Operating System (MS)

assignment #2: Multi-threaded Word Count

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 0
Freeday left 5

2024.10.19

1. 프로젝트 소개

1.1 프로젝트 개요

본 프로젝트는 다중 스레드를 활용하여 텍스트 파일에서 알파벳 문자의 빈도를 분석하는 프로그램을 작성하는 과제입니다. 주어진 샘플 코드를 통해 프로듀서와 컨슈머의 일대일 구조를 구현하고, 다대다 구조로 구현할 수 있도록 개선하여, 다중 컨슈머 지원을 하는 프로그램을 만들고, 시스템 분석 프로그램을 이용하여 스레드의 프로듀서와 컨슈머의 비율별로 실행 속도를 측정하고

1.2 프로젝트 목표

멀티스레드 환경에서 프로듀서-컨슈머 모델을 구현하고, 스레드 간 동기화를 통해 자원을 효율적으로 관리하는 것입니다. 여러 컨슈머를 지원하는 구조에서 발생하는 동시성 문제를 해결하고, htop 도구를 이용해 성능을 분석하여 최적의 프로듀서-컨슈머 비율을 찾아내어 시스템 자원의 활용과 성능의 최적화를 이해하는 것이 궁극적인 목표입니다.

2. 프로젝트 요구사항

1) **Sample Code 수정** : 프로듀서와 컨슈머가 1:1로 동작하도록 구현.
2) **다중 컨슈머 지원** : 여러 컨슈머가 동시에 동작하도록 지원
3) **프로그램의 실행** : 실행 시 파일 이름, 프로듀서 수, 컨슈머 수를 인자로 작동
4) **알파벳 통계 수집** : 텍스트에서 알파벳 문자의 빈도를 계산하고 통계를 출력
5) **스레드 실행시간 측정** : htop 프로그램으로 실행 시간을 측정
6) **동시성 극대화** : 최적의 성능을 내는 프로듀서와 컨슈머의 비율 분석

3. 프로젝트에 사용된 개념

3.1 제공된 Sample Code 분석

3.1.1 char_stat.c

: 텍스트 파일을 읽어 각 문자열의 길이와 알파벳 문자의 빈도를 계산하는 프로그램입니다.

주요기능

- 파일을 읽어 각 라인을 분리, 알파벳 문자와 문자열의 길이를 통계로 저장
- stat[] : 각 문자열의 길이를 기록하는 배열
- stat2[] : 알파벳 문자의 빈도를 기록하는 배열
- 프로그램을 종료할 경우, 길이별 문자열의 빈도와 알파벳의 빈도 출력

과제와의 관계 : 과제에서 요구하는 파일 내 문자의 통계를 수집하는 역할로, 위 코드에서 수집한 로직을 prod_cons.c의 consumer thread에 응용하여, 컨슈머가 동시에 문자의 통계를 수집할 수 있게 합니다.

3.1.2 prod_cons.c

: 하나의 프로듀서와 여러 consumer thread를 생성하여 파일에서 라인을 읽고 출력하는 프로그램입니다.

주요기능

- Producer : 파일에서 텍스트 라인을 읽어 공유 객체(sharedobject)에 저장하고, 라인을 읽을 때 'so->line'에 라인을 복사하고 저장, 읽을수 없을시 종료
- Consumer : producer가 읽어둔 라인을 가져와 출력. 라인의 문자열 길이를 계산하고 출력한 후, 해당 메모리를 해제. 더 이상 라인이 없을 경우 종료

과제와의 관계 : Producer - Consumer 패턴을 사용한 멀티스레드 구조의 기본 틀을 제공합니다. 과제에서 요구하는 핵심 요소인 동기화를 위해 pthread_mutex_lock()과 같은 동기화 메커니즘이 팔요합니다.

3.1.3 pthread.c

: 각각의 스래드를 생성하여 주어진 문자열을 대문자로 변환하는 프로그램입니다.

주요기능

- 스레드를 생성하고, 각 스레드에 argv[]로 전달된 문자열을 전달하여 대문자로 변환
- 메인 thread는 각 스레드가 변환한 값을 받아 출력하고, 스레드가 할당한 메모리를 해제
- pthread_attr_t를 사용하여 thread의 스택 크기 등 속성 설정 기능

과제와의 관계 : thread를 생성하고, 스레드의 속성 설정, 스레드의 종료 및 메모리 해제와 같은 스레드의 기본 동작을 보여줍니다.

3.2 개념 정리

3.2.1 **데드락 (DeadLock)**

두 개 이상의 스레드가 개별적인 자원을 점유하면서, 서로의 자원을 기다리면 무한 대기 상태에 빠지는 현상입니다.

해결방법

- 자원 할당 순서 규칙 : 자원을 점유할 때 항상 같은 순서로 자원을 획득하고, 해제하는 규칙을 지정하여 데드락을 방지할 수 있습니다.
- 타임아웃 설정 : 자원을 획득할 때 일정 시간 이상 대기하면 TimeOut을 발생시킵니다. TimeOut은 자원 획득을 포기하고 다른 작업을 수행합니다.
- 데드락 회피 알고리즘 : 운영체제에서 데드락의 발생을 회피하기 위해, 자원의 상태를 실시간으로 모니터링하고 스레드의 요청에 따라 자원의 할당 여부를 결정하는 알고리즘을 실행할 수 있습니다.

3.2.2 **스레드 안전 (Thread Safety)**

여러 스레드가 동시에 동일한 데이터를 접근하거나 조작할 때, 데이터의 일관성을 유지하고 오류없이 작동할 수 있도록 합니다. 하지만 여러 스레드가 동시에 접근하는 과정에서 데이터의 무결성이 손상될 수 있습니다.

해결방법

- 뮤텍스(Mutex) : (상호 배제(mutual exclusion)의 약자)

스레드 간의 자원에 대해 상호배제를 보장하기 위해 뮤텍스를 사용하여 스레드 안전을 구현할 수 있습니다. 자원에 접근할 때 뮤텍스를 잠그고, 작업이 끝난 후 뮤텍스를 해제하여 다른 스레드가 접근할 수 있도록 합니다.

- 스레드 안전 함수 사용 : 표준 라이브러리에 있는 함수 중 strtok_r()는 strtok()와 달리 안전하게 문자열을 분리할 수 있습니다.

3.2.3 **경쟁 상태 (Race Condition)**

두 개 이상의 스레드가 동시에 공유 자원에 접근할 때, 접근 순서에 따라 프로그램의 결과가 달라지는 상황을 뜻하며, 자원의 값을 변경하는 코드에서 경쟁 상태가 발생하면 이상값이 나타날 수 있습니다.

해결방법

- Mutex ; 자원에 대한 접근을 직렬화 합니다. 뮤텍스를 사용하면 스레드가 자원을 점유하는 동안 다른 스레드는 대기하게 되어 경쟁 상태를 방지합니다.
- 임계구역(Critical Section) 최소화 : (공유 자원이 될 수 있는 코드)

뮤텍스 사용 시, 잠금이 걸릴 수 있는 코드를 최소화 하여 스레드가 경쟁 상태가 발생할 수 있는 코드에 접근하는 시간을 줄이는 방법입니다.

3.2.4 Context Switching

운영체제가 실행 중인 스레드의 상태(레지스터, 스택, PC)를 저장하고, 다른 스레드의 상태를 복원하는 과정입니다. 이 과정으로 여러 스레드가 CPU에서 실행될 수 있습니다. 다만, 스레드가 실행 중인 작업을 멈추고 다른 스레드로 전환하는 작업이 빈번해지면 병목현상이

발생하여 CPU의 효율성이 떨어질 수 있습니다.

해결방법 (최적화)

- Thread의 수 제한 : 불필요하게 많은 스레드를 생성하지 않도록 조정합니다.
- Thread Pool 사용 : (스레드를 재활용하는 방법) 스레드의 생성과 제거 빈도를 줄여 컨텍스트 스위칭의 빈도를 줄일 수 있습니다.

3.2.5 **스핀락 (Spinlock)**

자원을 사용할 수 있을 때 까지 Pthr해서 반복적으로 락을 시도하는 방법으로, 짧은 시간동안 자원이 사용중일 때 사용하는 잠금방식입니다. 스핀락을 사용하는 동안, 스레드는 자원을 확보할 때 까지 CPU를 계속 사용하며 대기합니다. 만약 자원의 점유 시간이 길어지게 되면 스레드가 CPU를 계속 사용하며 대기하게 됨으로, 효율이 떨어질 수 있습니다.

해결방법 (최적화)

- 스핀락 대기 시간 조정 ; 스핀락의 대기시간을 짧게 설정하여 자원 점유시간을 최소화하고, 자원이 빠르게 해제되 경우에만 사용하도록 최적화합니다.

3.2.6 **비동기 I/O (Asynchronous I/O)**

입출력 작업을 수행할 때, 스레드가 작업완료를 기다리지 않고, 다른 작업을 바로 수행할 수 있도록 하는 방식입니다. 이는 특히 사용자 인터페이스가 있는 프로그램에서 응답성을 향상시키는데 유리합니다.

3.3 프로그램 구현에 필요한 개념

3.3.1 **프로듀서-컨슈머 모델 구현**

- **프로듀서** 역할: 파일에서 읽은 텍스트 라인을 공유 버퍼에 저장하고, 버퍼가 가득 차면 대기합니다. 데이터가 버퍼에 들어가면 컨슈머에게 신호를 보내 처리하게 하며, 동시 접근을 방지하기 위해 뮤텍스를 사용합니다.
- **컨슈머** 역할: 컨슈머는 공유 버퍼에서 데이터를 꺼내와 처리하는 역할을 담당합니다. 버퍼에서 데이터를 꺼내 알파벳 문자의 빈도를 계산합니다. 프로듀서가 데이터를 넣기 전까지 대기하고, 데이터를 처리한 후 프로듀서에게 신호를 보내 새 데이터를 생산할 수 있게 합니다.
- **공유 버퍼** 관리: 프로듀서와 컨슈머가 데이터를 주고받는 과정에서 데이터 충돌을 방지하기 위해 뮤텍스와 조건 변수를 사용해 동기화를 유지합니다.

3.3.2 **스레드 동기화**

- pthread_mutex_lock()/pthread_mutex_unlock(): 공유 자원에 대한 동시 접근을 방지하기 위해 뮤텍스를 사용합니다. 프로듀서와 컨슈머는 각자 데이터를 넣고 꺼낼 때 뮤텍스로 자원을 보호합니다.
- ppthread_cond_wait()/pthread_cond_signal(): 프로듀서와 컨슈머 간에 작업이 끝났음을 알리기 위해 조건 변수를 사용합니다. 버퍼 상태에 따라 스레드 간 신호를 주고받으며 대기 상태를 관리합니다.

3.3.3 **다중 컨슈머 지원**

여러 개의 컨슈머 스레드를 생성하여 병렬로 데이터를 처리할 수 있도록 구현합니다. 각각의 컨슈머는 독립적으로 데이터를 처리하나, 공유 자원(버퍼)에 대해서는 동기화를 통해 데이터 일관성을 유지해야 합니다.

다중 컨슈머는 프로그램 성능 향상을 기대할 수 있지만, 너무 많은 스레드를 사용할 경우 스레드 간 자원 경쟁으로 CPU의 성능 저하가 발생할 수 있으므로 적절한 스레드 수를 설정하는 것이 중요합니다.

3.3.4 **알파벳 통계 수집**

컨슈머는 버퍼에서 가져온 각 텍스트 라인을 분석하여 알파벳 문자의 빈도를 계산합니다. 여러 컨슈머가 독립적으로, 대소문자는 구분하지 않고, 알파벳의 총 빈도를 기록하며, 프로그램 종료 시 전체 텍스트에 대한 알파벳 빈도를 통합하여 출력합니다. 여러 스레드가 동시에 통계를 수집하므로, 최종 결과가 일관되도록 각 컨슈머의 통계를 적절히 합산해야 합니다.

3.3.5 **성능 측정 및 최적화**

다양한 스레드 구성을 사용하여 프로그램을 실행한 후, 각 스레드 구성에서의 실행 시간을 측정하고 성능을 비교합니다. 성능 최적화는 병렬처리로 효율을 극대화하는 것이 목표이며, 스레드 간 동기화를 최소화하고, 병렬 처리를 최대한 활용하는 것이 중요합니다.

htop(리눅스의 실시간 시스템 사용량 모니터링 도구)과 같은 시스템 모니터링 도구를 사용하여 실시간으로 스레드의 상태를 확인하고 프로그램이 병렬로 효율적으로 실행되고 있는지 확인할 수 있습니다.

4. 프로그램 구현

* **4.1 prod_cons_1_1.c** : 샘플 코드 수정
* **4.2 prod_cons_.c** : 최종 코드 수정
* **4.3 prod_cons_bin.c** : binary파일 읽기 기능 추가
* **4.4 개념의 코드 내 적용 방법**

4.1 Sample Code 수정 (prod_cons.c 기반)

4.1.1 prod_cons_1_1.c (일대일 구조)

1) 공유구조체 변경 `pthread_cond_t cond;`

pthread_cond_t cond 추가 : 프로듀서가 데이터를 생산한 후 컨슈머에게 신호를 보내거나, 컨슈머가 데이터를 소비한 후 프로듀서에게 신호를 보내는데 사용됩니다. 버퍼가 비어있을 때만 데이터를 생산하고, 컨슈머는 버퍼가 가득 찼을 때만

데이터를 소비할 수 있도록 합니다.

2) 프로듀서(Producer) 함수 변경

2.1) 뮤텍스와 조건 변수 사용 `pthread_mutex_lock(&so->lock);` `pthread_mutex_unlock(&so->lock);`

공유 자원인 so->line, so->full에 대한 접근을 보호하여 데이터의 경합(race condition)을 방지합니다.

```
while (so->full)
{
    // Buffer is full, wait for consumer to consume
    pthread_cond_wait(&so->cond, &so->lock);
}
```

버퍼가 가득 찼을 경우(프로듀서가 데이터를 소비하지 않음) 컨슈머가 데이터를 소비할 때까지 기다립니다. pthread_cond_wait는 lock을 해제하고 조건 변수에서 신호를 기다린 후 다시 lock을 획득합니다.

`pthread_cond_signal(&so->cond);`

데이터를 생산한 후 컨슈머에게 새로운 데이터가 준비되었을을 알리거나, 파일의 끝을 알립니다.

2.2) 파일 끝 처리

```
so->line = NULL;  // Signal end of file
pthread_cond_signal(&so->cond);
pthread_mutex_unlock(&so->lock);
```

파일의 끝을 알리기 위해 so->line을 NULL로 설정하여, 컨슈머가 파일의 끝을 인식하고, 루프를 종료하는데 사용합니다. 파일의 끝 이후에 pthread_cond_signal을 호출하여 컨슈머가 대기 중이라면 즉시 깨워서 종료시킵니다.

2.3) 데이터 생산과 동기화 `so->full = 1;`

새로운 데이터가 준비되었음을 표시하여 컨슈머가 데이터를 소비할 수 있도록 합니다.

3) 컨슈머(Consumer) 함수의 변경

`pthread_mutex_lock(&so->lock);` 3.1) 뮤텍스 조건 변수 사용

공유 자원에 대한 접근을 보호하여 데이터

`pthread_mutex_unlock(&so->lock);` 경합을 방지합니다.

```
pthread_mutex_lock(&so->lock);
while (!so->full)
{
    // Buffer is empty, wait for producer to produce
    pthread_cond_wait(&so->cond, &so->lock);
}
```

버퍼가 비어있을 경우(프로듀서가 데이터 생산 전) 데이터가 생산될 때 까지 기다립니다.

3.2) 파일 끝 처리

```
if (so->line == NULL)
{
    // End of file signal
    pthread_mutex_unlock(&so->lock);
    break;
}
```

프로듀서가 파일의 끝을 알리기 위해 so->line을 NULL로 설정했을 때를 인식하여 루프를 종료합니다.

3.3) 데이터 소비와 동기화

```
so->full = 0;
pthread_cond_signal(&so->cond);
pthread_mutex_unlock(&so->lock);
```

데이터를 소비한 후 프로듀서에게 새로운 데이터를 생산할 수 있음을 알립니다.

4) 메인(main) 함수의 변경

4.1) 단일 프로듀서 및 컨슈머로 변경

```
int main(int argc, char *argv[])
{
    pthread_t prod;
    pthread_t cons;
```

원래 코드의 다수의 프로듀서와 컨슈머 스레드를 지원하기 위한 배열을 단일 프로듀서와 단일 컨슈머 스레드를 지원하기 위해 수정하였습니다.

4,2) 쓰레드 생성 및 조인 방식 변경

```
pthread_create(&prod, NULL, producer, share);
pthread_create(&cons, NULL, consumer, share);

pthread_join(cons, (void **)&ret);
printf("main: consumer joined with %d\n", *ret);
free(ret);

pthread_join(prod, (void **)&ret);
printf("main: producer joined with %d\n", *ret);
free(ret);
```

단일 프로듀서와 스레드를 생성하고 각각을 조인합니다.

4.3) 조건 변수 초기화 및 자원 해제

```
pthread_cond_init(&share->cond, NULL);
```

조건 변수를 초기화하여 동기화에 사용할 수 있도록 하였습니다.

```
pthread_mutex_destroy(&share->lock);
pthread_cond_destroy(&share->cond);
```
프로그램 종료 시 조건 변수를 파괴하여 자원을 해제합니다.
프로그램 종료 시 뮤텍스를 파괴하여 자원을 해제합니다.

4.4) 메모리 및 파일 자원 관리
```
fclose(rfile);
free(share);
```
pthread_join으로 반환된 동적 할당된 메모리를 해제하여 메모리 누수를 방지합니다.

5) 전체적인 코드의 동작흐름

5.1) 초기화
- 메인 함수에서 공유 구조체(so_t)를 초기화하고, 파일을 엽니다.
- 뮤텍스와 조건 변수를 초기화합니다.

5.2) 쓰레드 생성
_ 프로듀서 스레드와 컨슈머 스레드를 각각 생성합니다.

5.3) 프로듀서 동작
- 파일에서 한 줄 씩 읽어옵니다.
- 버퍼가 가득 찼는지 확인하고, 가득 찼다면 컨디션 변수를 사용하여 소비자가 데이터를 소비할 때까지 기다립니다.
- 데이터를 생산하고 조건 변수를 통해 소비자에게 신호를 보냅니다.
- 파일의 끝에 도달하면 so->line을 NULL로 설정하고 컨슈머에게 종료신호를 보냅니다.

5.4) 컨슈머 동작
- 버퍼가 비어있는지 확인하고, 비어있다면 프로듀서가 데이터를 생산할 때까지 대기합니다.
- 데이터를 소비하고, 버퍼를 비운 후 프로듀서에게 신호를 보냅니다.

5.5) 종료 및 정리
- 메인 함수에서 두 쓰레드가 종료될 때까지 pthread_join을 호출하여 대기합니다.
- 모든 자원(mutex, 조건 변수, 파일 메모리)를 정리하고 프로그램을 종료힙니다.

4.2 프로그램 구현 (prod_cons_.c)

4.1.1 **전역 상수와 데이터 구조 정의**
```
#define ASCII_SIZE 256
#define MAX_STRING_LENGTH 30
```
- ASCII 문자를 사용할 수 있도록 256칸의 배열 크기를 정의
- 문자열의 최대 길이를 30으로 정의하여, 각 줄의 길이에 대한 통계를 수집할 때 이 범위를 넘지 않도록 설정

프로듀서가 읽어오는 파일 포인터
현재 읽고 있는 줄의 데이터 저장(버퍼)
현재 라인의 번호 저장

```
typedef struct sharedobject
{
    FILE *rfile;
    char *line;
    int linenum;
    int full;
    int done;
    int stat[MAX_STRING_LENGTH];
    int stat2[ASCII_SIZE];
    pthread_mutex_t lock;
    pthread_cond_t cond;
} so_t;
```

공유 버퍼가 채워졌는지를 나타내는 flag
파일을 모두 읽었는지 나타내는 flag
문자열 길이에 대한 통계를 저장하는 배열
알파벳 빈도에 대한 통계를 저장하는 배열
프로듀서와 컨슈머 간의 동기화를 위한 뮤텍스
프로듀서와 컨슈머 간의 통신을 위한 조건 변수

4.1.2 Producer 함수

- 뮤텍스 잠금 및 조건 변수 대기 : 프로듀서가 공유 데이터 에 접근하기 위해 so->lock

```
while (1)
{
    pthread_mutex_lock(&so->lock);
    while (so->full)
    {
        pthread_cond_wait(&so->cond, &so->lock);
    }
```

뮤텍스를 잠그고, so->full이 1인 경우(공유 버퍼의 데이터가 소비되지 않은 경우)에는 pthread_cond_wait을 통해 조건 변수를 사용하여 기다립니다. 이때 뮤텍스는 자동으로 해제되고, 다시 잠금이 해제된 후 소비자가 데이터를 처리할 때까지 대기합니다.

```
if (so->line)
{
    free(so->line);
    so->line = NULL;
}
```

- 새로운 줄을 읽기 전에 이전 줄에 할당된 메모리를 해제합니다. line은 동적으로 할당된 메모리로, 누수를 방지하기 위해 이를 명시적으로 해제합니다.

```
read = getline(&line, &len, so->rfile);
if (read == -1)
{
    so->done = 1;
    pthread_cond_broadcast(&so->cond);
    pthread_mutex_unlock(&so->lock);
    break;
}
so->line = strdup(line);
so->linenum = i++;
so->full = 1;
pthread_cond_broadcast(&so->cond);
pthread_mutex_unlock(&so->lock);
```

- 파일에서 한 줄을 읽고, 파일의 끝에 도달하면 so->done을 1로 설정하고, 모든 대기 중인 컨슈머에게 종료 신호를 보냅니다. pthread_cond_broadcast는 여러 컨슈머에게 동시에 신호를 보내기 위한 함수입니다.

- 파일에서 읽은 줄을 strdup을 사용해 복사한 후, 공유 버퍼인 so->line에 저장힙니다. so->full을 1로 설정해 버퍼가 가득 찬 것을 표시하고, pthread_cond_broadcast를 통해 컨슈머에게 데이터가 준비되었음을 알리고, 컨슈머가 데이터에 접근할 수 있도록 뮤텍스를 해제합니다.

4.1.3 Consumer 함수

```
pthread_mutex_lock(&so->lock);
while (!so->full && !so->done)
{
    pthread_cond_wait(&so->cond, &so->lock);
}
```

- 공유 데이터에 접근하기 위해 so->lock 뮤텍스를 잠그고, so->full이 1(파일을 다 읽지 않음)이 아니면, 조건 변수를 통해 프로듀서가 데이터를 생산할 때까지 기다립니다.
- 파일을 모두 읽고, 버퍼가 비어있으면 더 이상 처리할 데이터가 없다는 의미로, mutex를

```
if (so->done && !so->full)
{
    pthread_mutex_unlock(&so->lock);
    break;
}
```

해제하고 루프를 종료합니다.

```
char *line = so->line;
printf("Consumer %lx: [line %d] %s", pthread_self(), so->linenum, line);
```

- so->line에서 데이터를 읽고, 해당 줄의 번호(so->linenum)와 함께 출력합니다. pthread_self()는 현재 스레드의 ID를 16진수로 변환하는 함수로, 각각 컨슈머 스레드가 어느 라인을 처리했는지 표시할 수 있습니다.

```
// 문자열 길이 통계 수집
size_t len = strlen(line);
if (len >= MAX_STRING_LENGTH) len = MAX_STRING_LENGTH;
so->stat[len - 1]++;
```

- 문자열 길이 통계 수집을 담당하는 부분으로, line의 길이를 구하고, 그 길이에 해당하는 통계를 so->stat 배열에 기록합니다. 길이가 30(MAX_STRING_LENGTH)을 넘으면 최대 길이로 통일합니다.

```
// 알파벳 통계 수집
for (int i = 0; line[i] != '\0'; i++)
{
    if ((line[i] >= 'A' && line[i] <= 'Z') || (line[i] >= 'a' && line[i] <= 'z'))
    {
        so->stat2[(unsigned char)line[i]]++;
    }
}
```

- 문자열에서 각각의 문자를 확인하고, 대문자 또는 소문자인 경우 해당 문자의 빈도를 so->stat2 배열에 기록합니다.

```
so->full = 0;
pthread_cond_signal(&so->cond);
pthread_mutex_unlock(&so->lock);
(*ret)++;
```

- 데이터를 처리한 후, so->full을 0으로 설정해 버퍼가 비어 있음을 표시합니다. 이후에는 조건 변수를 통해 대기 중인 프로듀서를 깨워서 새로운 데이터를ㄹ 생산할 수 있도록 준비합니다.

4.1.4 **Main 함수**

```c
if (argc < 4) {  // 파일 이름, 프로듀서 수, 소비자 수를 인자로 받음
    printf("Usage: %s <filename> <num_producers> <num_consumers>\n", argv[0]);
    exit(EXIT_FAILURE);
}

int num_producers = atoi(argv[2]);
int num_consumers = atoi(argv[3]);
```

- 명령줄 인자 처리를 위해 프로듀서 수(argv[2])와 컨슈머 수(argc[3])를 인자로 받습니다.

```c
if (num_producers <= 0 || num_consumers <= 0)
{
    printf("Error: The number of producers and consumers must be greater than 0\n");
    exit(EXIT_FAILURE);
}
```

- 만약 입력된 프로듀서의 수와 컨슈머 수가 0보다 작을 경우 오류를 출력하고 프로그램을 종료합니다.

```c
// 파일 열기
shared.rfile = fopen(argv[1], "r");
if (!shared.rfile)
{
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

- 프로듀서가 읽어올 파일을 열고, 이를 shared.rfile에 저장하고, 파일을 열지 못하면 에러 출력을 하고 프로그램을 종료합니다.

```c
// 프로듀서 스레드 생성
pthread_t producers[num_producers];
for (int i = 0; i < num_producers; i++)
{
    pthread_create(&producers[i], NULL, producer, &shared);
}

// 컨슈머 스레드 생성
pthread_t consumers[num_consumers];
for (int i = 0; i < num_consumers; i++)
{
    pthread_create(&consumers[i], NULL, consumer, &shared);
}
```

- 프로듀서와 컨슈머 스레드를 각각 num_producers와 num_consumers의 개수 만큼 생성하고, 각각의 스레드는 공유데이터(&shared)를 인자로 받아 실행합니다.

```c
// 모든 프로듀서 스레드 종료 대기
for (int i = 0; i < num_producers; i++)
{
    pthread_join(producers[i], NULL);
}

// 모든 컨슈머 스레드 종료 대기
int total_processed = 0;
for (int i = 0; i < num_consumers; i++)
{
    int *lines_processed;
    pthread_join(consumers[i], (void **)&lines_processed);
    total_processed += *lines_processed;
    free(lines_processed);
}
```

- 모든 프로듀서 스레드가 종료될 때까지 기다리며, 각 컨슈머가 처리한 줄의 수를 합산하여 총 처리된 줄의 수를 계산합니다.

```
// 문자열 길이 통계 출력
printf("*** String length distribution ***\n");
for (int i = 0; i < MAX_STRING_LENGTH; i++)
{
    if (shared.stat[i] > 0) {
        printf("[%2d]: %4d\n", i + 1, shared.stat[i]);
    }
}
```

- 프로그램이 처리한 줄의 길이에 대한 통계를 출력하고, 길이가 1부터 30(MAX_STRING_LENGTH)까지의 문자열 개수를 보여줍니다.

```
// 알파벳 통계 출력
printf("\nAlphabet statistics:\n");
for (char c = 'A'; c <= 'Z'; c++)
{
    printf("%c: %d\n", c, shared.stat2[(unsigned char)c] + shared.stat2[(unsigned char)(c + 32)]);
}
```

- 대소문자를 구분하지 않고 각 알파벳이 얼마나 많이 사용됐는지 알파벳 통계를 출력합니다.

```
// 리소스 해제
fclose(shared.rfile);
pthread_mutex_destroy(&shared.lock);
pthread_cond_destroy(&shared.cond);
```

- 프로그램이 종료되기 전에 열린 파일과 사용한 뮤텍스, 조건 변수를 해제하여 자원을 반환합니다.

4.1.5 **실행 시간 측정 구현**

```
#define _POSIX_C_SOURCE 199309L
```

- clock_gettime() 같은 함수들이 정의되지 않은 경우나, CLOCK_MONOTONIC을 사용하기 위한 POSIX 기능을 활성화 합니다.

```
// 구조체 설정 및 시작 시간 측정
struct timespec start_time, end_time;
clock_gettime(CLOCK_MONOTONIC, &start_time);
```

- 각각 시간을 초와 나노초로 저장하는 구조체(timespec), 시작 시점(start_time)과 끝난 시점(end_time)을 각각 저장하는 변수를 정의합니다.
- 시스템의 특정 시간 값을 얻는 함수로, CLOCK_MONOTONIC은 단조시간을 의미하고, 부팅 이후의 경과 시간을 나타내며, 시스템 시간의 변경에 영향을 받지 않습니다. 시간 값을 저장할 timespec 포인터에 시작하는 시간(&start_time)을 저장합니다.

```
// 끝난 시간 측정 및 계산
clock_gettime(CLOCK_MONOTONIC, &end_time);
double elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
                      (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
printf("Elapsed time: %.6f seconds\n", elapsed_time);
```

- 프로그램의 모든 작업이 완료된 후, 동일하게 clock_gettime()함수를 사용하여 종료 시점 시간(end_time)에 저장합니다.
- 각각의 초단위 나노초 단위의 시간 차이를 계산하여 총 경과 시간을 계산합니다.
- 측정한 프로그램 실행 경과 시간을 소수점 6자리 까지 출력합니다.

4.3 prod_cons_bin.c

바이너리 파일을 읽기 위한 최종코드 수정

4.3.1 파일 읽기 단위 조정

```
#define BUFFER_SIZE 1024
```

바이너리 파일은 바이트 단위로 데이터를 처리하기에, BUFFER_SIZE는 한번에 읽어들일 데이터의 크기를 설정합니다. 이 프로그램의 경우 1024로 설정되어 1KB씩 데이터를 읽어들입니다.

4.3.2 binary 데이터 저장을 위한 버퍼 생성

```
unsigned char buffer[BUFFER_SIZE];
```

바이너리 데이터를 임시로 저장할 buffer배열입니다. 한번에 읽어오는 데이터량은 1024(BUFFER_SIZE)입니다. bytes_read는 실제 읽어들인 바이트 수를 저장하는 변수로 파일에서 읽은 데이터가 몇 바이트인지 확인하는데 사용됩니다.

4.3.3 파일 열기 모드 수정

```
// 바이너리 파일 열기
shared.rfile = fopen(argv[1], "rb");
if (!shared.rfile)
{
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

바이너리 파일을 읽기 위해 “r”에서 “rb” 모드로 수정합니다.

4.3.4 fread로 binary 데이터 읽기

```
// 바이너리 파일에서 데이터를 읽음
so->bytes_read = fread(so->buffer, 1, BUFFER_SIZE, so->rfile);
```

이전의 getline 대신 fread를 사용하여 바이너리 데이터를 읽어들입니다. 매개변수 중 so->buffer는 데이터를 저장할 버퍼, 1은 한번에 읽어들일 단위 크기(1byte), BUFFER_SIZE는 한번에 읽을 바이트 수, so->rfile은 읽을 파일의 포인터입니다.

4.4 개념의 코드 내 적용 방법

4.4.1 Deadlock

데드락을 방지하기 위해 자원을 적절한 순서로 할당하고 해제합니다.

* pthread_mutex_lock(&so->lock); 스레드가 공유자원에 접근할 때 자원을 획득
* pthread_mutex_unlock(&so->lock); 작업이 끝난 경우 자원을 해제.
* pthread_cond_wait(&so->cond, &so->lock);
* pthread_cond_signal(&so->cond);

대기 중인 자원을 기다릴 때 효율적으로 대기할 수 있게 하고, 불필요한 무한 대기를 방지하여 데드락을 방지합니다.

4.4.2 Thread Safety & Race Condition

여러 스레드가 동일한 데이터를 접근하거나 수정할 때 데이터의 일관성 유지와 무결성 보장을 위해 필요합니다.

* pthread_mutex_lock(&so->lock);
* pthread_mutex_unlock(&so->lock);

mutex를 조작하여 스레드 간의 공유자원(파일, 배열 등)에 대한 접근을 잠금과 해제를 통해 보호하며, 여러 스레드가 자원에 동시에 접근해 발생할 수 있는 경쟁 상태(Race Condition)을 방지합니다. 잠금이 걸린 코드 블록을 최소하하여 경쟁 상태가 발생할 수 있는 구역을 줄이고, 효율적으로 병렬처리가 이루어질 수 있도록 돕습니다.

4.4.3 Context Switching

운영체제가 실행 중인 스레드 상태를 저장하고 다른 스레드로 전환하는 과정으로, 코드에서 스레드는 동시에 실행되며, 적절한 동기화 후 context switching이 이루어집니다.

* pthread_create(&producers[i], NULL, producer, &shared);
* pthread_create(&consumers[i], NULL, consumer, &shared);
* pthread_join(producers[i], NULL);
* pthread_join(consumers[i], NULL);

pthread_create()으로 스레드를 생성하고, pthread_join()으로 스레드가 종료될 때까지 대기합니다. 이를 통해 생성된 스레드는 병렬로 실행됩니다.

4.4.4 Spinlock

스레드가 자원을 사용할 수 있을 때까지 지속적으로 자원을 요청하는 방법으로, 이 프로그램에서는 스핀락 대신 조건 변수와 뮤택스를 사용하여 구현하였습니다.

* pthread_cond_wait(&so->cond, &so->lock);
* pthread_cond_signal(&so->cond);

pthread_cond_wait()을 통해 자원을 사용할 수 있을 때까지 대기하고, pthread_cond_signal()을 통해 자원의 사용 가능하다는 신호를 보내 스레드가 대기 상태에서 벗어나게합니다.

5. 프로그램 빌드 환경

컴파일 환경: Ubuntu 22.04 -> VS Code 1.93
프로그래밍 언어: C언어
실행 파일: prod_cons_1_1.c / prod_cons_.c / Makefile
열람 파일: Book.txt
실행 방법:

* make : makefile을 이용하여 컴파일
* ./prod_cons_1_1 <실행할 파일>
  : 샘플코드(prod_cons.c)를 수정하여 프로듀서와 컨슈머 1대1로 실행
* ./prod_cons_ <실행할 파일> <프로듀서 수> <컨슈머 수>
  : 프로듀서와 컨슈머의 수를 지정하여 프로그램을 실행

6. 문제점과 해결방법

6.1 다중 컨슈머와 공유 자원 보호

여러 컨슈머 스레드가 공유 자원에 동시에 접근할 때 발생하는 경쟁 상태를 해결하기가 어려웠습니다. 특히 pthread_cond_wait()과 pthread_cond_signal()의 적절한 사용 시점을 결정하는 것이 복잡했습니다. 결국 GPT를 사용하여 코드에서의 적절한 신호전달 시기를 이해한 후에 프로듀서와 컨슈머 간의 정확한 시그널 전달을 구현하였습니다.

6.2 htop 시스템 함수 사용(시간 측정)

능 측정을 위한 시간 측정 구간을 결정하는 과정에서도 고민이 있었습니다. 파일을 읽어들이는 과정도 포함하여 전체 성능을 측정할지, 아니면 스레드 구현과 실행만을 측정할지 결정해야 했습니다. 과제의 목표는 프로듀서-컨슈머 모델의 효율적인 스레드 구성을 분석하는 것이었으므로, 파일 읽기 시간보다는 스레드 간 동작을 집중적으로 분석하기로 했습니다.

6.3 Segmentation fault

프로그램의 작성중 발생한 오류로, GPT를 통해 제시된 오류 해결 방안은 중복 메모리 해제 문제, 경계 조건의 문제, 프로듀서와 컨슈머 간의 동기화 문제 등이 문제의 원인일 수 있다고 설명해주었습니다. 각각을 확인하며 찾은 원인은 경계 조건 문제로 파일이 끝에 도달했을 때 프로그램에 알리지 않아 발생하는 문제로, so->done을 설정하여 파일 끝에서 무한 대기 상태가 되지 않도록 수정했습니다.

```
// 바이너리 파일에서 데이터를 읽음
so->bytes_read = fread(so->buffer, 1, BUFFER_SIZE, so->rfile);
if (so->bytes_read == 0) // 파일 끝에 도달한 경우
{
    so->done = 1;
    pthread_cond_broadcast(&so->cond); // 모든 컨슈머에 알림
    pthread_mutex_unlock(&so->lock);
    break;
}
```

수정된 코드

7. 실행결과 및 분석

7.1 실행 결과창

7.1.1 prod_cons_1_1.c

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$ ./prod_cons_1_1 Book.txt
Consumer b8744640: [00:00] The Project Gutenberg eBook of A Tale of Two Cities
...
Consumer 12d10640: [104313:104313] curiosity, Producer 13511640: 104314 lines
Consumer 12d10640: 104314 lines processed
main: consumer joined with 104314
main: producer joined with 104314
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$
```

7.1.2 prod_cons_.c

1) 프로그램 실행 (프로듀서-컨슈머 : 1-1)

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$ ./prod_cons_ Book.txt 1 1
Consumer 7fcf9c7a1640: [line 0] The Project Gutenberg eBook of A Tale of Two Cities
...
Consumer 7f8fd3fcf640: [line 104313] curiosity, *** String length distribution ***
[ 2]: 24260
[ 3]:    3
[ 4]:  105
...
[30]: 71138

Alphabet statistics:
A: 292887
B: 54647
...
Y: 72248
Z: 2578
Elapsed time: 5.537162 seconds
Total lines processed: 104314
```

2) 프로그램 실행 (프로듀서-컨슈머 : 10-10)

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$ ./prod_cons_ Book.txt 10 10
Consumer 7f7ce0ff9640: [line 0] The Project Gutenberg eBook of A Tale of Two Cities
...
Y: 72248
Z: 2578
Elapsed time: 6.864019 seconds
Total lines processed: 104314
```

7.1.3 prod_cons_bin.c

1) 프로그램 실행 (프로듀서-컨슈머 : 1-1)

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$ ./prod_cons_bin Book.txt 1 1

Alphabet statistics:
A: 292887
...
Z: 2578
Elapsed time: 0.164695 seconds
Total lines processed: 4812
```

2) 프로그램 실행 (프로듀서-컨슈머 : 10-10)

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw2/Code$ ./prod_cons_bin Book.txt 10 10

Alphabet statistics:
A: 292887
...
Z: 2578
Elapsed time: 0.162971 seconds
Total lines processed: 4812
```

* 각 라인별 데이터가 출력되지 않음

7.2 성능 최적화

7.2.1 prod_cons_.c 분석 (최종코드)

(행: Producer / 열 : Consumer)

	1	2	3	4	5	...	10	...	50
1	4.192	3.953	4.024	4.265	4.923		6.756		11.135
2	5.023	4.265	4.253	4.548	5.221				
3	6.681	5.915	6.212	5.756	5.983				
4	9.588	8.365	7.215	6.546	6.788				
5	13.215	9.985	8.542	7.945	7.752				
...									
10	32.452						18.23		
...									
50	111.422								83.232

전반적으로 프로듀서와 컨슈머의 수가 증가함에 따라 프로그램의 실행 효율이 점점 낮아지는 경향이 나타났습니다. 특히, 컨슈머의 수가 증가할수록 실행 시간이 급격하게 늘어나, 프로듀서의 수를 늘린 경우와 비교했을 때 상대적으로 대략 10배 가까이 더 많은 시간이 소요되는 것을 확인할 수 있었습니다.

그러나 프로듀서와 컨슈머의 수가 각각 5개 이하일 때는, 반드시 1:1 구조가 가장 효율적인 것은 아니었습니다. 예를 들어, 2대2 구조가 1대1 구조보다 더 나은 성능을 보여주었으며, 이는 스레드가 적절히 분산되고 코어 자원이 효율적으로 사용된 결과라고 분석할 수 있습니다. 하지만 너무 많은 스레드를 사용할 경우, 자원 경쟁과 컨텍스트 스위칭의 빈도가 증가하여 성능이 저하될 수 있음을 알 수 있습니다.

결론적으로, 스레드 수를 적절히 조정하는 것이 프로그램의 성능에 큰 영향을 미치며, 단순히 스레드 수를 늘리는 것만으로는 성능이 개선되지 않는다는 점을 확인할 수 있었습니다.

* 프로듀서의 수는 2개 컨슈머의 수가 1일 때 가장 효율적이었습니다.

7.2.2 prod_cons_bin.c 분석

파일을 읽어드리는 방법을 줄별로 하느냐, byte단위의 묶음으로 하는지는 실행시간이 각각 0.155초 5.537초로 매우 큰 차이를 보였습니다.

* prod_cons_bin.c

```
Elapsed time: 0.155276 seconds
Total lines processed: 4812
```

* prod_cons_.c

```
Elapsed time: 5.537162 seconds
Total lines processed: 104314
```

각각의 코드에 사용되어 파일을 읽어들이는 함수는 각각 getline()은 줄 단위로 파일을 읽어들이기 때문에, 줄바꿈 문자('\n')를 기준으로 한 줄씩 파일을 처리하고, getline()은 줄 단위로 파일을 읽어들이기 때문에, 줄바꿈 문자('\n')를 기준으로 한 줄씩 파일을 처리합니다.

실제 프로세스된 라인의 수를 분석하면 각각 4821줄과 104314줄로 약 27배가 차이가 나고, 실행시간은 약 36배의 차이가 있어, 한번에 처리하는 데이터의 양에 따라 멀티스레드의 성능에 영향을 줄 수 있다는 것을 알 수 있습니다.

7.2.3 추가적인 테스트

Test 1. prod_cons_bin.c의 파일에서 한번에 처리하는 데이터 묶음 단위를 바꾼다면?

* 기본값

```
#define BUFFER_SIZE 1024
```

```
Elapsed time: 0.164695 seconds
Total lines processed: 4812
```

* case1. 버퍼의 사이즈를 키운 후 실행

```
#define BUFFER_SIZE 1048576
```

```
Elapsed time: 0.022838 seconds
Total lines processed: 5
```

* case3. 버퍼의 사이즈를 줄인 후 실행

```
#define BUFFER_SIZE 64
```

```
Elapsed time: 2.199750 seconds
Total lines processed: 76992
```

이를 통해 버퍼의 사이즈가 프로그램의 실행시간에 영향을 미친다는 것을 알 수 있습니다.

Test 2. 멀티 스레드의 성능에 컴퓨터의 코어 성능이 영향을 미치는가?

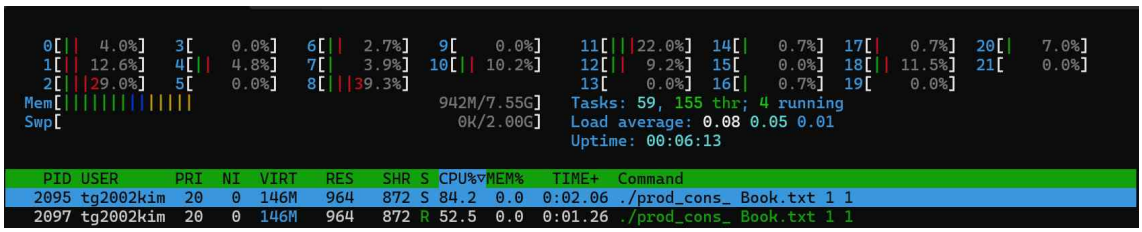
찾아본 바에 따르면 컴퓨터의 코어 사양은 멀티 스레드의 성능에 중요한 영향을 미친다. 코어 수가 부족한 경우, CPU는 여러 스레드를 실행해야 하므로 컨텍스트 스위칭(스레드 간 전환)이 자주 발생하고, 이 과정은 오버헤드를 발생시키며, 너무 많은 스레드가 있을 경우 성능 저하를 유발할 수 있습니다. 반면, 여러 코어가 있는 시스템에서는 이러한 오버헤드가 줄어들지만, 코어 수가 많더라도 스레드 동기화와 병목현상, I/O 대기 시간이 문제를 일으킬 수 있으므로, 스레드 수와 작업 특성에 맞게 조정하는 것이 중요합니다.

* 다른 요인으로 는 클럭의 속도도 영향을 미친다고 합니다.

7.3 htop 스레드 모니터링

7.3.1 htop 이란?

htop은 시스템에서 실행 중인 프로세스와 스레드의 상태를 실시간으로 모니터링할 수 있는 도구입니다. 특히, 멀티 스레드 프로그램에서 프로듀서-컨슈머 모델이 어떻게 동작하는지 확인하고, 각 스레드가 CPU 및 메모리 자원을 얼마나 효과적으로 사용하고 있는지 살펴보는 데 유용합니다.



7.3.2 **스레드 모니터링**

htop에서 H 키를 눌러 스레드 보기 모드를 활성화하면, 각 스레드의 PID가 표시되고 해당 스레드가 얼마나 많은 CPU를 사용하고 있는지 확인할 수 있습니다. 이를 통해 프로듀서와 컨슈머 스레드가 CPU 자원을 어떻게 분배하고 소모하고 있는지를 실시간으로 확인할 수 있습니다. 각 스레드의 상태는 실행 중, 대기 중, 또는 종료 상태로 존재하며, htop을 통해 스레드의 상태를 실시간으로 확인함으로써, 프로듀서-컨슈머 모델이 적절히 동작하고 있는지 점검할 수 있습니다.

다중 스레트 환경에서는 여러 스레드가 동시에 실행될 때 각 스레드들 이 얼마나 효율적으로 CPU자원을 사용하고 있는지 분석하여, 최적의 스레드 수와 병렬 실행 구조를 찾아낼 수 있고, 특정 스레드가 지나치게 많은 자원을 사용하고 있다면 병목 현상이 발생할 가능성이 있습니다. htop을 통해 프로듀서와 컨슈머 간의 작업 분배가 적절히 이루어지고 있는지, 병목 현상을 줄이기 위한 최적화 작업을 진행할 수 있습니다.

7.3.3 **CPU 및 메모리 사용율 확인**

htop을 사용하여 프로듀서 스레드와 컨슈머 스레드 각각이 CPU를 얼마나 사용하고 있는지 실시간으로 모니터링할 수 있습니다. CPU 사용률과 메모리 사용율을 확인하는 것은 프로그램의 성능을 최적화하는 데 중요한 요소로, 특정 스레드가 과도한 자원을 소모하고 있지는 않은지 확인할 수 있습니다.

멀티 스레드 프로그램에서는 공유 메모리 자원을 적절하게 관리하는 것이 중요하므로, 메모리 사용량을 모니터링하여 메모리의 사용효율을 높이는데 도움줄 수 있습니다.

8. 느낀점

가장 어려웠던 부분은 프로듀서와 컨슈머 스레드 간의 적절한 동기화였습니다. 특히, 단순히 pthread_mutex_lock(), pthread_mutex_unlock()과 같은 동기화 기법을 사용하는 것만으로는 충분하지 않았습니다. 초기에는 공유 자원에 대한 관리가 체계적으로 이루어지지 않았기 때문에, 프로그램이 제대로 동작하지 않는 경우가 빈번히 발생했습니다. 특히 다중

컨슈머로 전환되었을 때, 각 스레드가 자원을 보호하고 동기화하는 방식에 대한 이해가 부족했습니다. 프로듀서-컨슈머 패턴에서 pthread_cond_wait()와 pthread_cond_signal()의 사용 목적을 정확히 이해하고, 프로듀서와 컨슈머 간의 데이터 교환 시점을 조절하는 것이 생각보다 복잡했고, 이에 대한 혼란이 있었습니다.

htop을 활용해 스레드 실행 상태를 실시간으로 모니터링하고 분석할 수 있어서 매우 유익했습니다. 특히, 스레드별 CPU 사용량과 자원 분배 상태를 한눈에 파악할 수 있다는 점이 신기했고, 멀티스레드 프로그램의 성능을 보다 자세히 분석해볼 수 있었습니다. 이를 통해 프로그램의 효율성을 높이고 개선할 수 있는 방법을 찾는 데 큰 도움이 되었고, htop을 사용한 모니터링 방식이 실무에서도 매우 유용할 것이라고 느꼈습니다.

구현에 있어 당황했던 부분은 처음에는 샘플코드를 기반으로 텍스트 파일을 대상으로 getline() 함수를 사용해 프로듀서-컨슈머 스레드 구조를 작성했을 때였습니다. 교수님이 제공해주신 freeBSD9-orig 파일을 실행하려고 했지만, VS Code 환경에서 해당 파일을 업로드하는 데 문제가 있었습니다. 결국 텍스트 파일로 대신 실행해보았고, 알파벳 통계나 실행 시간 등이 모두 잘 출력되었기 때문에 이를 최종 코드로 보고서 작성을 시작했습니다. 뒤늦게 바이너리 파일을 읽는 기능이 빠져 있다는 것을 알게되었을 때는 보고서를 거의다 작성한 후여서 binary 파일도 실행할 수 있도록 구현하는 것까지만을 목표로 두었습니다.

과제에 대해 선배들과 논의하면서 얻은 아이디어가 있었습니다. 교수님이 짚어주신 '읽어들이는 byte 수에 따라 효율이 다르다는 점'에 대해 이야기를 나누다가 '파일 전체를 한번에 읽는 것이 더 효율적이다', '결과로 나온 가장 효율적인 구조는 작동시킨 컴퓨터의 코어에 영향을 받는다'는 의견이 나누어졌고, 다양한 의견을 바탕으로 분석을 해볼 수 있어서 좋았습니다. 특히 분석하면서 줄별로 파일을 읽어들이는 것과 byte 단위로 읽어들이는 것에 효율성이 매우 차이가 나는 것이 가장 놀라웠고, line에서 byte로 분석하는 것을 추가로 구현해보길 잘했다는 생각이 들었습니다.

이번 과제에서 GPT의 도움을 많이 받기는 했지만,멀티 스레드 구조와 동기화 방법에 대해 깊이 있게 이해할 수 있었습니다. 여러 번의 시행착오를 거치면서 동기화의 중요성, 자원 보호의 필요성, 그리고 효율적인 프로그램 설계에 대해 큰 배움을 얻었으며, 이를 기반으로 프로그램의 성능을 최적화할 수 있었습니다.

9. 참고자료

* DeadLock & RaceCondition & Mutex
https://pinopino.tistory.com/entry/4-Deadlock-%EC%84%B8%EB%A7%88%ED%8F%AC%EC%96%B4-%EB%AE%A4%ED%85%8D%EC%8A%A4

* Context Swtiching & Mutex
https://woo9282.tistory.com/122

* Thread Satety
https://developer-ellen.tistory.com/205

* Multi Thread
https://studyandwrite.tistory.com/8