계 영역 보호만을 제공하기 때문에 아래와 같은 문제가 있습니다.

1. 버퍼 공간 관리가 되지 않아 덮어쓰기 발생 가능성 존재

- 생산자가 데이터를 계속 queue[wptr]에 쓴 뒤 wptr++만 수행합니다.
- 소비자가 읽기 전에 producer가 wptr을 여러 번 증가시키면 소비자는 덮어쓴 값을 나중에 읽게 됩니다.

2. 유효한 데이터 개수에 대한 제어가 없음

- 현재 queue에 데이터가 몇 개 들어 있는지 소비자가 알 수 없기 때문에 소비자가 비어있는 큐를 읽거나, 이미 읽은 값을 다시 읽을 수 있게 됩니다.

이러한 문제를 해결하기 위해 semWrite 및 semRead와 같은 세마포어를 사용하는데, 현재 코드에서는 이를 생략한 상태이므로 정확한 생산자-소비자 순서를 보장하지 못하게 됩니다.

전체적인 코드 설명 이후 세마포어를 추가하여 수정한 procon2.c에 대해 다시 소개드리겠습니다.

3-5. 생산자 쓰레드

- 0부터 9까지의 값을 공유 버퍼에 하나씩 쓰고 출력합니다.
- 임계 영역(버퍼 접근)은 filter_lock과 filter_unlock으로 보호하고, 랜덤한 시간 간격으로 다음 데이터를 생산합니다.

```
// producer thread
void* producer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        filter_lock(id);

        // critical section: write
        queue[wptr] = i;
        wptr = (wptr + 1) % N_COUNTER;
        printf("producer: wrote %d\n", i);

        filter_unlock(id);

        usleep(MILLI * (rand() % 10) * 10);
    }
    return NULL;
}
```

그림 40. procon2.c의 생산자 쓰레드

3-6. 소비자 쓰레드

- 10번 데이터를 소비하는데, 이때 생산자가 쓴 데이터를 읽은 후 읽은 위치 포인터를 증가시킵니다.
- 생산자 쓰레드와 마찬가지로 임계 영역은 filter_lock 함수를 통해 보호됩니다.

```
// consumer thread
void* consumer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        filter_lock(id);

        // critical section: read
        int value = queue[rptr];
        rptr = (rptr + 1) % N_COUNTER;
        printf("\tconsumer: read %d\n", value);

        filter_unlock(id);

        usleep(MILLI * (rand() % 10) * 10);
    }
    return NULL;
}
```

그림 41. procon2.c의 소비자 쓰레드

3-7. 메인 함수

- 쓰레드 ID 0을 생산자, 1을 소비자로 지정하였습니다.
- 두 개의 쓰레드를 각각 생성하고 실행한 뒤 join()을 통해 기다리도록 구현하였습니다.

```
int main() {
    pthread_t threads[N_THREADS];
    int ids[N_THREADS] = {0, 1};
    srand(time(NULL));

    for (int i = 0; i < N_THREADS; i++)
        level[i] = 0;

    pthread_create(&threads[0], NULL, producer, &ids[0]);
    pthread_create(&threads[1], NULL, consumer, &ids[1]);

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

    return 0;
}
```

그림 42. procon2.c의 메인 함수

3-8. 세마포어를 추가하여 수정한 procon2.c

- 생산자 쓰레드에서 버퍼 공간이 남아있을 때만 쓰고(sem_wait), 쓴 후에는 소비자에게 읽을 수 있는 신호(sem_post)를 주도록 하였습니다.
- 소비자 쓰레드에서는 버퍼에 데이터가 있을 때만 읽고, 읽은 후에는 생산자에게 쓸 수 있는 공간을 되돌려 주도록 하였습니다.

```

void* producer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        sem_wait(&semWrite);
        filter_lock(id);
        queue[wptr] = i;
        wptr = (wptr + 1) % N_COUNTER;
        printf("producer: wrote %d\n", i);
        filter_unlock(id);
        sem_post(&semRead);
        usleep(MILLI * (rand() % 10) * 10);
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        sem_wait(&semRead);
        filter_lock(id);
        int value = queue[rptr];
        rptr = (rptr + 1) % N_COUNTER;
        printf("\tconsumer: read %d\n", value);
        filter_unlock(id);
        sem_post(&semWrite);
        usleep(MILLI * (rand() % 10) * 10);
    }
    return NULL;
}

```

그림 43. 세마포어를 추가한 생산자 및 소비자 쓰레드

```

int main() {
    pthread_t threads[N_THREADS];
    int ids[N_THREADS] = {0, 1};
    srand(time(NULL));

    for (int i = 0; i < N_THREADS; i++)
        level[i] = 0;

    sem_init(&semWrite, 0, N_COUNTER);
    sem_init(&semRead, 0, 0);

    pthread_create(&threads[0], NULL, producer, &ids[0]);
    pthread_create(&threads[1], NULL, consumer, &ids[1]);

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

    sem_destroy(&semWrite);
    sem_destroy(&semRead);

    return 0;
}

```

그림 44. 세마포어를 추가한 메인 함수

세마포어를 추가한 후 실행 결과는 다음과 같습니다.

```

os@test03:~/Desktop/os2$ ./procon2
producer: wrote 0
        consumer: read 0
producer: wrote 1
        consumer: read 1
producer: wrote 2
        consumer: read 2
producer: wrote 3
        consumer: read 3
producer: wrote 4
        consumer: read 4
producer: wrote 5
        consumer: read 5
producer: wrote 6
producer: wrote 7
        consumer: read 6
producer: wrote 8
        consumer: read 7
producer: wrote 9
        consumer: read 8
        consumer: read 9

```

그림 45. 수정된 procon2.c의 첫 번째 실행 결과

```

os@test03:~/Desktop/os2$ ./procon2
producer: wrote 0
        consumer: read 0
producer: wrote 1
        consumer: read 1
producer: wrote 2
        consumer: read 2
producer: wrote 3
        consumer: read 3
producer: wrote 4
producer: wrote 5
        consumer: read 4
        consumer: read 5
producer: wrote 6
producer: wrote 7
        consumer: read 6
        consumer: read 7
producer: wrote 8
        consumer: read 8
producer: wrote 9
        consumer: read 9

```

그림 46. 수정된 procon2.c의 두 번째 실행 결과

Step 4 - Bonus.

오리지널 pthread_mutex와 나의 software lock이랑 프로그램의 성능을 비교해본다.

기존 procon.c와 필터 락 알고리즘을 적용한 procon2.c의 성능 차이를 비교해 보았습니다.

- procon.c : pthread_mutex
- procon2.c : filter_lock

둘 중 어떤 방식이 더 빠르고 효율적인지 또는 실제 체감 성능 차이가 있는지를 비교하는 것이 목적입니다.

실행 시간을 측정하기 위해 time 명령어를 이용했지만, 한 번만 실행했을 때 둘의 성능 차이가 미미하게 나타났기 때문에 동일한 조건에서 10번 실행하여 평균값을 통해 비교해 보았습니다.

```
os@test03:~/Desktop/os2$ for i in {1..10}; do time ./procon; done
```

real	0m0.548s
user	0m0.000s
sys	0m0.004s

그림 47. pthread_mutex 기반의 procon.c를 10번 반복했을 때 결과

```
os@test03:~/Desktop/os2$ for i in {1..10}; do time ./procon2; done
```

real	0m0.539s
user	0m0.000s
sys	0m0.006s

그림 48. filter_lock 기반의 procon2.c를 10번 반복했을 때 결과

4-1. real time

- procon2가 0.009초 정도 더 빠르게 종료되었습니다. 이는 무작위 대기 시간(usleep)에 따른 자연스러운 편차 수준으로, 실제 동기화 방식에 따른 결정적 차이는 아니었습니다.

4-2. user time

- 사용자 공간에서 두 프로그램 모두 거의 CPU 사용(연산 소모)이 없는 모습을 보여주었습니다.

4-3. sys time

- procon이 0.004초, procon2가 0.006초로, procon2가 조금 더 많은 커널 리소스를 사용했음을 알 수 있습니다.
- 이는 procon2에서 세마포어 두 개(sem_wait, sem_post)가 추가로 호출되기 때문에, 시스템 호출(syscall)의 횟수가 많아졌기 때문입니다.

#03. 내 컴퓨터의 페이지 크기는 얼마일까?

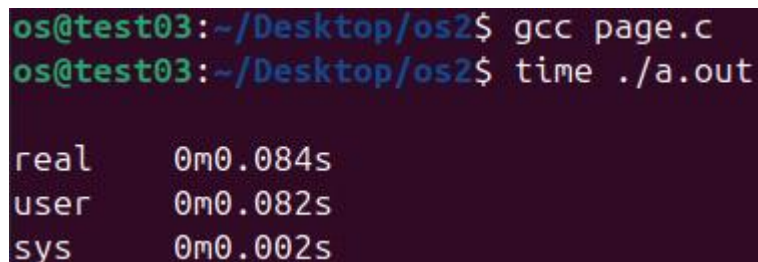
- ❖ 기대 결과물 : 찾아낸 pagesize에 대한 실행시간 캡처, 명령어를 통해 확인한 실제 page 크기 캡처, 관련 설명이 담긴 레포트
- ❖ 검색 키워드 : check linux page size

Step 1.

page.c 코드를 컴파일하고 실행시켜본다.

실행시킬 때 time 명령어와 함께 실행하여 실행 시간을 측정해본다.

ex) time ./a.out



```
os@test03:~/Desktop/os2$ gcc page.c
os@test03:~/Desktop/os2$ time ./a.out

real    0m0.084s
user    0m0.082s
sys     0m0.002s
```

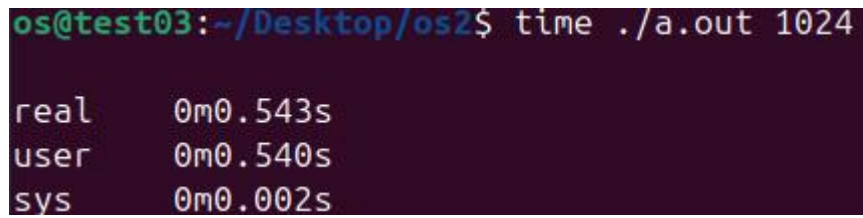
그림 49. page.c 코드 컴파일 및 실행 후 결과

Step 2.

컴파일된 파일은 pagesize 변수를 인자값으로 받는다.

값을 변경해가며 실행시간의 변화를 확인해본다.

ex) time ./a.out 1024



```
os@test03:~/Desktop/os2$ time ./a.out 1024  
  
real    0m0.543s  
user    0m0.540s  
sys     0m0.002s
```

그림 50. pagesize를 1024 바이트로 설정한 후 실행 결과

Step 3.

입력 값이 특정 값에 이르면(i.e., 실제 page size) 갑자기 실행시간이 확 증가한다.

이를 통해 컴퓨터의 페이지 크기를 확인해보고,
시간 값의 차이가 잘 보이지 않거나 실행에 이상이 있다면 #define된 각종 값을 변경해본다.

3-1. 점진적으로 pagesize 증가시키기

- page 프로그램의 입력 값으로 pagesize를 점진적으로 변경하면서 실행시간을 관측하였습니다.
- 실제 시스템의 페이지 크기와 일치하는 값을 입력할 경우, 매 접근마다 page fault가 발생하게 되어 실행 시간이 급격히 증가하게 됩니다.

```

os@test03:~/Desktop/os2$ time ./a.out 200
real    0m0.086s
user    0m0.085s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 500
real    0m0.090s
user    0m0.089s
sys     0m0.000s
os@test03:~/Desktop/os2$ time ./a.out 512
real    0m0.495s
user    0m0.493s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 800
real    0m0.096s
user    0m0.092s
sys     0m0.004s
os@test03:~/Desktop/os2$ time ./a.out 1000
real    0m0.097s
user    0m0.093s
sys     0m0.004s
os@test03:~/Desktop/os2$ time ./a.out 1024
real    0m0.547s
user    0m0.546s
sys     0m0.000s
os@test03:~/Desktop/os2$ time ./a.out 2000
real    0m0.099s
user    0m0.097s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 2048
real    0m0.541s
user    0m0.539s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 4000
real    0m0.106s
user    0m0.105s
sys     0m0.000s
os@test03:~/Desktop/os2$ time ./a.out 4096
real    0m0.550s
user    0m0.548s
sys     0m0.002s

```

그림 51. 점진적으로 pagesize를 키웠을 때 실행 결과

3-2. 실행 시간 측정 결과

Pagesize [Bytes]	Real-time [s]	User-time [s]	Sys-time [s]
200	0.086	0.085	0.002
500	0.090	0.089	0.000
512	0.495	0.493	0.002
800	0.096	0.092	0.004
100	0.097	0.093	0.004
1024	0.547	0.546	0.000
2000	0.099	0.097	0.000
2048	0.541	0.539	0.002
4000	0.106	0.105	0.000
4096	0.550	0.548	0.002
8000	0.093	0.091	0.001
8192	0.542	0.540	0.002
16000	0.161	0.159	0.002
16384	0.541	0.540	0.000

- ✓ pagesize가 256 바이트의 배수일 때 실행 시간이 급격하게 증가함을 확인하였습니다.
- ✓ 해당 지점에서 page fault가 발생하면서 실행 시간이 증가했음을 알 수 있고, 해당 값이 현재 시스템의 실제 페이지 크기일 가능성이 매우 높을 것이라고 생각했습니다.

실제 페이지 크기	0	1	2	3	4	5	6
size A	0	1	2	3	4	5	6
size B	0	1	2	3	4	5	6
size C	0	1	2	3	4	5	6
size D	0	1	2	3	4	5	6
size E	0	1	2	3	4	5	6
size F	0	1	2	3	4	5	6

그림 52. Assignment02 #3 - Pagesize

그림 53의 내용을 기반으로 위 그림 54의 내용을 확인해 보았습니다.

실제 페이지 크기를 기반으로 pagesize를 A부터 F까지 바꿔가며 테스트 하였을 때, size A-D는 실제 pagesize보다 작기 때문에 한 페이지 안에 여러 값들이 있으므로 page fault의 횟수가 더 작게 나타납니다.

하지만 예를 들어 1024, 4096, .. 와 같이 실제 페이지 크기와 page size의 크기가 똑같아지는 경우 매 접근마다 page fault가 발생하기 때문에 프로그램 실행 시간이 더 길어지게 됩니다.

즉, size E에서 실행 시간이 가장 크게 나타나고 '3-2. 실행 시간 측정 결과'의 빨간색 하이라이트가 되어있는 경우와 같이 나타나게 됩니다.

size F의 경우 실제 페이지 크기보다 더 커진 경우는 Step 4에서 더 알아볼 예정이지만, 서로 겹쳐 있는 부분이 있다보니 size E의 경우보다는 page fault가 덜 발생하게 됩니다.

Step 4.

리눅스에서 컴퓨터에 설정된 페이지 크기를 확인하는 명령어가 무엇인지 찾아보고,
찾아낸 값과 일치하는지 비교해본다.

거꾸로 실제 값을 확인해놓고 값을 찾아본다.

ex) 실제 값이 1000이라면 999, 1000, 1001과 같이 실행

4-1. 컴퓨터에 설정된 페이지 크기를 확인하는 명령어

```
os@test03:~/Desktop/os2$ getconf PAGE_SIZE
4096
```

그림 53. 시스템에서 사용하는 기본 페이지 크기 확인

- ✓ 실제 pagesize 값이 4096임을 확인하였습니다.

4-2. 실제 값과 가까운 값들 실행

```
os@test03:~/Desktop/os2$ time ./a.out 4094
real    0m0.098s
user    0m0.096s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 4095
real    0m0.461s
user    0m0.459s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 4096
real    0m0.561s
user    0m0.559s
sys     0m0.002s
os@test03:~/Desktop/os2$ time ./a.out 4097
real    0m0.381s
user    0m0.377s
sys     0m0.004s
os@test03:~/Desktop/os2$ time ./a.out 4098
real    0m0.098s
user    0m0.096s
sys     0m0.002s
```

그림 54. 시스템에서 사용하는 기본 페이지 크기와 가까운 값들에 대한 실행 결과

- ✓ 4095일 때 실행 시간이 눈에 띄게 증가했지만 4096에서 가장 높은 실행 시간을 보이고 4097에서 다시 감소하는 경향을 확인하였습니다.
- ✓ 이를 통해 실제 시스템의 페이지 단위가 성능에 영향을 미친다는 점과 페이지 크기에 따라 메모리 접근 효율성이 달라질 수 있음을 직접 확인할 수 있었습니다.