Operating System

Homework 2:

Report of Thread

< 0 free-days left>

Name : SeokBeom Lim (임석범)

Department : Mobile System Engineering

Student number : 32203743

Table of contents

서론

현대 Computer System에서 거의 대부분의 작업들은 병렬처리를 통한 효율적 설계를 따른다. 이 설계 중 Multi Thread Programming은 하나의 Process 내에서 여러 Thread를 동시에 실행함으로써, 병렬처리의 이점을 얻을 수 있게 설계 되었다. 이는 CPU 자원을 더욱 효율적으로 활용 할 수 있게 해주면 대규모 데이터 처리에서도 그 효과를 극대화 할 수 있다. 본 보고서는 Multi-Thread Programming Programming 과 관련된 기본 개념을 다룬 후 Producer-Consumer 구조의 대규모 데이터 통계 처리를 위한 Program의 설계와 최적화 과정에 대해서 다루고자 한다.

본론

1. General Concept of Thread

1) Concept of Thread

Thread 는 CPU에서 실행 할 수 있는 가장 작은 실행 단위이며 Process를 구성하는 실행 흐름의 단위이다. Thread는 Process 내에서 실행이 되며, 메모리 영역의 Hardware 자원을 공유한다. 하나의 Process가 하나의 일을 하는 것이 아닌 여러 일을 할 수 있게 만들어주는 것도 Thread를 통해 가능하게 한다.

전통적인 관점에서 하나의 Process 는 한번에 하나의 일만 처리 할 수 있었다. 이러한 Process를 단일 스레드 프로세스라 부른다. 하지만, Thread의 개념이 도입 되며 하나의 Process가 여러 일을 동시에 처리할 수 있는 개념이 도입되었다. 즉, 하나의 Process로 여러가지 일을 수행 할 수 있도록 만들었다. 이러한 개념은 One Process, Multi Thread에 대한 개념과 관련 있다. 하나의 Process가 여러 일을 하기 위해 여러 Thread를 통해 여러 Instruction 을 동시에 실행 할 수 있게 한다.

Process와 Thread의 차이점에 대해서 알아보면, Process는 독립적인 Memory 영역을 가지며 서로의 Memory Access를 허용 하지 않는다. 이에 반면, Thread는 하나의 Process 내부에서 Memory를 공유하며 Process 보다 작은 작업 단위로서 빠르게 실행되고 종료 될 수 있다. 또한 Thread는 자기 자신의 Stack과 TCB를 가진다. Thread Control Block(TCB) 는 Thread ID, Status, Priority, Stack Pointer 등의 Thread 관련 값을 저장하여 OS가 Thread를 관리하고 Scheduling 할 수 있도록 돕는다.

OS의 가장 중요한 기능 중 하나는 Process 관리이다. OS는 여러 Process가 동시에 실행될 수 있도록 스케줄링(Scheduling)하고, 각 Process가 적절하게 자원을 사용할 수 있도록 관리한다. 위 Scheduling이 중요한 부분을 차지하는 Multitasking은 여러 Process가 동시에 실행되는 것을 가능하게 한다. 이때 Processor Scheduler는 각 Process의 우선순위와 실행 시간을 고려하여 CPU의 작업 순서를 결정한다.

Thread는 빠른 생성과 종료를 할 수 있으며, Thread는 같은 Process 내의 Memory와 HW resource를 공유하기 때문에 서로 간의 Data 접근에 용이하다. 여러 Thread를 사용 할 시 CPU의 여러 Core를 활용 할 수 있어 프로그램의 성능을 높일 수 있다. 이러한 Thread 사용시 생길 수 있는 문제점은, Thread간의 자원 공유 문제와 높은 디버깅 난이도가 있다. 여러 Thread간 자원을 공유하기 위해서 Data의 접근과 동기화를 생각해야 하며, 잘못된 동기화는 Data의 오염이 있을 수 있다. 또한, 동기화 문제를 해결하기 위한 디버깅이 까다롭다. Thread는 진행 순서가 매번 다르므로 문제가 생기는 부분을 정확하게 파악하기 힘들다.

2) Multi-Thread

한 Process에서 여러 Thread 를 동시에 작업 할 수 있게 하는 기술을 Multi-Threading 이라고 한다. Multi-Thread는 CPU에 의해 Scheduling 되며, 주로 OS가 Thread의 우선순위와 상태를 관리하여 실행 순서를 결정한다. 이 과정에서 생길 수 있는 단점이 있다. 아래는 Multi-Threading으로 발생할 수 있는 3가지 문제와 해결책이다.

첫번째는 Context Swithching Overhead 이다. Context Switching Overhead는 CPU가 한 Thread나 Process의 실행을 중지하고 다른 Thread나 Process를 실행 하기 위해서 수행해야 하는 작업으로 인해 발생 할 수 있는 성능 저하를 의미 한다. 이러한 Overhead를 막기 위해서는 Thread와 Process의 사용을 최소화 해야한다. 즉, 불필요한 Thread와 Process를 줄이는 것이 Overhead를 줄이는데 도움이 된다. 또한, 효율적인 Scheduling Policy 를 사용하는 것 또한 Overhead를 막을 수 있다. Thread는 두가지로 분류 할 수 있는데, I/O 작업을 주로 수행하는 Thread와 CPU 작업을 주로 수행하는 Thread이다. I/O 작업을 주로 수행하는 Thread의 대기 시간이 길기 때문에, 이를 먼저 실행하고 그 다음으로 CPU 위주 Thread를 수행 하게 하면 대기 시간이 크게 줄어 들 수 있다.

두번째는 Race condition 이다. Race condition은 여러 Thread가 동일한 자원을 동시에 수정하 수정 할 때 발생 하는 문제이다. 이러한 Race Condition 은 Shared object의 값을 의도 하지 않는 값으로 이끌 수 있다. 이 문제를 해결하기 위해 Thread Synchronization 이 요구 된다.

세번째는 Deadlock 이다. Deadlock은 두개 이상의 Thread가 서로의 자원을 점유한 상태에서 필요 자원이 들어오기를 기다리는 상태이다. 이러한 상태는 여러가지 이유가 있는데, 상호배제(Mutual exclusion) 를 보장하지 않은 프로그램에서 서로 다른 Thread가 자원을 점유 한채 다른

자원을 기다리는 상태이기 때문이다. 또한, Process가 자원을 강제로 빼았지 못하는 비선점 Process 인 경우 Deadlock 상태는 지속되게 된다. 위 문제를 해결하기 위해 Thread Synchronization이 요구 된다.

3) Synchronization of Thread

Multi-Thread Programming 에서 모든 Thread는 Process의 Address Space를 공유하는데 이로 인해, 각 Thread가 같은 Memory 위치에 접근 할 수 있으므로 이를 이를 해결하기 위해 Synchronization Alogirhm을 사용한다. 즉, 여러 Thread가 Memory 공간을 공유할 때, 동시 접근을 조정해야 Data 무결성을 보장 할 수 있다. Assembly 언어와 같이 저급 언어에서는 임계구역(Critical Section)을 실행 시 Context Switching 과정에서 문제가 발생한다. 아래는 Context Swithch로 발생하는 부분의 예시이다.

Producer():

Buffer 에 Data 삽입.

SUM 변수 1 추가

```
move r1 SUM
addi r1 r1 1
move SUM r1
```

Consumer():

Buffer에서 Data 추출

SUM 변수 1 감소

```
move r2 SUM
addi r2 r2 -1
move SUM r2
```

초기 상태 : SUM = 5

위의 자료는 Producer 와 Consumer관계의 Pseudocode 와 그에 해당하는 저급 언어 부분이다. Producer는 Buffer에서 Data를 삽입하여 SUM 변수에 1을 추가하며, Consumer는 Buffer 에

서 Data를 추출하고 SUM 변수를 1감소 시키는 역할을 한다. 위 상황에서, 생길 수 있는 문제는 다음과 같다. Producer assembly 언어에서 addi r1 r1 1 까지 진행이 된 상태에서 Thread Context Swich 가 발생하여, Producer Thread에서 Consumer Thread 로 넘어 갔을 때, Consumer Thread에서도 마찬가지로, addi r2 r2 -1 까지 진행한후 Producer로 넘어가게 된다면, Producer Thread의 move SUM r1 으로 인해 SUM 값은 5에서 6이 되고 이후 Consumer Thread로 넘어가게 되면, 총합은 addi r2 r2 -1 의 결과인 4가 SUM 변수에 저장되게 된다. 이를 표로 나타내면 다음과 같다.

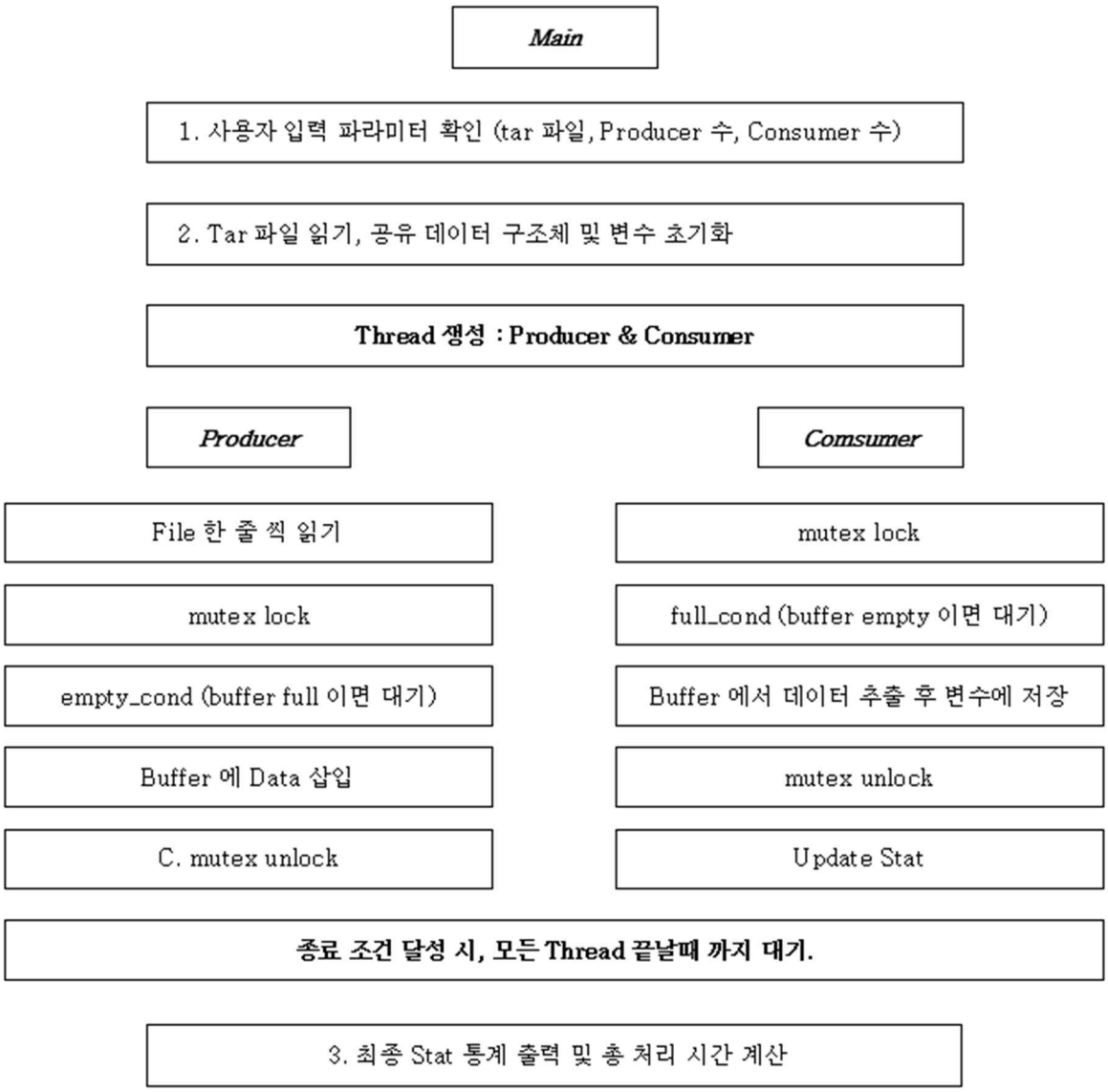
Producer	Consumer	r1	r2	SUM
move r1 SUM		5	5	5
addi r1 r1 1		6	5	5
(context switch)		6	5	5
	move r2 SUM	6	5	5
	addi r2 r2 -1	6	4	5
	(context switch)	6	4	5
move SUM r1		6	4	6
	move SUM r2	6	4	4
				SUM = 9

위와 같은 문제를 위해 사용되는 대표적인 기법은 Mutex Lock이 있다. Mutex Lock은 하나의 Thread만 Shared Object에 접근할 수 있게 한다. 하나의 Thread에 자원을 점유 중인 동안 다른 Thread는 대기 상태를 가지며, Lock 이 해제 되면, 대기 중인 Thread 중 하나가 선택되어, 접근 권한을 얻게 된다. mutex lock의 경우 <pthread.h> 헤더 파일의 pthread_mutex_lock 함수를 통해 구현 할 수 있으며, lock 을 푸는 함수는 pthread_mutex_unlock이다. Mutex Lock에 조건 변수를 사용 할 수 있는데 조건 변수는 특정 조건이 만족될 때까지 Thread를 대기 상태로 전환 하며, 조건이 충족되면 대기중인 Thread를 깨어나게 한다. 이 조건 변수의 경우 while 루프와 함께 pthread_cond_wait 함수를 통해 사용 할 수 있다. 마지막으로, pthread_cond_signal 은 특정조건

이 만족 될 때 까지 Thread를 대기 상태로 만든 후 조건이 충족되면 대기 상태인 Thread를 깨우기 위해 사용된다. 해당 함수는 대기중 Thread에 조건이 충족되었음을 알리는 신호를 보내는 함수이다.

2. Structure of Code

본 보고서의 Code는 Data를 읽고 Text 통계를 분석하는 Multi-thread Producer-Consumer 기반의 프로그램이다. 해당 프로그램은 POSIX Thread 를 활용 하여 병렬 처리를 수행한다. Thread의 Mutex lock과 조건 변수를 사용하여 Thread의 안전성과 Data의 무결점을 보장한다. 본 Code



의 주요 목표는 Multi-Thread를 활용 하여, 통계를 정확히 수집하고, 성능 최적화를 통해 실행 시간을 줄이고자 한다. 다음은 Code의 전반적인 구조이다.

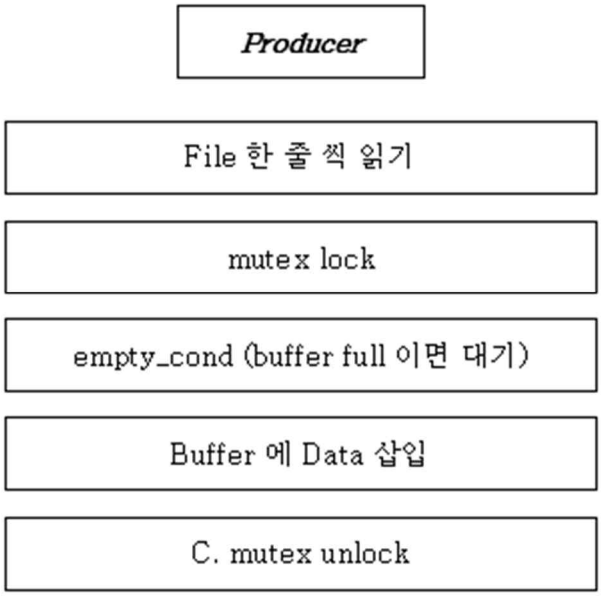
위에 대한 자세한 코드를 살펴보면 다음과 같다.

[so_t 구조체]

```
typedef struct sharedobject {
    char *lines[BUFFER_SIZE];        // 여러 줄을 저장하는 버퍼
    int head, tail, count;           // 버퍼의 시작, 끝, 현재 데이터 수를 관리
    pthread_mutex_t lock;            // 버퍼에 대한 접근을 제어하는 뮤텍스
    pthread_cond_t full_cond;        // 버퍼가 가득 찬 경우의 조건 변수
    pthread_cond_t empty_cond;       // 버퍼가 비었을 때의 조건 변수
    int prod_count;                  // 남은 Producer의 수
    int prod_done_count;             // 종료된 Producer의 수
    int fd;                          // tar 파일 디스크립터
} so_t;
```

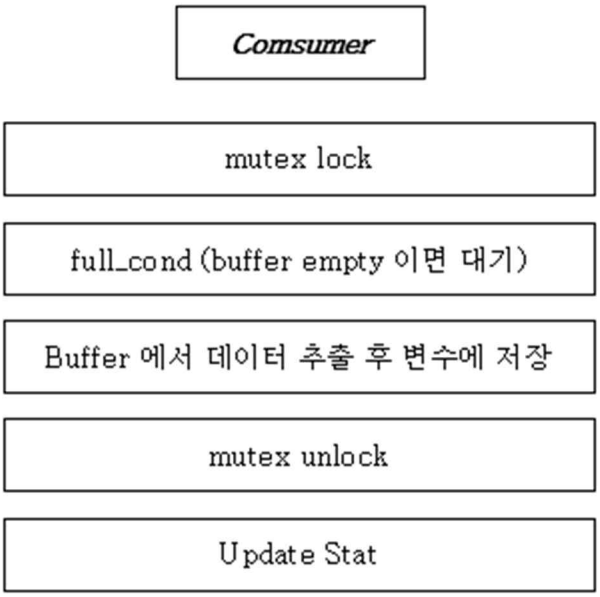
위 구조체는 Share Object 구조체이며, line 포인터를 통해 File의 값을 저장하는 공유 파일을 load 하고 store 한다. pthread_cond_t full_cond 와 pthread_cond_t empty_cond 를 통해 버퍼의 상태를 확인하여 해당 Thread가 대기상태를 머물지, Thread를 진행 할지 결정한다. 이후, head, tail, count를 통해 Producer와 Consumer Thread의 Data의 인덱스를 관리하여 Data를 처리하게 된다. 마지막으로, 안전한 종료를 위해 prod_count와 prod_done_count를 사용한다. 대기 상태 조건도 중요하지만, 종료상태에 두 변수를 배치한 이유는 제대로 된 종료 조건이 없으면, 해당 Thread는 무한 루프(Deadlock 으로 추정)를 가지는 현상이 있었기 때문이다.

[Producer]



Producer Thread의 경우, File을 한 줄 씩 읽고, 이후 mutex lock을 통해 Critical Section을 잠그고, 다른 Thread 가 접근하지 못하게 한다. 이후 Buffer가 가득 차 있으면, cond_wait을 통해 대기 상태에 있다가, 대기 상태가 풀리게 되면, File로부터 읽은 Data를 Buffer 에 삽입 한다. 이후 Mutex Unlock을 통해, Producer Thread는 다른 루프를 가게 된다.

[Consumer]



Consumer Thread의 경우, Producer Thread와 다르게, mutex_lock을 먼저 하고, 공유자원인 Buffer로부터 값을 읽어 들인 후 Temporary 변수에 변수를 저장해준다. 직후 바로 Mutex Unlock 을 실행 하여 준다. Unlock 이후 저장된 Temporary 변수의 값을 통해 Stat을 최신화 해준다. 위와 같이 진행하는 이유는 mutex_lock 과 unlock의 사이의 연산을 최소화 시켜, 대기 시간을 줄이기 위해서 이다. 즉, 임시 변수에 저장한 이후, unlock을 한후 update stat을 진행 하는것이 성능면에서 뛰어나다. 또한, Update Stat 또한 마찬가지로 다른 여러 Thread 들과 공유하는 공유 값이기

때문에, mutex_lock, unlock을 해주어야 한다. 따라서, Consumer 의 lock 과 unlock 사이에 Update Stat 이 들어간다면, 이중으로 대기를 할 수 있기 때문에, 해당 부분은 Consumer를 Unlock 해준 후 Update stat을 진행하는 것이 합리적이다.

3. Experiments of Code

본 장은 코드의 최적화 과정을 위한 여러 실험을 진행한 과정을 보여주는 장이다. 총 5가지의 주제로 실험을 진행 하였다. 모든 코드는 여러 Producer와 Consumer를 입력 받고 돌릴 수 있게 설계하였다.

1) Single Buffer (single_buffer.c <이전 버전>)

실험 초기에 싱글 버퍼(char* line)를 사용 하였으며, 그 결과 Thread 간의 병렬성이 제대로 발휘되지 않아 성능이 기대 이하로 나타났다. Producer 와 Consumer 간의 수를 달리하여 실험을 진행한 결과 Thread 수에 따른 성능 향상은 미미 했으며, 결과적으로 처리량이 크게 개선되지 않는 문제가 발생 하였다. 다음 두 표는 해당 부분의 실험 결과 이다.

Producer	Consumer	Throuput
1	1	6.473879
1	2	6.650707
1	3	6.701772
1	4	6.687051
1	5	6.706404
1	6	6.724777
1	7	6.715712
1	8	6.404807
1	9	6.689855
1	10	6.652289

Producer	Consumer	Throuput
2	1	5.499975
23	3	5.644321
27	2	5.661775
30	2	5.675159
29	2	5.688873
27	3	5.706149
26	3	5.706647
30	3	5.71452
25	3	5.715837
29	4	5.721217

해당 실험은 Tar file에서 가장 큰 파일(100000줄)을 기준으로 실험했으며, Thruput은 가장 큰 파일이 돌아가는데 걸린 시간이다. 해당 실험 결과를 보면, Consumer 개수의 증가에도 Thruput

이 증가하지 않는 것을 확인 할 수 있다. 두번째 표는 Producer 와 Consumer의 개수를 1부터 30까지 모두 돌려본 결과 중 가장 좋은 Throuput 상위 10개를 가져온 것이다. 이 결과에 대해서 분석하면, consumer가 적은 것 외에 Producer의 값이 아무 상관관계를 가지지 않는 것을 확인 할 수 있다. 따라서, 해당 실험은 잘못된 코드로 인한 실험이라 결론을 내렸다.

2) Single Buffer & Multi Buffer Multi_buffer.c)

첫번째 실험을 통해 Single Buffer는 Producer와 Consumer가 증가해도 Throuput이 증가하지 않고 상관 관계를 찾을 수 없다는 결론을 지었었다. 이에 따라, Buffer의 개수를 늘리는 방향으로 코드를 최적화 하기로 하였다. 또한, 첫번째 실험이 100000줄을 처리하는데 시간이 오래 걸리는 것을 확인 할 수 있다. 이는, 출력함수들로 인해 코드의 돌아가는 시간이 더 길어 졌었다. 따라서, 이번 Single Buffer에서는 모든 출력 함수를 지우고 진행 하였다. 이번 실험은 FreeBSD9-Orig 파일을 기준으로 진행 하였으며, Producer와 Consumer의 개수는 2,2 로 진행 하였다. 그 결과는 다음과 같다.

[Single Buffer]

(2,2)

```
andy@DESKTOP-7CAOFQ6:~$ ./wow FreeBSD9-orig.tar 2 2
파일 처리 시간: 26652273 microseconds ( 26.652273 sec )
```

(5,5)

```
andy@DESKTOP-7CAOFQ6:~$ ./wow FreeBSD9-orig.tar 5 5
파일 처리 시간: 25532390 microseconds ( 25.532390 sec )
```

(20,20)

```
andy@DESKTOP-7CAOFQ6:~$ ./wow FreeBSD9-orig.tar 20 20
파일 처리 시간: 26124406 microseconds ( 26.124406 sec )
```

[Multi Buffer] 2, 2

```
파일 처리 시간: 14487472 microseconds ( 14.487472 sec )
```

Buffer를 하나 늘린 것 만으로, 2배의 성능을 보여주는 것을 확인 할 수 있다. 또한, Single Buffer 에서의 Producer와 Consumer 개수를 늘리는 것은 큰 의미가 그대로 없는 것을 확인 할 수 있었다.

3) Multi Buffer Producer&Consumer 개수 조정(Multi_buffer.c)

Multi Buffer 에서의 Producer와 Consumer 개수의 크기가 커지거나, Producer와 Consumer의 비율을 맞추어 준다면, 결과 값이 개선 될 것이라 가정하고, 다음 실험을 진행 하였다. 이번 실험 역시 FreeBSD9-Orig 파일을 기준으로 진행 하였으며 다음의 표를 통해 확인 할 수 있다.

Producer	Consumer	Throuput
4	4	14.878861
8	8	15.565128
16	16	16.336826
32	32	16.683105
64	64	16.944499
100	100	17.09556
10	20	21.513575
20	40	23.1059

이번 실험 역시, 원하는 결과를 얻지 못하였으며, Producer와 Consumer의 크기가 늘어나도, 큰 영향을 미치지 못하였으며 다른 비율로 해보았을 때 더 안 좋은 성능을 가졌다.

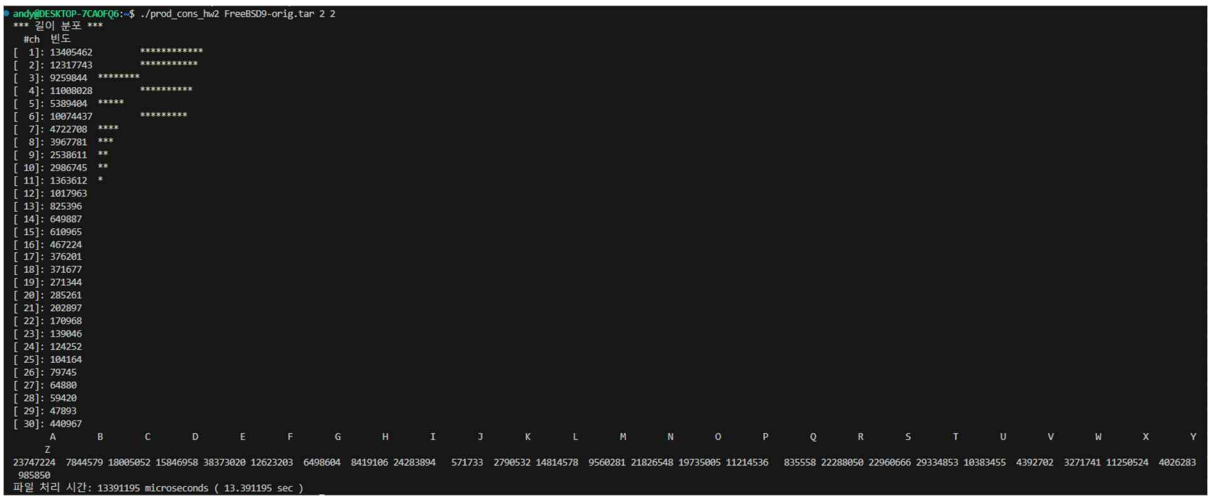
4. Optimized Multi Buffer Producer&Consumer 개수 조정

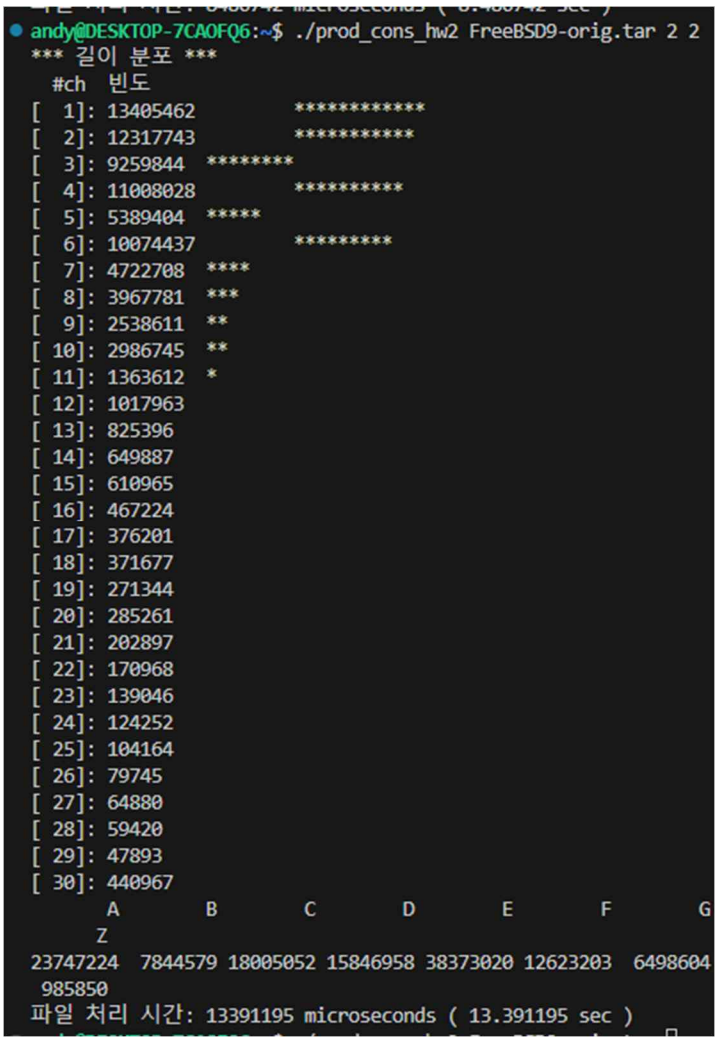
이전 Multi Buffer의 Context Swtiching 과정에서 대기시간이 길어지는 것을 가설로 update stat 함수에 대해서 확인해본 결과, Consumer Thread 내에 Update_stat 함수에서 Consumer의 mutex lock 이외에 Update_stat의 mutex로 있어서 이중으로 Mutex lock으로 인한 대기시간 증가가 3번 실험의 결과를 좋지 않은 결과로 보여주었다. 따라서, 이번 Code는 Update_stat을 Consumer mutex unlock 이후에 사용하기로 결정하였다. 아래는 위 와 같은 최적화 과정을 거친 결과이다.

Producer	Consumer	Throuput
4	4	11.124153
8	8	10.153029
16	16	10.033345
32	32	9.749869
64	64	9.656788
100	100	9.50756
10	20	9.269774
20	40	9.417537
50	100	9.316556
30	90	9.46736
20	80	9.33098
25	100	9.384108
2	2	13.391195

위 결과를 통해 Producer와 Consumer의 개수가 늘어나면, throuput이 전체적으로 좋아지는 것을 확인 할 수 있었다. 또한, Consumer가 더 많을 경우, 해당 Program의 Throuput이 올라가는 것을 확인 할 수 있었다.

(결과 값)


andy@DESKTOP-7CAOFQ6:~$ ./prod_cons_hw2 FreeBSD9-orig.tar 2 2
*** 길이 분포 ***
#ch 빈도
[1]: 13405462 **********
[2]: 12317743 **********
[3]: 9259844 ********
[4]: 11008028 **********
[5]: 5389404 *****
[6]: 10074437 *********
[7]: 4722708 ****
[8]: 3967781 ***
[9]: 2538611 **
[10]: 2986745 **
[11]: 1363612 *
[12]: 1017963
[13]: 825396
[14]: 649887
[15]: 610965
[16]: 467224
[17]: 376201
[18]: 371677
[19]: 271344
[20]: 285261
[21]: 202897
[22]: 170968
[23]: 139046
[24]: 124252
[25]: 104164
[26]: 79745
[27]: 64880
[28]: 59420
[29]: 47893
[30]: 440967
A B C D E F G H I J K L M N O P Q R S T U V W X Y
Z
23747224 7844579 18005052 15846958 38373020 12623203 6498604 8419106 24283894 571733 2790532 14814578 9560281 21826548 19735005 11214536 835558 22288050 22960666 29334853 10383455 4392702 3271741 11250524 4026283
985850
파일 처리 시간: 13391195 microseconds (13.391195 sec)


andy@DESKTOP-7CAOFQ6:~$ ./prod_cons_hw2 FreeBSD9-orig.tar 2 2
*** 길이 분포 ***
#ch 빈도
[1]: 13405462 **********
[2]: 12317743 **********
[3]: 9259844 ********
[4]: 11008028 **********
[5]: 5389404 *****
[6]: 10074437 *********
[7]: 4722708 ****
[8]: 3967781 ***
[9]: 2538611 **
[10]: 2986745 **
[11]: 1363612 *
[12]: 1017963
[13]: 825396
[14]: 649887
[15]: 610965
[16]: 467224
[17]: 376201
[18]: 371677
[19]: 271344
[20]: 285261
[21]: 202897
[22]: 170968
[23]: 139046
[24]: 124252
[25]: 104164
[26]: 79745
[27]: 64880
[28]: 59420
[29]: 47893
[30]: 440967
A B C D E F G
Z
23747224 7844579 18005052 15846958 38373020 12623203 6498604
985850
파일 처리 시간: 13391195 microseconds (13.391195 sec)

결론

이번 과제를 통해 Multi-Thread 프로그래밍 핵심 개념과 동기화 문제를 깊이 이해 할 수 있었다. Producer-Consumer 패턴을 기반으로 구현한 Program의 Thread간 자원 공유와 동기화 문제를 해결하는 과정에서 해당 개념에 대한 이해를 심화 시킬 수 있었다. 여러 실험을 통해 하나의 버퍼가 아닌 Multi Buffer 구조로 확장하면 성능을 개선할 수 있다는 사실을 최적화를 통해 확인 할 수 있었다. 물론, 초기 실험 결과는 성능과 동시성의 이점을 살리지 못했지만, Buffer 개수를 늘리고, Producer와 Thread 수를 조정한 최적화 실험을 통해 동시성 관리와 Buffer 관리가 Program의 중요한 영향을 미칠 수 있다는 사실을 알 수 있었다. 이번 과제에서 아쉬운점은 Pipeline 병렬화를 통해 Buffer를 여러 개로 구현한다면 더 좋은 성능이 나올 수 있을거라고 생각한다.

Ubuntu-20.04 에서 Compile 됨.

MakeFile은 없습니다.

gcc -o (코드이름) (코드이름).c -lpthread

./prod_cons_hw2 FreeBSD9-orig.tar (producer 개수) (consumer 개수)

를 쳐주시면 되겠습니다.

C file 이름은 single_buffer.c, multi_buffer.c 입니다.

Ai에 도움 받은 부분 : Tar file 읽기, Thread 관련 함수 찾아보기.