Semaphore 과제

수리과학부 2019-16022 박채연

1. The goal of the assignment

본 과제의 주된 목적은 thread와 process의 차이 및 thread의 의도적이지 않은 공유로 인해 발생할 수 있는 문제점을 이해하는 것이다. 강의 자료에 제공된 'goodcnt.c'와 'badcnt.c' 코드를 직접 작성해보면서 'pthread_create()', 'pthread_join'과 같은 대표적인 Posix thread standard interface를 익히고, thread의 code, data 및 kernel context 공유로 인해 발생할 수 있는 문제의 해결방안을 이해한다.

2. 실행 결과

.badcnt.c'를 'gcc -o bad badcnt.c -lpthread' 명령어를 통해 컴파일하여 'bad'이라는 이름의 실행파일을 만들고 niters가 100, 1000, 10000, 100000, 1000000, 10000000, 100000000일 때 실행결과를 출력하였다. 이때, 시간 단위는 밀리초이다. niters가 100, 1000일 때는 OK가 출력되지만 이보다 큰 값일 때는 'BOMB'가 출력되는 것을 확인가능하다. 이는 두 개의 thread가 각각 한 번에 1만큼 총 niters번씩 cnt를 증가시킬 때 niters 값이 작으면 progress graph에서 trajectory가 unsafe region을 침범하지 않지만 niters 값이 크면 unsafe region을 침범할 가능성이 커진다는 것을 의미한다. 또한, niters 값이 증가할 때 대체적으로 프로그램 실행시간 역시 증가함을 확인 가능하다.

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 100
Total Running Time : 107
OK cnt = 200
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 1000
Total Running Time : 125
OK cnt = 2000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 10000
Total Running Time : 91
BOOM! cnt = 19766
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 100000
Total Running Time : 94
BOOM! cnt = 106949
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 1000000
Total Running Time : 3419
BOOM! cnt = 1042718
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 10000000
Total Running Time : 60462
BOOM! cnt = 10353780
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming$ ./bad 100000000
Total Running Time : 692016
BOOM! cnt = 126255642
```

'goodcnt.c'를 'gcc -o good goodcnt.c -lpthread' 명령어를 통해 컴파일 하여 'good'이라는 이름의 실행파일을 만들고 niters가 100, 1000, 10000, 100000, 1000000, 10000000, 100000000일 때 실행결과를 출력하였다. 이때, 시간 단위는 밀리초이고, 모든 경우에 대해 'OK'가 출력되는 것을 확인가능하다. 이는 두 개의 thread가 각각 한 번에 1만큼 총 niters번씩 cnt를 증가시킬 때 progress graph에서 trajectory가 unsafe region을 침범하지 않았음을 의미한다. 또한, niters 값이 증가할 때 프로그램 실행시간 역시 증가하며 같은 niters 값일 때 badcnt.c의 실행시간보다 더 오

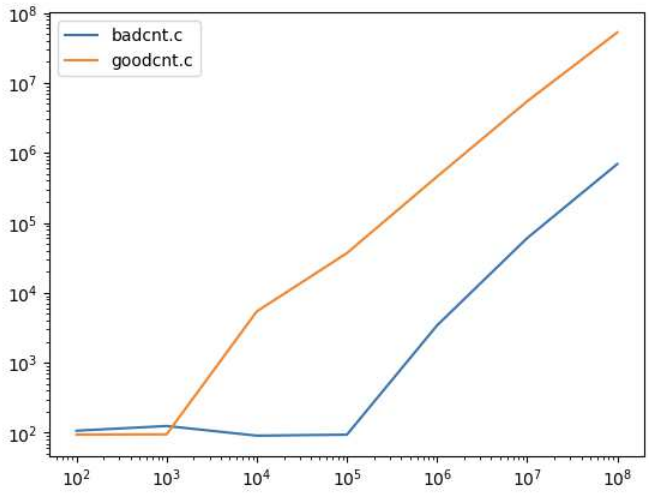
래걸린다는 것을 확인 가능하다.

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 100
Total Running Time : 94
OK cnt = 200
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 1000
Total Running Time : 95
OK cnt = 2000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 10000
Total Running Time : 5415
OK cnt = 20000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 100000
Total Running Time : 36904
OK cnt = 200000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 1000000
Total Running Time : 457136
OK cnt = 2000000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 10000000
Total Running Time : 5446374
OK cnt = 20000000
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming$ ./good 100000000
Total Running Time : 52475599
OK cnt = 200000000
```

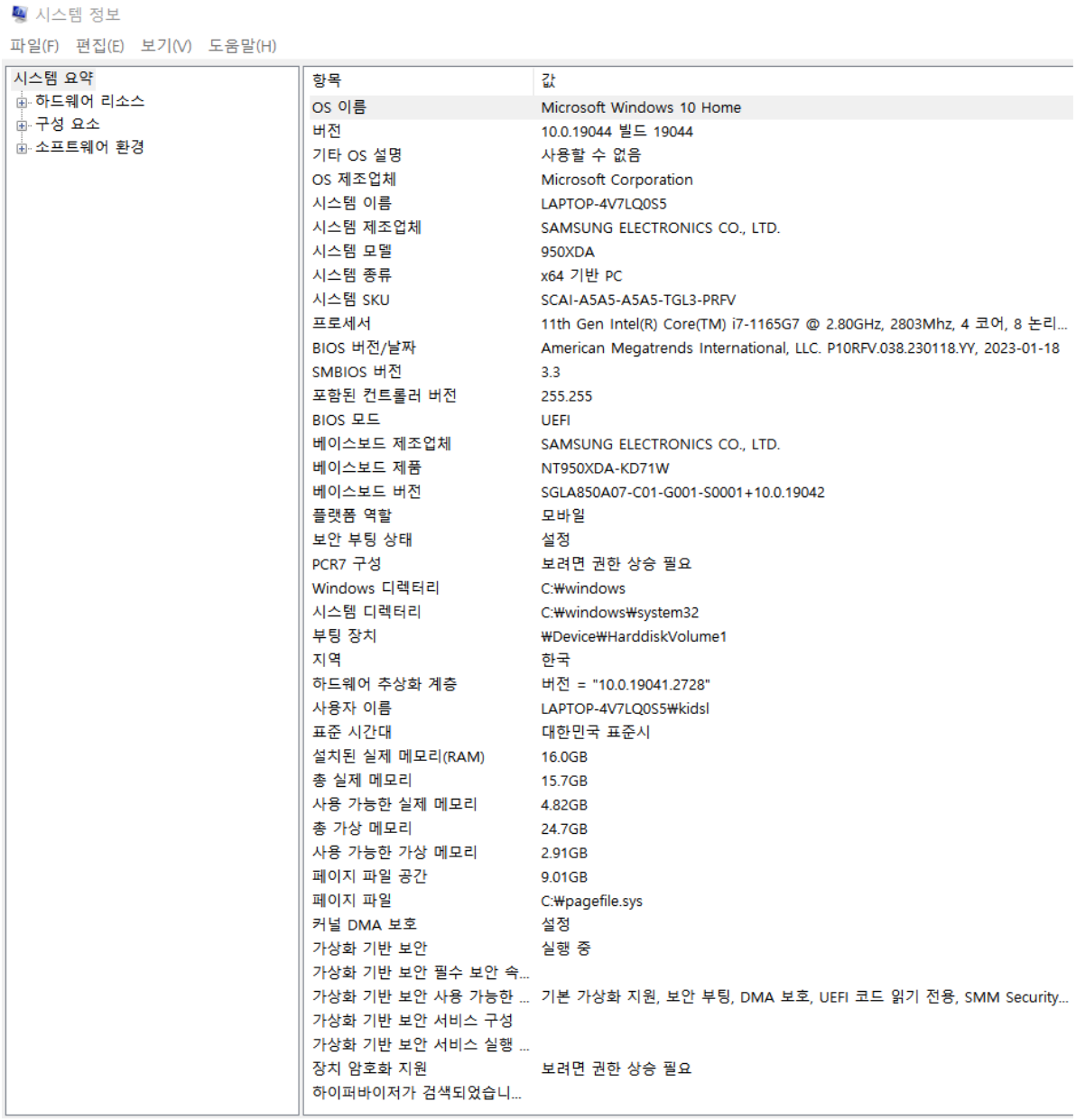
파이썬의 matplotlib 라이브러리를 이용하여 아래와 같이 코드를 작성하여 niters값에 따른 badcnt.c 와 goodcnt.c의 실행시간을 그래프로 나타내었다.

```
import matplotlib.pyplot as plt

x=[10**i for i in range(2, 9, 1)]
y1=[107, 125, 91, 94, 3419, 60462, 692016]
y2=[94, 95, 5415, 36904, 457136, 5446374, 52475599]

plt.loglog(x, y1, label='badcnt.c')
plt.loglog(x, y2, label='goodcnt.c')

plt.legend(loc=2)
plt.show()
```

위의 코드를 실행시킨 결과는 아래와 같다. niters 값이 커질수록 badcnt.c와 goodcnt.c의 실행시간이 증가하고, 일반적으로 goodcnt.c의 실행시간이 badcnt.c의 실행시간보다 훨씬 오래 걸린다는 것을 한눈에 확인가능하다.



3. 컴퓨터 사양



윈도우의 '시스템 정보' 를 통해 살펴본 컴퓨터(노트북)의 사양은 위의 이미지와 같다. 위의 사양을 가지는 윈도우 컴퓨터(노트북)에서 'WSL2'를 통해 'badcnt.c'와 'goodcnt.c'를 실행시켰는데, 'uname -r'을 통해 우분투 버전을 확인해보면 아래와 같다.

```
chaeyeon@LAPTOP-4V7LQ0S5:/mnt/c/windows/system32$ uname -r
5.15.90.1-microsoft-standard-WSL2
```

4. 어려웠던 점

badcnt.c 코드와 goodcnt.c 코드가 강의자료에 제시되어있었기 때문에 코드를 작성하는 과정에서 큰 어려움은 없었다. 다만 아래와 같은 시행착오 때문에 많은 시간을 소모하였다.

'VS code' 내에서 #include<pthread.h> 헤더파일을 include 하고 강의자료와 동일하게 코드를 작성한 후 'gcc -o bad badcnt.c'와 같이 컴파일 했을 때 아래와 같은 에러메시지가 떴다.

```
/usr/bin/ld: /tmp/cckCdQM8.o: in function `Pthread_create':
goodcnt.c:(.text+0x70): undefined reference to `pthread_create'
/usr/bin/ld: /tmp/cckCdQM8.o: in function `Pthread_join':
goodcnt.c:(.text+0xb9): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
```

이를 해석해보면 pthread_create와 pthraed_join 함수에 대한 참조를 찾지 못했다는 뜻인데, <pthread.h> 헤더 파일 내에 이 두 개의 함수가 모두 정의있기 때문에 에러 원인을 이해하기 어려웠다. 해당 메시지를 복사하여 구글링을 해보니 <pthread.h> 헤더 파일이 소스 파일 내에 포함되어 있을 때 gcc의 플래그로 '-o'를 사용하게 되면 뒤에 -lpthread를 붙여 'gcc -o bad badcnt.c -lpthread'와 같이 컴파일해야 해당 에러가 발생하지 않는다는 것을 알게 되었다.

정상적으로 컴파일까지 마친 뒤 './bad' 명령어를 통해 실행시켜보았으나 'Segmentation fault'가 떴고, VS code내에서도 '#include <pthread.h>'에 빨간 밑줄이 그어있었다. 헤더 파일 자체의 문제라고 착각하여 Visual studio 2019 버전에서 해당 코드를 돌려보았으나 아래와 같은 에러메시지가 떴고, 이는 헤더파일에 문제가 있다고 확신하게 만들었다.

```
빌드 시작...
1>------ 빌드 시작: 프로젝트: Project14, 구성: Debug Win32 ------
1>code.c
1>C:₩Users₩kids1₩source₩repos₩Project14₩Project14₩code.c(2,10): fatal error C1083: 포함 파일을 열 수 없습니다. 'pthread.h': No such file or directory
1>"Project14.vcxproj" 프로젝트를 빌드했습니다. - 실패
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

https://webnautes.tistory.com/1452 링크를 참고하여 프로젝트 설정을 마쳤더니 #include <pthread.h> 밑의 빨간 줄은 사라졌지만 여전히 아래와 같은 에러메시지가 떴다.

```
빌드 시작...
1>------ 빌드 시작: 프로젝트: Project15, 구성: Debug Win32 ------
1>LINK : fatal error LNK1104: 'pthreadV32.lib' 파일을 열 수 없습니다.
1>"Project15.vcxproj" 프로젝트를 빌드했습니다. - 실패
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

다시 VS code로 돌아와 차분히 코드를 읽어보던 중 인자로 'niters'가 전달하지 않았기 때문에 발생한 문제임을 알게 되었다. './bad 1000'과 같이 niters를 인자로 전달하니 정상적으로 동작하는 것을 확인하였다.

5. 신기했던 점 및 새롭게 배운 점

niters 값이 커질수록 badcnt.c와 goodcnt.c의 실행시간이 증가하고, 같은 niters 값에 대해 일반적으로 goodcnt.c의 실행시간이 badcnt.c의 실행시간보다 훨씬 오래 걸린다는 것을 확인하였다. 이는 goodcnt.c에서 semaphore를 통해 progress graph에서 trajectory가 unsafe region을 침범하는 것을 방지하기 때문이다. 하나의 thread가 cnt 값을 증가시키고 V 함수을 호출하기 전까지 P 함수는 다른 thread가 cnt 값을 증가시키지 못하도록 막는다. 즉, 하나의 thread가 cnt를 증가시키고나서 다시 V 함수를 호출할 때까지 다른 thread는 기다리고 있어야 하므로 실행시간이 더 길어

지는 것이다.

이번 과제를 수행하면서 thread는 process의 hierarchy 구조와는 다르게 code, stack, kernel context를 공유하는 'pool of peers' 구조를 가진다는 것을 알게되었다. 이러한 구조 때문에 여러 개의 thread가 전역 변수 등의 공유 자원에 동시에 접근할 때 원하지 않는 결과를 얻게 될 가능성이 있다. Semaphore는 progress graph에서 trajectory가 unsafe region에 침범하여 사용자가 원하지 않는 결과를 얻는 것을 방지해주는 해결 방안임을 배웠다.

조금 더 자세히 설명하자면, P operation은 semaphore의 값이 0이면 해당 thread를 lock 하는 역할을 하고, 1이면 lock 하지 않고 semaphore의 값을 0으로 바꾼다. V operation은 semaphore의 값을 0에서 1로 증가시키는 역할을 한다. 편의상 두 개의 thread를 각각 A, B라 하자. 처음에 semaphore의 값을 1로 초기화하고, thread A가 P operation을 수행하면 이는 lock 되지 않고 semaphore의 값을 0으로 바꾼다. thread A가 cnt에 접근하는 동안 만약 thread B가 P operation을 수행하면 thread A가 cnt값을 증가시키고 V operation을 수행하여 semaphore의 값을 다시 1로 증가시킬 때까지 thread B는 lock 되어 있기 때문에 progress graph에서 trajectory가 unsafe region에 침범하는 것을 막을 수 있는 것이다. 또한, Semaphore 덕분에 thread가 code, stack 및 kernel context를 공유함으로써 발생할 수 있는 문제를 방지할 수 있지만 프로그램의 실행시간은 증가한다는 단점이 있음을 배웠다.

4번(어려웠던 점)에서 설명한 시행착오 때문에 약 1시간 동안 헤맸지만 코드작성 습관을 되돌아볼 수 있는 기회를 가졌다. 실행파일 실행 시 Segmentation fault가 발생했을 때 해당 이슈가 코드의 어떤 파트에서 발생했는지 한 눈에 알아보기 어려웠기 때문에 겪은 시행착오였다. 아래와 같이 argc가 2인지 확인하는 코드를 삽입하여 만약 2가 아니라면 에러메시지를 출력하도록 함으로써 더 좋은 코드를 작성할 수 있을 것이다. 앞으로는 코드를 작성할 때 프로그래머의 실수로 인한 에러가 발생할 위험이 있는 곳에 적절히 아래와 같은 코드를 삽입하여 문제를 방지해야겠다는 생각이 들었다.

```
int main(int argc, char **argv){

    if (argc != 2){
        printf("An argument 'niters' was not passed.\n");
        return -1;
    }
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;
    time_t start, end;
```