운영체제 과제 (Assignment #2. Synchronization and memory)

박지환(184128) | 전남대학교 지역바이오시스템공학과 생물산업기계공학전공

#1. 생산자-소비자로 구성된 응용프로그램 만들기

Step 1)

- 생산자-소비자 문제는 공유 버퍼를 사이에 두고 공유 버퍼에 데이터를 공급하는 생산자들과 데이터를 읽고 소비하는 소비자들이 공유 버퍼를 문제 없이 사용하도록 생산자와 소비자를 동기화시키는 문제입니다.
- 생산자-소비자에서 고려해야할 3가지 문제점
  - 상호 배제 해결: 생산자들과 소비자들의 공유 버퍼에 대한 상호 배제
  - 비어 있는 공유 버퍼 문제: 비어 있는 공유 버프를 소비자가 읽으면 안됨
  - 꽉 찬 공유 버퍼 문제: 꽉 찬 공유 버퍼에 생산자가 더 이상 데이터를 입력해선 안됨
- 문제 해결 방법: 세마포어 2개 이용하기
  - 읽기용 세마포어로 읽기 가능한 버퍼 갯수를 확인해 비어있는 버퍼 문제 해결
  - 쓰기용 세마포어로 쓰기 가능한 버퍼 갯수를 확인해 비어있는 버퍼 문제 해결

Step 2 & Step 3)

- mywrite 함수
  - 세마포어 semWrite를 기다리면서 빈 슬롯이 있는지 확인 후 뮤텍스를 잠그고 공유 버퍼에 데이터를 씁니다.
  - 데이터를 쓰고 semRead 세마포어를 증가시켜 소비자가 데이터를 읽을 수 있다는 것을 알려줍니다.
- myread 함수

- 세마포어 semRead를 기다리면서 빈 슬롯이 있는지 확인 후 뮤텍스를 잠그고 공유 버퍼에 데이터를 읽습니다.
- 데이터를 읽고 semWrite 세마포어를 증가시켜 생산자가 데이터를 쓸 수 있다는 것을 알려줍니다.

```
45  // write n into the shared memory
46  void mywrite(int n) {
47    /* [Write here] */
48      sem_wait(&semWrite);
49      pthread_mutex_lock(&critical_section);
50
51      queue[wptr] = n;
52      wptr = (wptr + 1) % N_COUNTER;
53
54      pthread_mutex_unlock(&critical_section);
55      sem_post(&semRead);
56  }
57
58  // write a value from the shared memory
59  int myread() {
60    /* [Write here] */
61      sem_wait(&semRead); // wait for a filled slot
62      pthread_mutex_lock(&critical_section);
63
64      int n = queue[rptr];
65      rptr = (rptr + 1) % N_COUNTER;
66
67      pthread_mutex_unlock(&critical_section);
68      sem_post(&semWrite);
69
70    return n;
71  }
```

- main 함수
  - 뮤텍스와 세마포어를 초기화하고, 생산자와 소비자 쓰레드를 생성해 실행합니다.
  - 모든 쓰레드가 종료될 때까지 기다리고 세마포어와 뮤텍스를 소멸시킵니다.

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

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

  // init semaphore
  /* [Write here] */
    sem_init(&semWrite, 0, N_COUNTER);
    sem_init(&semRead, 0, 0);

  // 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] */
    sem_destroy(&semWrite);
    sem_destroy(&semRead);

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

```

#2. 소프트웨어로 문을 만드는 방법

Step 1)

- Dekker 알고리즘
  - 두 개의 프로세스가 임계 구역에 진입하지 않도록 보장하는 최초의 알고리즘 중 하나입니다.
  - 두 개의 플래그와 하나의 턴 변수를 사용해 상호 배제를 구현하는데, 각 프로세스는 임계 구역에 진입하기 전에 자신의 플래그를 세우고 다른 프로세스의 플래그와 턴 변수를 확인해 진입 여부를 결정하게 됩니다.
- Peterson 알고리즘
  - 두 개의 프로세스가 임계 구역에 동시에 진입하지 않도록 하기 위해 고안된 알고리즘입니다.

    - 두 개의 플래그와 하나의 턴 변수를 사용해 상호 배제를 보장하는데, 각 프로세스는 임계 구역에 진입하기 전 자신의 플래그를 세우고 다른 프로세스의 플래그와 턴 변수를 확인해 진입 여부를 결정하게 됩니다.
- Lamport의 Bakery 알고리즘
    - 여러 프로세스가 임계 구역에 동시에 진입하기 않도록 하기 위해 고안된 알고리즘입니다.
    - 각 프로세스에 번호를 할당하고 번호가 작은 프로세스가 먼저 임계 구역에 진입할 수 있도록 하며, 프로세스는 임계 구역에 진입하기 전에 번호를 할당받고 임계 구역을 나올 때 번호를 초기화하게 됩니다.

Step 2 & Step 3)

Peterson 알고리즘

- Peterson 알고리즘을 사용해 생산자-소비자 문제를 해결하였습니다.
- pthread_mutex 대신 peterson_lock과 peterson_unlock을 사용하여 상호 배제를 구현하였습니다.

```c
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

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

void mfence() {
    asm volatile("mfence" ::: "memory");
}

void peterson_lock(int self) {
    int other = 1 - self;
    flag[self] = 1;
    turn = other;
    mfence();
    while (flag[other] && turn == other) {
        // Busy-wait
    }
}

void peterson_unlock(int self) {
    flag[self] = 0;
}

#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int count = 0;
```

```c
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int count = 0;

void *producer(void *param) {
    int self = *(int *)param;
    for (int i = 0; i < 10; i++) {
        peterson_lock(self);
        if (count < BUFFER_SIZE) {
            buffer[count++] = i;
            printf("Produced: %d\n", i);
        }
        peterson_unlock(self);
    }
    return NULL;
}

void *consumer(void *param) {
    int self = *(int *)param;
    for (int i = 0; i < 10; i++) {
        peterson_lock(self);
        if (count > 0) {
            int item = buffer[--count];
            printf("Consumed: %d\n", item);
        }
        peterson_unlock(self);
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    int id1 = 0, id2 = 1;

    pthread_create(&tid1, NULL, producer, &id1);
    pthread_create(&tid2, NULL, consumer, &id2);

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

    return 0;
}
```

Step 4)

오류

```
1 error generated.
 ✘  Latency  ~/desktop/assignment02  main  gcc -o mutex_program procon2ex.c -pthread
 Latency  ~/desktop/assignment02  main  gcc -o peterson_program procon2.c -pthread
procon2.c:9:18: error: unrecognized instruction mnemonic
    asm volatile("mfence" ::: "memory");
                 ^
<inline asm>:1:2: note: instantiated into assembly here
        mfence
        ^
1 error generated.
```

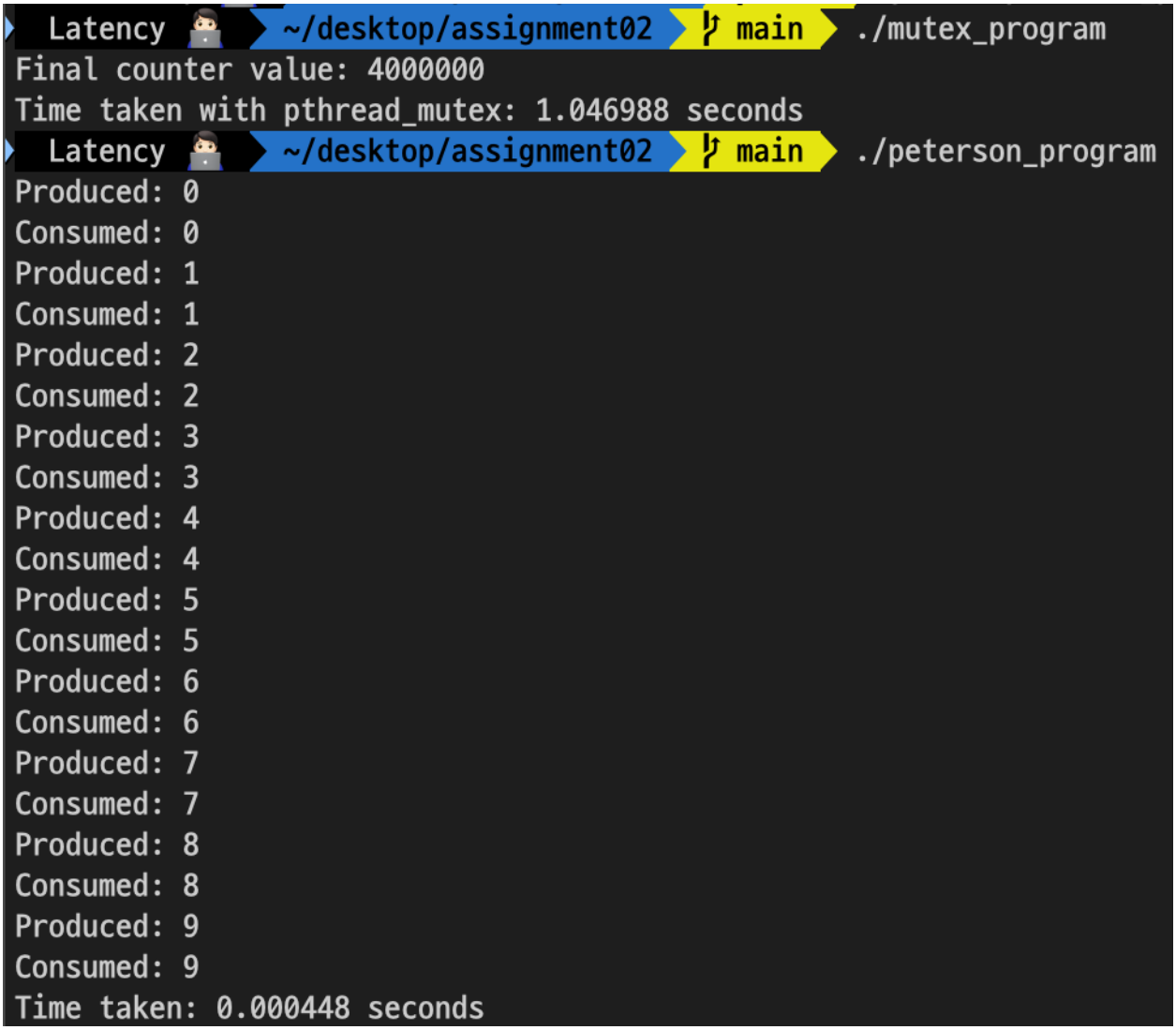
- mfence 명령어가 다음과 같이 인식되지 않는 문제를 해결하기 위해 컴파일러가 x86 아키텍처용으로 컴파일 되도록 gcc를 사용할 때 -march 옵션을 추가해 x86-64 아키텍처용으로 컴파일하도록 설정하였습니다.

```
 ✘  Latency  ~/desktop/assignment02  main  gcc -o peterson_program procon2.c -pthread -march=x86-64
clang: error: unsupported argument 'x86-64' to option '-march='
 ✘  Latency  ~/desktop/assignment02  main  gcc -o peterson_program procon2.c -pthread
 Latency  ~/desktop/assignment02  main
```

- 하지만 에러가 여전히 발생하게 되어 다음과 같이 gcc의 __sync_synchronize를 사용해 보았습니다.

```
 8  void mfence() {
 9      // asm volatile("mfence" ::: "memory");
10      __sync_synchronize();
11  }
```

결과



- 다음과 같은 코드를 컴파일하여 비교해본 결과, pthread_mutex의 성능보다 peterson의 성능이 좋았음을 알 수 있었습니다.
  - `gcc -o mutex_program procon2ex.c -pthread`
  - `gcc -o peterson_program procon2.c -pthread`

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

Step 1)

- page.c 파일을 gcc를 사용해 컴파일하고 time 명령어와 함께 실행하였습니다.

```
Latency ~/desktop/assignment02 main gcc -o page page.c
Latency ~/desktop/assignment02 main time ./page out
./page out  0.13s user 0.01s system 22% cpu 0.596 total
```

Step 2 & Step 3)

- 512, 1024, 2048 처럼 값을 변경해보며 실행시간의 변화를 알아보았습니다.

```
Latency ~/desktop/assignment02 main time ./page 1024
./page 1024  0.36s user 0.01s system 98% cpu 0.375 total
Latency ~/desktop/assignment02 main time ./page 512
./page 512  0.15s user 0.00s system 98% cpu 0.155 total
Latency ~/desktop/assignment02 main time ./page 2048
./page 2048  0.50s user 0.01s system 99% cpu 0.507 total
```

- pagesize의 값이 커질수록 실행시간이 증가함을 확인할 수 있었습니다.

Step 4)

- getconf PAGESIZE 및 sysctl hw.pagesize를 통해 컴퓨터에 설정된 페이지 크기가 16384 임을 확인할 수 있었습니다.

```
Latency ~/desktop/assignment02 main getconf PAGESIZE
16384
Latency ~/desktop/assignment02 main sysctl hw.pagesize
hw.pagesize: 16384
```