운영체제 과제(6/21)

211239 신혜서

1) 07강 - 생성자-소비자 문제 정리

1-1) 생산자-소비자 문제 요약
멀티스레드 환경에서 공유 자원인 대기 큐를 사용할 때 발생하는 문제이다. 생산자 스레드는 대기 큐에 요청을 추가하고, 소비자 스레드는 대기 큐에서 요청을 꺼내 처리한다. 이 과정에서 race condition이 발생할 수 있어 상호배제와 조건 동기화 전략이 필요하다.
또한 대기 큐의 크기가 제한되어 있어 발생하는 문제를 해결하기 위해서는 조건 동기화 기법을 활용해야 한다.

1-2) 생산자-소비자 문제의 개요
문제 상황 : 생산자가 데이터를 생산하고 소비자가 이를 소비하는 과정에서 발생하는 문제
해결책 : 버퍼를 사용하지만, 버퍼의 크기가 유한하므로 생산자와 소비자 간의 동기화가 필요
동기화 기법 : 세마포어를 사용하여 임계 구역에 대한 상호 배타적 접근을 보장함으로써 데이터 변화 오류를 해결함

1-3) 생산자-소비자 문제의 해결 방법
1. 세마포어를 이용한 해결
   - 기본사항 : 세마포어 e를 사용하여 버퍼가 차있는 것을 검사한다.
   - 생산자 스레드 : 버퍼에 데이터 추가 시 세마포어 e를 감소시켜 버퍼가 차있음을 표시한다.
   - 소비자 스레드 : 버퍼에서 데이터를 꺼낼 때 세마포어 e를 증가시켜 버퍼가 비어있음을 표시한다.
2. 모니터를 이용한 해결 :
   - 모니터는 상호배제와 조건 동기화를 제공하는 동기화 기법이다.
   - 생산자와 소비자 스레드가 모니터의 메서드를 호출하여 상호작용한다.
   - 모니터 내부에서 상호배제와 조건 동기화를 보장하여 race condition을 해결한다.

1-4) 운영체제의 프로세스 동기화
1. 운영체제에서 프로세스 간 동기화 문제는 매우 중요한 주제이다.
2. 프로세스 간 경쟁 상황에서 발생할 수 있는 race condition을 해결하기 위해 세마포어, 모니터 등의 동기화 기법을 사용한다.
3. 프로세스 간 상호배제와 동기화를 보장하여 안정한 프로그램을 실행할 수 있도록 만든다.
4. 데이터 손실, 데드락 등의 문제를 방지하고 프로세스 간 협력을 보장할 수 있다.

2) 생산자-소비자 어플리케이션 만들기
우분투 실행이 불가능하여 윈도우 운영체제에서 C파일을 구동시켰다.

문제에 제시된대로 단일 생산자(single producer)와 단일 소비자(single consumer)가 공유 버퍼를 사용하여 데이터를 주고받는 간단한 생산자-소비자 문제를 해결하는 Windows 기반 C프로그램이다.
프로그램은 Windows API를 사용하여 동기화 메커니즘(세마포어와 뮤텍스)을 구현했다.

[코드의 주요 구성 요소와 동작 방식]

1. 주요 변수 및 객체

- HANDLE semWrite, semRead;
  생산자가 데이터를 쓸 수 있는 공간의 수를 나타내는 세마포어(semWrite)와 소비자가 데이터를 읽을 수 있는 데이터의 수를 나타내는 세마포어(semRead)를 정의했다.
- int queue[N_COUNTER];
  공유 버퍼로 사용되는 정수 배열이다. 생산자와 소비자가 공유하여 데이터를 주고받는다.
- int wptr, rptr;
  공유 버퍼 내에서 쓰기와 읽기 위치를 가리키는 포인터이다.
- HANDLE mutex;
  공유 자원(여기서는 queue, wptr, rptr)에 대한 접근을 동기화하기 위한 뮤텍스이다.

2. 함수 설명

- DWORD WINAPI producer(LPVOID arg);
  생산자 스레드의 시작 지점을 나타낸다. mywrite함수를 통해 공유 버퍼에 데이터를 쓰고, 쓰기 작업이 끝날 때마다 콘솔에 메시지를 출력하는 역할을 한다.
- DWORD WINAPI consumer(LPVOID arg);
  소비자 스레드의 시작 지점이다. myread함수를 통해 공유 버퍼에서 데이터를 읽고, 읽기 작업이 끝날 때마다 콘솔에 메시지를 출력한다.
- void mywrite(int n);와 int myread(void);
  각각 공유 버퍼에 데이터를 쓰고 데이터를 읽는 함수이다. 세마포어와 뮤텍스를 사용하여 동기화를 처리하는 역할을 한다.

```c
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N_COUNTER 10

HANDLE semWrite, semRead; // Windows semaphore
int queue[N_COUNTER]; // shared buffer
int wptr = 0; // write pointer for queue[]
int rptr = 0; // read pointer for queue[]
HANDLE mutex; // Windows mutex

DWORD WINAPI producer(LPVOID arg);
DWORD WINAPI consumer(LPVOID arg);
void mywrite(int n);
int myread(void);

// producer thread function
DWORD WINAPI producer(LPVOID 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;
        Sleep(m * 10); // m*10
    }
    return 0;
}
```

3. 동작과정

1) 초기화

main 함수는 뮤텍스와 세마포어를 초기화하고, 생산자와 소비자 스레드를 생성한다.

```
DWORD WINAPI producer(LPVOID arg);
DWORD WINAPI consumer(LPVOID arg);
void mywrite(int n);
int myread(void);
```

2) 생산자 동작

생산자는 semWrite 세마포어를 통해 버퍼에 쓸 공간이 있는지 확인한다.
공간이 있다면, 뮤텍스를 통해 공유 버퍼에 독점적인 접근 권한을 얻고 데이터를 쓴다.
작업이 완료되면 뮤텍스를 해제하고 semRead 세마포어를 증가시켜 소비자가 읽을 수 있도록 알린다.

```
// producer thread function
DWORD WINAPI producer(LPVOID 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;
        Sleep(m * 10); // m*10
    }
    return 0;
}
```

Would you

```
// read a value from the shared memory
int myread() {
    WaitForSingleObject(semRead, INFINITE);
    WaitForSingleObject(mutex, INFINITE);

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

    ReleaseMutex(mutex);
    ReleaseSemaphore(semWrite, 1, NULL);

    return n;
}
```

3) 소비자 동작

소비자는 semRead 세마포어를 통해 읽을 데이터가 있는지 확인한다. 데이터가 있다면, 뮤텍스를 통해 공유 버퍼에 독점적인 접근 권한을 얻고 데이터를 읽는다.
작업이 완료되면 뮤텍스를 해제하고 semWrite 세마포어를 증가시켜 생산자가 쓸 수 있음을 알린다.

```
// consumer thread function
DWORD WINAPI consumer(LPVOID 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;
        Sleep(m * 10); // m*10
    }
    return 0;
}
```

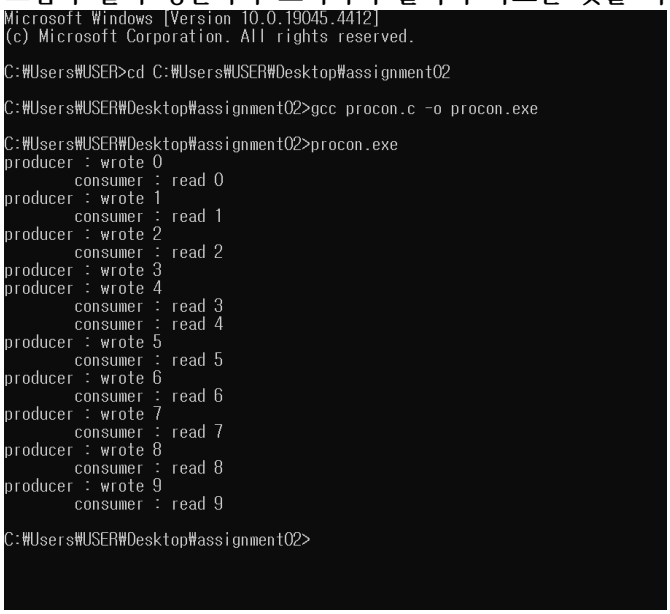
```
// write n into the shared memory
void mywrite(int n) {
    WaitForSingleObject(semWrite, INFINITE);
    WaitForSingleObject(mutex, INFINITE);

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

    ReleaseMutex(mutex);
    ReleaseSemaphore(semRead, 1, NULL);
}
```

4) 완료

모든 작업이 완료되면 main함수는 스레드와 동기화 객체의 핸들을 닫는다.

```
int main() {
    HANDLE t[2]; // thread handles
    srand((unsigned int)time(NULL));

    // Initialize mutex
    mutex = CreateMutex(NULL, FALSE, NULL);

    // Initialize semaphores
    semWrite = CreateSemaphore(NULL, N_COUNTER, N_COUNTER, NULL);
    semRead = CreateSemaphore(NULL, 0, N_COUNTER, NULL);

    // Create the threads for the producer and consumer
    t[0] = CreateThread(NULL, 0, producer, NULL, 0, NULL);
    t[1] = CreateThread(NULL, 0, consumer, NULL, 0, NULL);

    WaitForMultipleObjects(2, t, TRUE, INFINITE);

    // Close handles
    CloseHandle(t[0]);
    CloseHandle(t[1]);
    CloseHandle(mutex);
    CloseHandle(semWrite);
    CloseHandle(semRead);

    return 0;
}
```

4. 어플리케이션 구현
그림과 같이 생산자와 소비자의 출력이 나오는 것을 확인할 수 있었다.



2-1) 소프트웨어 기반 동기화 방식(알고리즘)들에 대해서 간단한 조사를 하고,
그에 대한 설명 작성

소프트웨어 기반 동기화 방식은 여러 프로세스나 스레드가 공유 자원에 동시에 접근할 때 발생할 수 있는 문제를 해결하기 위해 사용된다.
이러한 공유방식은 공유 자원의 일관된 상태를 유지하고, 동시성 관련 오류를 방지하기 위해 설계되었다.

1. 뮤텍스(Mutex)
   - 뮤텍스는 Mutual Exclusion(상호 배제)의 약자로, 한 번의 하나의 스레드만 특정 자원에 접근할 수 있도록 한다. 뮤텍스는 공유 자원을 사용하려는 스레드가 뮤텍스를 '잠그고(lock)' 자원에 접근한 후, 사용이 끝나면 '해제(unlock)'함으로써 다른 스레드가 접근할 수 있도록 한다.
2. 세마포어(Semaphore)
   - 세마포어는 뮤텍스와 유사하지만, 한 번에 여러 스레드가 공유 자원에 접근할 수 있는 허용 수를 정할 수 있는 차이가 있다. 세마포어는 주로 두 가지 종류로 나뉘어진다.
     이진 세마포어(뮤텍스와 유사)와 카운팅 세마포어(동시에 여러 스레드의 접근 허용)
3. 모니터(Monitor)
   - 모니터는 뮤텍스와 조건 변수를 사용하여 동기화를 제공하는 고수준 추상화이다.
     모니터 내의 모든 메서드는 자동적으로 뮤텍스로 보호되므로, 한 번에 하나의 스레드만이 모니터의 메서드를 실행할 수 있다. 또한, 스레드가 특정 조건을 만족할 때까지 대기하게 하는 조건 변수를 사용하여, 복잡한 동기화 문제를 해결할 수 있다.
4. 데커(Dekker)와 피터슨(Peterson) 알고리즘
   - 두 가지 알고리즘들은 소프트웨어 수준에서 상호 배제를 달성하기 위해 개발되었다.
     두 알고리즘 모두 공유 변수를 사용하여 스레드 간의 실행 순서를 조정하며, 스레드가 공유 자원에 접근하려고 할 때 다른 스레드가 접근하지 못하도록 한다.
     피터슨 알고리즘은 두 스레드 간의 상호 배제를 위해 설계되었으며, 데커 알고리즘은 두 개 이상의 스레드에 대해 확장될 수 있다.
5. Lamport 알고리즘
   - 분산 컴퓨팅 환경에서 프로세스 간 시간 동기화를 위해 제안된 알고리즘이다.
     시계 동기화를 위한 별도의 중앙 시계가 필요하지 않고, 메시지 송수신 과정에서 논리적 시계 값을 업데이트 하여 시간 순서를 유지할 수 있다.
     또한 분산 환경에서 발생하는 이벤트의 순서를 정확하게 파악할 수 있다.

Lamport 알고리즘은 분산 시스템 설계 및 구현에 널리 사용되며, 분산 데이터베이스, 분산 파일 시스템, 분산 트랜잭션 처리 등 다양한 분야에 적용되고 있다.

동기화 방식은 공유 자원에 대한 접근을 제어하고, 데이터 경쟁과 같은 동시성 문제를 방지하기 위해 널리 사용된다.

2-2) 조사한 동기화 방식들 중 마음에 드는 방법
[뮤텍스(Mutex)]
- 윈도우 API는 뮤텍스를 직접 지원하므로 C나 C++같은 언어를 사용한다면 쉽게 구현할 수 있다는 정보를 얻었다. 또한, NET 프레임워크를 사용하는 C# 같은 경우에도 뮤텍스를 사용할 수 있다. 뮤텍스는 간단한 상호 배제 요구에 적합한 방식이기 때문에 선택했다.
- 세마포어와 뮤텍스 사이에서 많은 고민을 했다. 세마포어는 윈도우 API와 .NET 프레임워크 모두에서 지원하며, 만약 한 번에 여러 스레드가 자원에 접근해야 하고, 그 접근을 제어해야 한다면 세마포어가 유용할 수 있다. 하지만 구현이 뮤텍스보다 다소 복잡하다고 해서 최종적으로 뮤텍스를 선택했다.

[뮤텍스(Mutex) 구현]
간단하게 뮤텍스를 사용하여 두 스레드가 “sharedResource” 라는 공유 자원에 동시에 접근하지 못하도록 한다. 각 스레드는 WaitForSingleObject 함수를 사용하여 뮤텍스를 얻으려고 시도하고, 뮤텍스를 얻으면 공유 자원에 접근한다. 접근이 끝나면 ReleaseMutex 함수를 호출하여 뮤텍스를 해제하고, 다른 스레드가 접근할 수 있도록 한다.

```
// 공유 자원
int sharedResource = 0;

// 뮤텍스 핸들
HANDLE hMutex;

DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // 뮤텍스를 얻기 위해 대기
    WaitForSingleObject(hMutex, INFINITE);

    // 공유 자원에 접근
    sharedResource++;
    printf("Thread %d has updated the shared resource: %d\n %d\n", GetCurrentThreadId(), sharedR

    // 뮤텍스 해제
    ReleaseMutex(hMutex);

    return 0;
}
```

```
int main() {
    hMutex = CreateMutex(NULL, FALSE, NULL);
    if (hMutex == NULL) {
        printf("Failed to create Mutex\n");
        return 1;
    }

    // 스레드 생성
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // 스레드 종료 대기
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // 핸들 닫기
    CloseHandle(hThread1);
    CloseHandle(hThread2);
    CloseHandle(hMutex);

    return 0;
}
```

2-3) step2에서 구현한 알고리즘을 #1의 문제에서 pthread_mutex 대신 활용하기

```
Microsoft Windows [Version 10.0.19045.4529]
(c) Microsoft Corporation. All rights reserved.

C:₩Users₩USER>cd C:₩Users₩USER₩Desktop₩assignment02

C:₩Users₩USER₩Desktop₩assignment02>procon2.exe
Thread 8360 has updated the shared resource: 1
 0
Thread 20324 has updated the shared resource: 2
 0

C:₩Users₩USER₩Desktop₩assignment02>procon.exe
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
producer : wrote 4
        consumer : read 3
        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

C:₩Users₩USER₩Desktop₩assignment02>
```

둘의 성능비교를 위해 두 파일 모두 cmd에서 실행을 시켜보았다.

1. procon2.exe
   - 멀티스레드 환경에서 작동하고 있으며, 8360과 20324가 공유 자원을 업데이트 하고 있다. 공유 자원에 대한 접근을 동기화하여 안전하게 처리하고 있다.
2. procon.exe
   - 단일 프로세스 내에서 producer와 consumer역할을 수행하고 있다. producer는 0부터 9까지의 숫자를 순서대로 출력하고, consumer는 이를 읽어 출력하는 것을 볼 수 있다. producer와 consumer 간 데이터 전송을 위해 버퍼를 사용하고 있다.
     버퍼를 사용하여 producer와 consumer의 실행 속도 차이를 어느 정도 흡수한다.
3. 성능 비교
   - procon2.exe는 멀티스레드 환경에서 동작해, 병렬 처리를 통해 전반적인 속도가 향상될 수 있다.
   - 하지만 procon.exe는 단일 프로세스 환경에서 동작하므로, 전체적인 처리 속도는 procon2.exe보다 느릴 수 있다.
   - procon.exe는 consumer와 producer간 데이터 전송을 위해 버퍼를 사용해서, 실행 속도 차이를 어느정도 해결 할 수 있다.
   - 멀티스레드 환경에서 병렬처리를 통해 작동하는 procon2.exe도 성능이 우수하지만, procon.exe도 버퍼를 활용하여 생산자와 소비자간 속도 차이를 극복하고 있다.

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

3-1) page.c의 코드를 컴파일 하고 실행시켜보기

```
PS C:₩Users₩USER₩Desktop₩assignment02> gcc -o page.exe page.c
PS C:₩Users₩USER₩Desktop₩assignment02> .₩page.exe
PS C:₩Users₩USER₩Desktop₩assignment02>
```

3-2) 컴파일된 파일은 pagesize 변수를 인자값으로 받는다. 값을 변경해가며 실행시간의 변화를 확인해보자.

```
PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 199 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 309
Ticks             : 3091872
TotalDays         : 3.57855555555556E-06
TotalHours        : 8.58853333333333E-05
TotalMinutes      : 0.00515312
TotalSeconds      : 0.3091872
TotalMilliseconds : 309.1872



PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 200 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 292
Ticks             : 2920441
TotalDays         : 3.3801400462963E-06
TotalHours        : 8.11233611111111E-05
TotalMinutes      : 0.00486740166666667
TotalSeconds      : 0.2920441
TotalMilliseconds : 292.0441



PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 1000 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 311
Ticks             : 3112263
TotalDays         : 3.60215625E-06
TotalHours        : 8.645175E-05
TotalMinutes      : 0.005187105
TotalSeconds      : 0.3112263
TotalMilliseconds : 311.2263



PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 1001 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 346
Ticks             : 3462163
TotalDays         : 4.00713310185185E-06
TotalHours        : 9.61711944444444E-05
TotalMinutes      : 0.00577027166666667
TotalSeconds      : 0.3462163
TotalMilliseconds : 346.2163



PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 1002 }
```

step3) 컴퓨터의 페이지 크기 확인해보기
계속 숫자를 입력해가며 페이지를 찾으려고 노력했지만, 찾지 못해서 결국 페이지사이즈를 코드를 입력해 직접 찾았다.

```
PS C:₩Users₩USER₩Desktop₩assignment02> ./page.exe
PS C:₩Users₩USER₩Desktop₩assignment02> [System.Environment]::SystemPageSize
4096
PS C:₩Users₩USER₩Desktop₩assignment02>
```

페이지의 크기는 4096이다. 4096일때의 시간을 다시 체크해보니 갑자기 급격히 시간이 증가하는 것을 알 수 있었다.

```
PS C:₩Users₩USER₩Desktop₩assignment02> Measure-Command { .₩page.exe 4096 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 801
Ticks             : 8016025
TotalDays         : 9.27780671296296E-06
TotalHours        : 0.000222667361111111
TotalMinutes      : 0.0133600416666667
TotalSeconds      : 0.8016025
TotalMilliseconds : 801.6025
```

내 컴퓨터 운영체제의 페이지 크기는 4096bytes(4KB)이고, 운영체제가 물리 메모리를 4096bytes 단위의 페이지로 나누어 관리한다고 볼 수 있다.
가상 메모리 주소 공간도 4096bytes 단위의 페이지로 나누어진다. 이를 통해 프로세스가 사용할 수 있는 가상 메모리 공간을 효과적으로 관리할 수 있다.
프로세서가 메모리에 접근할 때, 페이지 단위로 데이터를 읽어오거나 쓰게 된다. 4096bytes 크기의 페이지는 메모리 접근 성능 향상에 적합한 크기이다.
운영체제는 메모리에 없는 데이터를 디스크에서 읽어올 때, 4096bytes 단위로 페이지를 로드한다.

[과제 소감]
이번 과제를 통해 운영체제의 핵심 개념인 메모리 관리, 페이징, 가상 메모리 등을 실제로 구현해볼 수 있었다. 이론으로만 배웠던 내용들이 실제 코드를 통해 구현되는 과정을 경험하며, 운영체제의 작동 원리를 보다 깊게 이해할 수 있었다.

과제를 수행하면서 page.c의 페이지를 확인하려고 명령어를 cmd에서 입력했었는데, cmd에서는 이상하게 페이지 확인 명령어 컴파일 오류가 발생했다. 왜 그런지 찾아보니, Windows 명령 프롬포트(층)는 Windows API 중 일부 기능만을 지원한다. PowerShell은 .NET 프레임워크를 기반으로 하여, 더 풍부한 API 기능을 사용한다고 한다.
페이지 크기 확인을 위한 sysconf(_SC_PAGESIZE)함수는 POSIX 표준 API이다.
PowerShell은 이걸 지원하지만 cmd는 지원하지 않는다고 한다.

윈도우 운영체제에 대한 지식을 향상시킬 수 있는 유익한 과제였다.