assignment 02

인공지능학부
211852 조나현

#1. 생산자-소비자로 구성된 응용프로그램 만들기

- 코드 분석(procon.c)

1. 전역변수

N_COUNTER는 공유 버퍼의 크기를 정의하는 매크로입니다.

MILLI는 밀리초 단위의 시간 단위를 정의하는 매크로입니다.

crtical_section은 뮤텍스를 나타내는 POSIX mutex입니다.

semWrite와 semRead는 세마포어를 나타내는 POSIX semaphore입니다.

queue는 공유 버퍼를 나타내는 정수 배열입니다.

wptr는 공유 버퍼에 쓰기 작업을 위한 쓰기 포인터입니다.

rptr는 공유 버퍼에서 읽기 작업을 위한 읽기 포인터입니다.

#1. 생산자-소비자로 구성된 응용프로그램 만들기

2. mywrite() 함수

mywrite() 함수는 n을 공유 메모리에 쓰는 함수입니다.

sem_wait(&semWrite)는 쓰기 가능한 슬롯이 있는지 기다립니다.

pthread_mutex_lock(&critical_section)은 임계 영역을 보호하기 위해 뮤텍스를 잠급니다.

queue[wptr] = n은 공유 버퍼에 값을 씁니다.

wptr = (wptr + 1) % N_COUNTER는 쓰기 포인터를 업데이트합니다.

pthread_mutex_unlock(&critical_section)은 뮤텍스를 해제합니다.

sem_post(&semRead)는 읽을 수 있는 데이터의 사용 가능성을 알립니다.

#1. 생산자-소비자로 구성된 응용프로그램 만들기

3. myread() 함수

myread() 함수는 공유 메모리에서 값을 읽어오는 함수입니다.

sem_wait(&semRead)는 읽을 수 있는 데이터가 있는지 기다립니다.

pthread_mutex_lock(&critical_section)은 임계 영역을 보호하기 위해 뮤텍스를 잠급니다.

int n = queue[rptr]는 공유 버퍼에서 값을 읽어옵니다.

rptr = (rptr + 1) % N_COUNTER는 읽기 포인터를 업데이트합니다.

pthread_mutex_unlock(&critical_section)은 뮤텍스를 해제합니다.

sem_post(&semWrite)는 쓰기 가능한 슬롯의 사용 가능성을 알립니다.

return n은 읽어온 값을 반환합니다.

#1. 생산자-소비자로 구성된 응용프로그램 만들기

4. producer() 함수
producer() 함수는 생산자 스레드의 역할을 수행합니다.
0부터 9까지 반복하면서 mywrite(i)를 호출하여 값을 공유 메모리에 씁니다.
쓴 값 i를 출력합니다.
usleep(MILLI*m*10)은 임의의 시간 동안 스레드를 일시 정지합니다.

5.consumer() 함수
consumer() 함수는 소비자 스레드의 역할을 수행합니다.
0부터 9까지 반복하면서 myread()를 호출하여 공유 메모리에서 값을 읽어옵니다.
읽어온 값 n을 출력합니다.
usleep(MILLI*m*10)은 임의의 시간 동안 스레드를 일시 정지합니다.

#1. 생산자-소비자로 구성된 응용프로그램 만들기

6. main() 함수

t[2]는 스레드 구조체 배열입니다.

srand(time(NULL))는 난수 발생을 위해 시간을 기반으로 시드 값을 초기화합니다.

pthread_mutex_init(&critical_section, NULL)는 뮤텍스를 초기화합니다.

sem_init(&semWrite, 0, N_COUNTER)는 semWrite를 쓰기 가능한 슬롯의 최대 개수인 N_COUNTER로 초기화합니다.

sem_init(&semRead, 0, 0)는 semRead를 0으로 초기화합니다.

pthread_create(&t[0], NULL, producer, NULL)는 생산자 스레드를 생성합니다.

pthread_create(&t[1], NULL, consumer, NULL)는 소비자 스레드를 생성합니다.

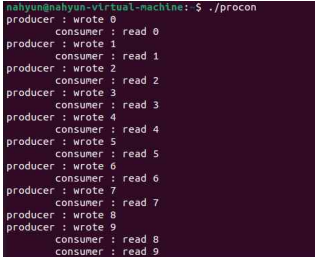
pthread_join(t[i],NULL)은 스레드의 종료를 기다립니다.

sem_destroy(&semWrite)와 sem_destroy(&semRead)는 세마포어를 파괴합니다.

pthread_mutex_destroy(&critical_section)은 뮤텍스를 파괴합니다.

#1. 생산자-소비자로 구성된 응용프로그램 만들기

- 결과물



#2.소프트웨어로 문을 만드는 방법

Dekker 알고리즘:
Edsger Dijkstra에 의해 개발된 알고리즘으로, 상호배제를 위한 먼저들어온 스레드가 임계 영역에 진입하는 방식으로 동작합니다.
두 개의 스레드가 번갈아가며 임계 영역에 진입하도록 하는데, 상호배제와 교착상태 없이 동작합니다.
하지만 하드웨어의 최신 메모리 모델에서는 동작하지 않을 수 있기 때문에, 일부 제약이 있습니다.
Peterson 알고리즘:
Peterson에 의해 제안된 알고리즘으로, 두 개의 스레드 간의 상호배제를 달성하기 위해 사용됩니다.
상호배제를 위한 플래그와 교착상태 방지를 위한 턴 변수를 사용하여 동작합니다.
각 스레드는 자신의 플래그를 설정하고 턴 변수를 사용하여 상대 스레드에게 차례를 넘겨줍니다.
Lamport 알고리즘:
Leslie Lamport에 의해 개발된 알고리즘으로, 분산 시스템에서의 상호배제를 위해 사용됩니다.
논리적인 시계 값을 사용하여 각 프로세스 간의 상호작용을 조정합니다.
이 알고리즘은 메시지 전달 프로토콜을 기반으로 하여, 분산 시스템에서의 동기화를 달성합니다.

#2.소프트웨어로 문을 만드는 방법

- 동기화 방식(알고리즘) 선택

=> Peterson 알고리즘

<선택 이유>

=> 상호배제, 교착 상태 없음, 두개의 프로세스에 적합, Busy-Wait가 없다는 특징이 선택한 이유입니다.

#2.소프트웨어로 문을 만드는 방법

- Peterson 알고리즘 구현

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

#define NUM_THREADS 2

int flag[2];
int turn = 0;

pthread_mutex_t mutex;
```

#2.소프트웨어로 문을 만드는 방법

- Peterson 알고리즘 구현

```
void* thread_func(void* arg) {
    int thread_id = *((int*)arg);

    for (int i = 0; i < 10; i++) {
        flag[thread_id] = 1;
        turn = 1 - thread_id;

        while (flag[1 - thread_id] && turn == 1 - thread_id) {
            // Busy-waiting
        }

        // Critical Section
        pthread_mutex_lock(&mutex);
        printf("Thread %d is in the critical section.\n", thread_id);
        pthread_mutex_unlock(&mutex);

        flag[thread_id] = 0;

        // Remainder Section
        printf("Thread %d is in the remainder section.\n", thread_id);
    }

    return NULL;
}
```

#2.소프트웨어로 문을 만드는 방법

- Peterson 알고리즘 구현

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

    pthread_mutex_init(&mutex, NULL);

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

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

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

#2.소프트웨어로 문을 만드는 방법

- 결과물

```
nahyun@nahyun-virtual-machine:~$ ./procon2
producer : wrote 0
        consumer : read 0
producer : wrote 1
producer : wrote 2
        consumer : read 1
producer : wrote 3
producer : wrote 4
        consumer : read 2
producer : wrote 5
        consumer : read 3
        consumer : read 4
        consumer : read 5
producer : wrote 6
producer : wrote 7
producer : wrote 8
        consumer : read 6
producer : wrote 9
        consumer : read 7
        consumer : read 8
        consumer : read 9
```

#2.소프트웨어로 문을 만드는 방법

- 코드 분석(procon2.c)
전역 변수 및 상수:
N_COUNTER: 공유 버퍼의 크기를 나타내는 상수입니다.
MILLI: 시간 단위를 나타내는 상수입니다.
함수 선언:
void lock(int threadId): Peterson 알고리즘의 락 함수입니다. threadId를 기준으로 상호 배제를 수행합니다.
void unlock(int threadId): Peterson 알고리즘의 언락 함수입니다. threadId에 해당하는 스레드의 상호 배제 상태를 해제합니다.
void* producer(void* arg): 생산자 스레드의 동작을 정의한 함수입니다. 0부터 9까지의 값을 공유 메모리에 쓰고, 일정 시간 동안 대기합니다.
void* consumer(void* arg): 소비자 스레드의 동작을 정의한 함수입니다. 공유 메모리에서 값을 읽고, 일정 시간 동안 대기합니다.
void mywrite(int n): 값을 공유 메모리에 쓰는 함수입니다.
int myread(): 공유 메모리에서 값을 읽는 함수입니다.
전역 변수 초기화:
semWrite: 쓰기 가능한 슬롯을 나타내는 POSIX 세마포어입니다.
semRead: 읽을 수 있는 데이터를 나타내는 POSIX 세마포어입니다.
queue: 공유 버퍼로 사용되는 배열입니다.
wptr: 쓰기 포인터로, 공유 버퍼의 다음 쓰기 위치를 가리킵니다.
rptr: 읽기 포인터로, 공유 버퍼의 다음 읽기 위치를 가리킵니다.
turn: Peterson 알고리즘의 순서를 결정하는 변수입니다.
interested: Peterson 알고리즘의 상호 배제 상태를 저장하는 배열입니다.

#2.소프트웨어로 문을 만드는 방법

- 코드 분석(procon2.c)
메인 함수:
sem_init: 세마포어를 초기화합니다.
pthread_create: 생산자와 소비자 스레드를 생성합니다.
pthread_join: 스레드의 종료를 기다립니다.
sem_destroy: 세마포어를 해제합니다.
producer 함수:
0부터 9까지의 값을 순차적으로 생성합니다.
lock(0) 함수를 호출하여 생산자 스레드의 상호 배제를 시작합니다.
mywrite(i) 함수를 호출하여 i 값을 공유 메모리에 씁니다.
"producer : wrote %d" 형식의 메시지를 출력합니다.
unlock(0) 함수를 호출하여 생산자 스레드의 상호 배제를 해제합니다.
usleep 함수를 사용하여 일정 시간 동안 스레드를 대기시킵니다.
consumer 함수:
0부터 9까지의 값을 순차적으로 생성합니다.
lock(1) 함수를 호출하여 소비자 스레드의 상호 배제를 시작합니다.
myread() 함수를 호출하여 공유 메모리에서 값을 읽습니다.
"\tconsumer : read %d" 형식의 메시지를 출력합니다.
unlock(1) 함수를 호출하여 소비자 스레드의 상호 배제를 해제합니다.
usleep 함수를 사용하여 일정 시간 동안 스레드를 대기시킵니다.

#2.소프트웨어로 문을 만드는 방법

- 코드 분석(procon2.c)

myread 함수:

sem_wait(&semRead) 함수를 호출하여 읽을 수 있는 데이터를 기다립니다.

공유 버퍼의 rptr 위치에서 값을 읽습니다.

rptr 값을 다음 읽기 위치로 업데이트합니다.

sem_post(&semWrite) 함수를 호출하여 쓰기 가능한 슬롯의 가용성을 신호화합니다.

lock(int threadId) 함수:

threadId를 기준으로 현재 스레드의 상호 배제 상태를 설정합니다. interested[threadId]를 1로 설정하여 현재 스레드가 상호 배제를 원한다는 것을 나타냅니다.

turn 변수를 1 - threadId로 설정하여 번갈아가며 실행할 수 있도록 합니다.

나머지 스레드가 상호 배제를 원하고, 현재 차례가 아닌 경우에는 busy waiting 상태로 대기합니다.

unlock(int threadId) 함수:

threadId를 기준으로 현재 스레드의 상호 배제 상태를 해제합니다. interested[threadId]를 0으로 설정하여 현재 스레드가 상호 배제를 원하지 않는다는 것을 나타냅니다.

#2.소프트웨어로 문을 만드는 방법

- 코드 분석(procon2.c)

생산자 스레드(producer)는 0부터 9까지의 값을 순차적으로 생성하고, mywrite(i) 함수를 호출하여 값을 queue에 씁니다. 그 후, 해당 값을 출력하고 unlock(0) 함수를 호출하여 상호 배제를 해제합니다.

소비자 스레드(consumer)는 0부터 9까지의 값을 순차적으로 생성하고, myread() 함수를 호출하여 queue에서 값을 읽습니다. 그 후, 해당 값을 출력하고 unlock(1) 함수를 호출하여 상호 배제를 해제합니다.

메인 함수에서는 세마포어를 초기화하고, 생산자와 소비자 스레드를 생성합니다. 생성된 스레드가 종료될 때까지 pthread_join 함수를 사용하여 기다립니다. 스레드의 실행이 완료되면 세마포어를 해제하여 메모리를 정리합니다.

이렇게 Peterson 알고리즘을 사용하여 생산자와 소비자 스레드가 상호 배제를 구현하고, queue라는 공유 자원을 안전하게 접근하도록 보장합니다.

#2.소프트웨어로 문을 만드는 방법

- 성능 비교

Peterson 알고리즘은 pthread_mutex와 비교했을 때 성능적으로 비효율적일 수 있습니다. 이는 Peterson 알고리즘이 busy-waiting을 사용하기 때문입니다. Busy-waiting은 자원을 얻기 위해 반복적으로 확인하는 방식으로, CPU 시간을 낭비할 수 있습니다.

반면에 pthread_mutex는 대부분의 운영 체제에서 지원하는 동기화 메커니즘으로, 효율적인 방식으로 자원의 잠금과 해제를 처리합니다. pthread_mutex는 운영 체제 수준에서의 지원을 받으며, 스레드 스케줄링 및 자원 할당과 같은 작업을 운영 체제에게 맡깁니다. 따라서 pthread_mutex는 대부분의 상황에서 더 효율적이고 안정적인 선택입니다.

실제로 Peterson 알고리즘 pthread_mutex에 비해 결과값이 도출되는 시간이 훨씬 오래걸렸습니가.

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

- 결과물(step1)

```
nahyun@nahyun-virtual-machine:~$ time gcc -o page page.c -pthread

real    0m0.065s
user    0m0.056s
sys     0m0.005s
nahyun@nahyun-virtual-machine:~$ time ./page

real    0m0.287s
user    0m0.278s
sys     0m0.004s
```

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

- 결과물(step2)

```
nahyun@nahyun-virtual-machine:~$ time ./page 1024

real    0m1.034s
user    0m1.021s
sys     0m0.004s
```

```
nahyun@nahyun-virtual-machine:~$ time ./page 2048

real    0m0.930s
user    0m0.926s
sys     0m0.000s
```

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

- 결과물(step3)

```
nahyun@nahyun-virtual-machine:~$ time ./page 4095

real    0m0.652s
user    0m0.647s
sys     0m0.000s
nahyun@nahyun-virtual-machine:~$ time ./page 4096

real    0m0.932s
user    0m0.925s
sys     0m0.000s
nahyun@nahyun-virtual-machine:~$ time ./page 4097

real    0m0.638s
user    0m0.634s
sys     0m0.000s
```

=>4096값에서 실행시간이 증가하는 것을 볼 수 있다.

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

- 페이지 크기 확인 명령어

=> getconf PAGE_SIZE(step4)