

OPERATING SYSTEMS

HW2 :: Multi-threaded word count

By using C programming

모바일시스템공학과 손보경 (32212190)

ssg020910@naver.com

2024.10.19.

목 차

1. Introduction	3
2. Requirement	3
3. Concepts	4
3-1. Thread	4
3-2. Multi-Thread	4
3-3. Synchronization	5
4. Implementation	7
4-1. Makefile	7
4-2. prod_cons.c 파일	7
4-3. multi_cons.c 파일	9
4-4. word_count_cons.c 파일	13
5. Result	14
6. Build Environment	16
7. Lesson	17
7-1. Conclusion	17
7-2. Review	17
7-2. Using ChatGPT	18

1. Introduction

멀티 프로세스를 이용한 병렬 처리는 비용이 많이 든다는 문제가 있다. 프로세스는 Text, Data, Heap, Stack의 공간이 있어서 매우 무겁고, 이를 동시에 수행한다는 것은 오버헤드가 발생할 확률이 높다. 이를 해결하기 위해 나타난 개념이 멀티 스레딩이다.

스레드란 실행 단위를 의미한다. 스레드는 레지스터와 스택을 포함한 execution context를 가지게 된다. 메모리 공간은 같은 프로세스 내 스레드들이 공유하게 된다. 단일 스레드는 한 번에 하나의 작업만 수행할 수 있지만, 현대 운영체제는 멀티 스레딩을 통해 동시에 여러 작업을 수행할 수 있게 지원한다.

멀티 스레딩은 오버헤드가 적어지며 리소스를 공유하기 때문에 IPC가 쉬워진다는 장점 등 여러 장점이 있지만, 동기화, 상호 배제, 교착 상태, 기아 상태, 최적화 등의 문제를 해결해야 한다. 이를 위해 뮤텍스, 세마포어와 같은 동기화 기법과 최적화 방법을 사용한다.

해당 보고서에서는 스레드, 멀티스레드의 개념과 멀티스레드 프로그래밍 과정에서 발생하는 문제 및 해결 방법을 설명할 것이고, 요구 사항에 맞는 3가지의 코드를 보여줄 것이다.

2. Requirement

Category	Requirement
과제 설명	- 멀티스레드로 단어 수 세기
프로듀서-컨슈머 문제 해결	- 1개의 프로듀서와 1개의 컨슈머가 공유 버퍼를 이용하여 데이터를 주고받는 문제를 해결
다중 컨슈머 지원	- 여러 컨슈머가 동시에 공유 버퍼에서 데이터를 소비할 수 있도록 프로그램을 확장
알파벳 문자 통계 수집	- 소비자가 파일의 각 알파벳 문자의 개수를 세고, 프로그램 종료 시 통계 정보를 출력

동기화 처리	- pthread 라이브러리 내에 함수를 사용하여 동기화 문제 해결
프로그램 실행 및 인자 전달	- 프로그램 실행 시 파일명과 프로듀서 및 컨슈머의 수를 인자로 받아 실행하기. - 예: ./prod_cons ./sample 1 2
성능 최적화 및 동시성 향상	- 가능한 많은 동시성을 달성하고 빠른 실행 시간을 위해 프로그램을 최적화
실행 시간 측정	- clock_gettime() 또는 gettimeofday()를 사용하여 프로그램의 실행 시간을 측정하고, 스레드 수에 따른 성능을 비교

3. Concepts

프로젝트 설명에 앞서, 주요 개념인 스레드와 멀티스레드, Synchronization에 관해 설명하고, 추가로 뮤텝스와 세마포어에 대해 설명하도록 하겠다.

3-1. Thread

스레드는 CPU에서 실행되는 프로그램의 가장 작은 실행 단위이다. 즉, 스레드는 프로세스 내에서 실행되는 하나의 작업 흐름을 의미한다. 일반적으로 하나의 프로세스는 단일 스레드로 구성되어 한 번에 하나의 작업만 수행할 수 있지만, 멀티스레드를 지원하는 시스템에서는 여러 스레드가 하나의 프로세스 내에서 동시에 실행될 수 있다. 스레드를 지원하는 시스템에서는 프로세스 제어 블록(PCB)이 확장되어 스레드에 대한 정보를 포함하게 된다.

3-2. Multi-Thread

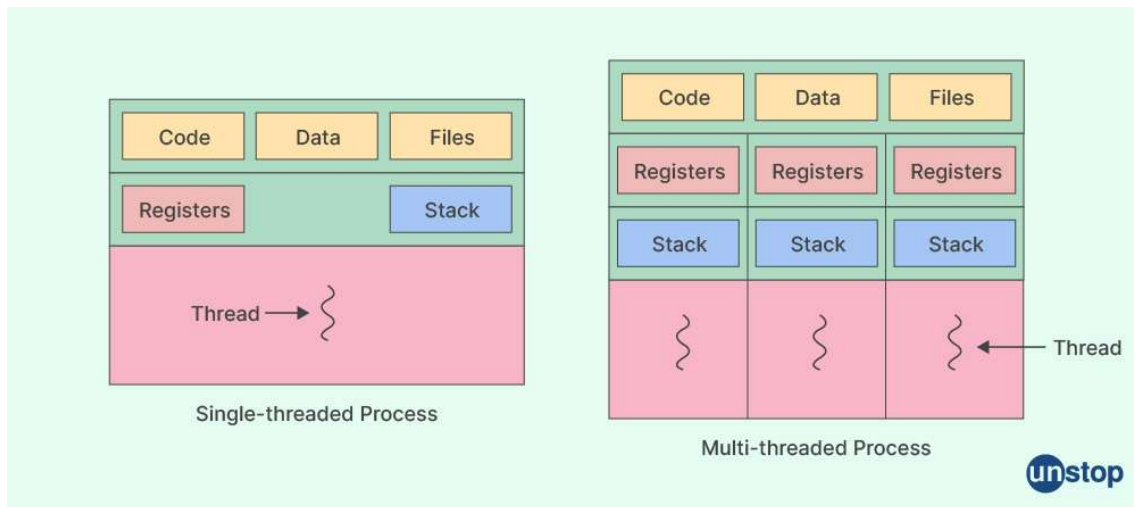


그림 1 싱글-스레드와 멀티-스레드

멀티스레드란, 하나의 프로세스에서 여러 개의 스레드를 동시에 실행하여 여러 작업을 병렬로 처리하는 개념이다. 스레드는 CPU 활용의 기본 단위로, 현대 운영체제에서는 대부분 멀티스레드를 지원하여 단일 프로세스에서 여러 작업을 동시에 실행할 수 있다. 하나의 스레드는 고유의 스레드 ID, 프로그램 카운터(PC), 레지스터 집합, 스택 등을 가지고 있지만, 프로세스 내의 메모리 자원을 공유한다.

멀티스레드 프로그래밍의 장점은 다음과 같다. :

- **경제성** : 새로운 프로세스를 생성하는 것은 메모리와 자원을 더 많이 소모하지만, 스레드는 프로세스의 자원을 공유하므로 스레드를 생성하고 전환하는 것이 오버헤드를 줄일 수 있다.
- **자원 공유** : 스레드는 데이터와 힙의 세그먼트를 공유하기 때문에 IPC가 쉬워진다.
- **Interactive system 구현 용이** : 다양하게 들어오는 사용자의 입력 사항을 각 스레드가 일을 분담하여 반응이 빠르며 이러한 시스템의 구현도 쉬워진다.

하지만, 멀티 스레딩에는 단점 또한 존재한다. :

- **동기화 문제** : 멀티스레드 환경에서는 여러 스레드가 동일한 자원에 접근할 수 있으므로, 자원 접근을 올바르게 동기화하지 않으면 경쟁 상태가 발생할 수 있다. 또한 데드락과 기아 상태도 발생한다.
- **디버깅 및 테스트의 어려움** : 멀티스레딩 프로그램은 여러 스레드가 동시에 실행되므로 디버깅이 어렵다. 동시성 이슈로 인해 발생하는 문제는 테스트

트에서 잡히지 않을 가능성이 크며, 실행 환경이나 실행 횟수에 따라 다르게 나타날 수 있다.

- **복잡성 증가** : 멀티스레드 프로그램은 설계 및 구현이 복잡하다. 스레드 간의 자원 공유와 동기화 문제를 해결하기 위해 세심한 설계가 필요하며, 잘못 설계되면 예상치 못한 오류가 발생할 수 있다.

3-3. Synchronization

Synchronization은 멀티스레드 프로그래밍에서 여러 스레드가 공유 자원에 안전하게 접근하도록 제어하는 기법이다. 멀티스레드 환경에서는 여러 스레드가 동일한 자원에 동시에 접근하려고 할 때 **경쟁 상태(Race Condition)**가 발생할 수 있다. 이는 스레드들이 자원을 접근하는 순서에 따라 잘못된 결과를 초래하는 상황을 의미한다. 이러한 문제를 해결하기 위해서는 **크리티컬 섹션(Critical Section)**이라는 개념이 도입된다. 크리티컬 섹션은 공유 자원에 접근하는 코드의 중요한 부분으로, 여러 스레드가 동시에 접근해서는 안 되는 구역이다. 따라서, 크리티컬 섹션을 보호하기 위해서는 스레드 간의 접근을 조정해야 하며, 이를 위한 동기화 메커니즘이 필요하다.

크리티컬 섹션 문제를 해결하기 위한 세 가지 중요한 조건이 존재한다. 첫 번째는 **상호 배제**로, 하나의 스레드가 크리티컬 섹션을 실행하고 있을 때 다른 스레드는 그 섹션에 접근하지 못하게 하는 것이다. 두 번째는 **진행성**으로, 크리티컬 섹션을 실행 중인 스레드가 없을 경우 접근하려는 스레드들 간에 빠르게 순서를 결정할 수 있어야 한다. 마지막으로 **한정 대기**가 있으며, 이는 스레드가 크리티컬 섹션에 들어가고자 요청한 후 다른 스레드들이 반복해서 그 섹션에 먼저 들어가는 일이 과도하게 일어나지 않도록 제한하는 것이다.

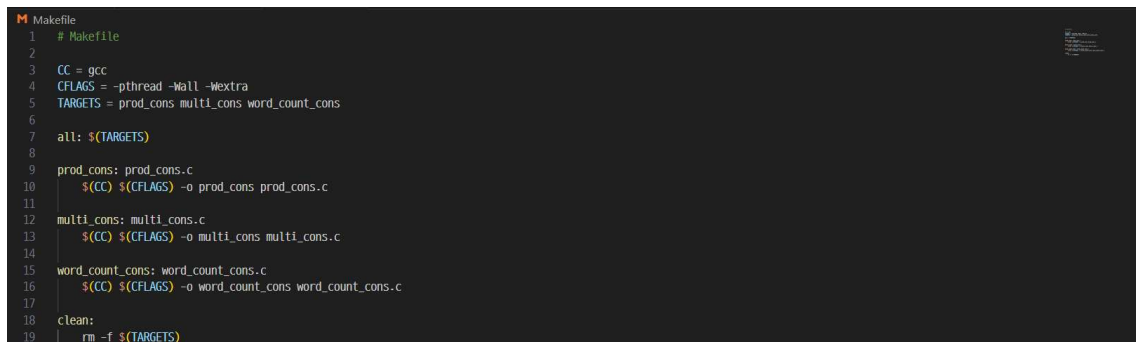
이러한 요구를 충족시키기 위해 다양한 동기화 도구들이 사용된다. 그 중 대표적인 것이 **뮤텍스(Mutex)**와 **세마포어(Semaphore)**이다. 뮤텍스는 상호 배제를 구현하는 가장 단순한 도구로, 스레드가 크리티컬 섹션에 들어가기 전에 락을 걸고, 나올 때 락을 해제하는 방식으로 작동한다. 락이 걸려 있는 동안 다른 스레드는 크리티컬 섹션에 접근할 수 없으며, 락이 해제될 때까지 대기해야 한다. 세마포어는 주어진 자원의 수를 제한할 수 있으며 여

러 스레드가 제한된 자원에 접근하는 것을 제어한다. 세마포어는 카운팅 세마포어와 이진 세마포어로 나뉘는데, 이진 세마포어는 뮤텝스와 유사하게 동작한다.

이처럼, 멀티스레드 환경에서는 스레드 간 자원 접근을 올바르게 관리하지 않으면 프로그램의 예측 불가능한 동작이나 데이터 손상이 발생할 수 있다. Synchronization은 이러한 문제를 예방하고, 스레드 간 자원 접근을 조정하여 안정적인 멀티스레드 프로그램을 구현하는 데 필수적인 역할을 한다.

4. Implementation

4-1. Makefile()



```
1 # Makefile
2
3 CC = gcc
4 CFLAGS = -pthread -Wall -Wextra
5 TARGETS = prod_cons multi_cons word_count_cons
6
7 all: $(TARGETS)
8
9 prod_cons: prod_cons.c
10     $(CC) $(CFLAGS) -o prod_cons prod_cons.c
11
12 multi_cons: multi_cons.c
13     $(CC) $(CFLAGS) -o multi_cons multi_cons.c
14
15 word_count_cons: word_count_cons.c
16     $(CC) $(CFLAGS) -o word_count_cons word_count_cons.c
17
18 clean:
19     rm -f $(TARGETS)
```

그림 2 Makefile 코드 전문

Makefile은 다수의 C 프로그램을 컴파일하는 데 사용된다. 컴파일로 gcc를 사용했으며, -pthread 플래그로 스레드 라이브러리를 사용해 스레드 동작을 지원하게 하였다. 또한, -Wall, -Wextra로 경고 메시지를 표시하여 코드의 잠재적인 오류를 잡아낼 수 있게 컴파일했다. make 명령어로 세 개의 코드를 한 번에 컴파일 할 수 있으며, make clean 명령어로 프로그램을 삭제할 수 있다.

4-2. prod_cons.c 파일

해당 파일은 프로듀서-컨슈머(Producer-Consumer) 패턴을 멀티스레드 환경에서 구현한 프로그램이다. 한 개의 프로듀서가 데이터를 생산하고, 한

개의 컨슈머가 그 데이터를 소비하는 과정을 스레드 기반으로 처리하는 것이다. 미리 제공된 코드 중에서 코드를 수정한 부분만 설명하도록 하겠다.

```

7  typedef struct sharedobject {
8      FILE *rfile;
9      int linenum;
10     char *line;
11     pthread_mutex_t lock;
12     pthread_cond_t full; // 데이터가 사용할 수 있을 때 신호를 보내는 조건 변수
13     pthread_cond_t empty; // 버퍼가 비어 있을 때 신호를 보내는 조건 변수
14     int full_flag; // 버퍼에 데이터가 있는지 여부를 나타내는 플래그
15 } so_t;

```

그림 3 오브젝트 구조체

프로듀서와 컨슈머 간의 동기화를 더욱 효율적으로 관리하기 위해 pthread_cond_t 조건 변수 full과 empty를 추가했다. 이 변수들은 각각 스레드가 필요할 때만 작업을 수행하게 도와주며, busy-waiting을 방지하고 스레드 간의 대기 시간을 효율적으로 처리한다.

또한 full_flag를 추가하여 버퍼에 데이터가 있는지 없는지 확인하는 역할을 수행하게 된다. 프로듀서가 데이터를 생산하면 full_flag를 1로 설정하여 버퍼가 꽉 찼음을 알리고, 컨슈머가 데이터를 소비한 후에는 0으로 바꿔 버퍼가 비었음을 알린다. 이 플래그는 조건 변수와 결합하여 프로듀서와 컨슈머가 올바른 시점에 작업을 수행하도록 도와준다.

```

26     while (1) {
27         pthread_mutex_lock(&so->lock);
28         // 버퍼가 비어 있을 때까지 대기
29         while (so->full_flag == 1) {
30             pthread_cond_wait(&so->empty, &so->lock);
31         }
32
33         read = getdelim(&line, &len, '\n', rfile);
34         if (read == -1) {
35             so->line = NULL; // 생산 종료를 나타냄
36             so->full_flag = 1; // 생산이 끝났음을 알림
37             pthread_cond_signal(&so->full);
38             pthread_mutex_unlock(&so->lock);
39             break;
40         }
41
42         so->linenum = i;
43         so->line = strdup(line); // 읽은 줄을 공유
44         i++;
45         so->full_flag = 1; // 버퍼가 꽉 찼음을 표시
46
47         pthread_cond_signal(&so->full); // 새로운 데이터가 사용 가능함을 알림
48         pthread_mutex_unlock(&so->lock);
49     }
50
51     free(line);
52     printf("Prod %x: %d lines\n", (unsigned int)pthread_self(), i);
53     *ret = 1;
54     pthread_exit(ret);
55 }

```

그림 4 void *producer(void *arg) 내에 무한 루프와 메모리 정리 부분

```

63     while (1) {
64         pthread_mutex_lock(&so->lock);
65         // 소비할 데이터가 있을 때까지 대기
66         while (so->full_flag == 0) {
67             pthread_cond_wait(&so->full, &so->lock);
68         }
69
70         line = so->line;
71         if (line == NULL) {
72             break; // 생산자가 종료되었으면 루프 탈출
73         }
74
75         printf("Cons %x: [%02d:%02d] %s", (unsigned int)pthread_self(), i, so->linenum, line);
76         free(so->line);
77         so->full_flag = 0; // 버퍼가 비었음을 표시
78
79         pthread_cond_signal(&so->empty); // 버퍼가 비었음을 알림
80         pthread_mutex_unlock(&so->lock);
81         i++;
82     }

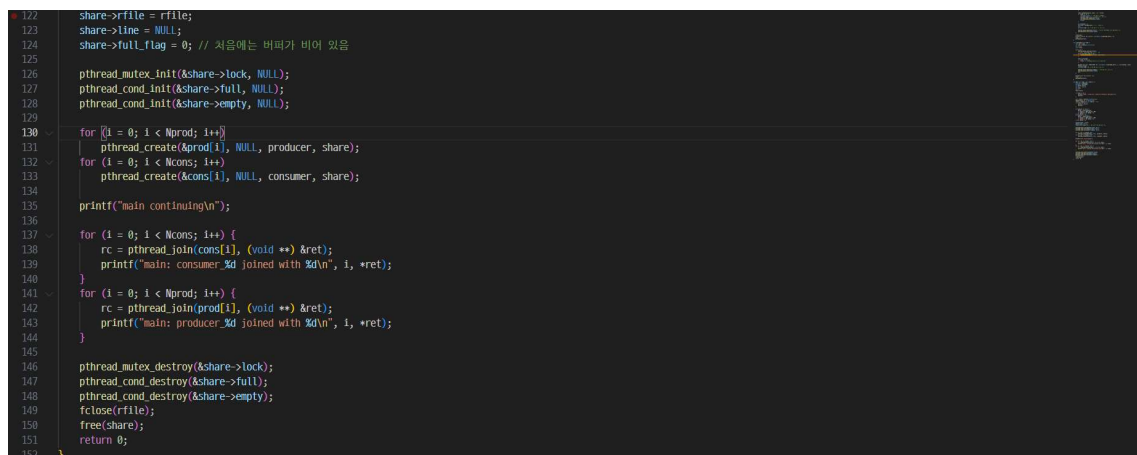
```

그림 5 void *consumer(void *arg) 내에 무한 루프 부분

기존의 코드에서는 단순히 full_flag 변수를 사용해 프로듀서와 컨슈머 간의 상태를 전달하고 있으며, 이 과정에서 busy-waiting 현상이 발생할 수 있다.

이를 해결하기 위해 조건 변수(pthread_cond_t)를 도입했다. 이 조건 변수는 중요한 역할을 수행하는데, pthread_cond_t full 변수는 버퍼에 새로운 데이터가 들어왔을 때, 컨슈머에게 신호를 보내 소비할 수 있음을 알리는데 사용된다. 반면에 pthread_cond_t empty는 버퍼가 비어 있음을 프로듀서에게 알리는 신호로 사용된다. 이를 통해 프로듀서가 데이터를 생산하려 할 때, 버퍼가 가득 차 있는 경우 불필요한 작업을 방지하고, 컨슈머가 데이터를 소비할 때는 버퍼에 데이터가 없으면 대기 상태로 들어가게 된다.

또한, 뮤텝스를 사용해 프로듀서와 컨슈머가 동시에 공유 데이터에 접근하지 않도록, 공유 자원에 접근할 때마다 락을 걸고, 작업이 끝나면 락을 해제하여 동시 접근을 방지한다.



```

122 share->rfile = rfile;
123 share->wline = NULL;
124 share->full_flag = 0; // 처음에는 버퍼가 비어 있음
125
126 pthread_mutex_init(&share->lock, NULL);
127 pthread_cond_init(&share->full, NULL);
128 pthread_cond_init(&share->empty, NULL);
129
130 for (i = 0; i < Nprod; i++)
131     pthread_create(&prod[i], NULL, producer, share);
132 for (i = 0; i < Ncons; i++)
133     pthread_create(&cons[i], NULL, consumer, share);
134
135 printf("main continuing\n");
136
137 for (i = 0; i < Ncons; i++) {
138     rc = pthread_join(cons[i], (void **)&ret);
139     printf("main: consumer %d joined with %d\n", i, *ret);
140 }
141 for (i = 0; i < Nprod; i++) {
142     rc = pthread_join(prod[i], (void **)&ret);
143     printf("main: producer %d joined with %d\n", i, *ret);
144 }
145
146 pthread_mutex_destroy(&share->lock);
147 pthread_cond_destroy(&share->full);
148 pthread_cond_destroy(&share->empty);
149 fclose(rfile);
150 free(share);
151 return 0;
152 }

```

그림 6 main() 내에 동적 메모리 초기화, 파일 열기 구문 후의 코드. 동기화 객체 초기화 및 스레드 생성, 동료 대기, 자원 해제 코드가 담겨 있다.

이 외에도 프로그램 종료 시 뮤텝스와 조건 변수 해제를 통해 시스템 자원을 반환하며, 메모리 누수를 방지하는 코드를 추가했다.

기존 코드를 수정하여 동기화와 자원 관리가 더 강화되게 하였으며, 멀티스레드 환경에서 busy-waiting 문제를 해결하고 성능을 최적화하였다.

4-3. multi_cons.c 파일

multi_cons.c는 멀티스레드 프로듀서-컨슈머 모델을 구현한 프로그램이

다. 해당 프로그램은 여러 생산자 스레드가 데이터를 생성하고, 여러 소비자 스레드가 데이터를 소비하는 구조로 되어 있다. 해당 프로그램은 텍스트 파일을 읽어 생산자가 파일의 줄을 읽고, 소비자가 줄을 소비하는 구조다. `so_t`라는 구조체를 사용하여 공유 데이터를 저장하고, 뮤텍스와 조건 변수를 통해 스레드 간의 동기화를 관리한다.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <pthread.h>
6
7 #define BUFFSIZE 4096
```

그림 7 헤더 파일 및 매크로 정의

멀티스레드를 위한 라이브러리를 추가하고, `BUFFSIZE`를 4096으로 정의하여 공유 버퍼의 크기를 4KB로 설정했다.

```
9 typedef struct sharedobject {
10     int nextin;
11     int nextout;
12     FILE* rfile;
13     char* line[BUFFSIZE];
14
15     pthread_mutex_t lock;
16     pthread_cond_t cond_producer; // 생산자 조건 변수
17     pthread_cond_t cond_consumer; // 소비자 조건 변수
18     int full; // 버퍼에 채워진 항목 수
19     int done; // 생산자 종료 플래그
20 } so_t;
```

그림 8 구조체 정의

`so_t` 구조체는 생산자와 소비자 간에 공유되는 데이터를 저장하는 구조체이다. `nextin`과 `nextout`은 각각 버퍼에 데이터를 넣고 빼는 위치를 나타내며, `rfile`은 읽기 파일 포인터, `line`은 버퍼를 저장하는 배열이다. `lock`, `cond_producer`, `cond_consumer`는 동기화 관련 변수이며, `full`은 현재 버퍼에 있는 항목 수, `done`은 생산자가 작업을 완료했는지를 나타내는 플래그이다.

```
22 void *producer(void *arg) {
23     so_t *so = arg;
24     size_t len = 0;
25     ssize_t read = 0;
26     char *line = NULL;
27     int *ret = malloc(sizeof(int));
28     int count = 0;
29
30     while (1) {
31         pthread_mutex_lock(&so->lock);
32
33         // 버퍼가 가득 차면 대기
34         while (so->full >= BUFFSIZE) {
35             pthread_cond_wait(&so->cond_producer, &so->lock);
36         }
37
38         read = getdelim(&line, &len, '\n', so->rfile);
39         if (read == -1) {
40             so->done = 1; // 생산자가 완료되었음을 표시
41             pthread_cond_broadcast(&so->cond_consumer); // 소비자에게 신호를 보냄
42             printf("Producer done.\n");
43             pthread_mutex_unlock(&so->lock);
44             free(line); // 완료 시 line 메모리 해제
45             break;
46         }
47
48         // 읽은 줄에 대한 메모리 할당
49         char *new_line = strdup(line);
50         if (new_line == NULL) {
51             perror("Failed to allocate memory");
52             free(line); // strdup 실패 시 line 해제
53             pthread_mutex_unlock(&so->lock);
54             break;
55         }
```

```

57     so->line[so->nextin] = new_line; // 줄을 버퍼에 저장
58     so->nextin = (so->nextin + 1) % BUFFSIZE;
59     so->full++;
60     count++;
61
62     printf("Prod %x: [%02d:%02d] %s", (unsigned int)pthread_self(), count, so->full, new_line);
63
64     pthread_cond_broadcast(&so->cond_consumer); // 소비자에게 신호를 보냄
65     pthread_mutex_unlock(&so->lock);
66 }
67
68 *ret = count;
69 pthread_exit(ret);
70 }

```

그림 9 생산자 함수

producer 함수는 인자로 받은 공유 객체를 사용하여 파일에서 줄을 읽어 버퍼에 저장하는 역할을 한다. pthread_mutex_lock(&so->lock);로 뮤텝스를 잠근다. 이후 조건문을 통해 버퍼가 가득 차면 pthread_cond_wait로 대기한다. getdelim 함수를 사용하여 파일에서 줄을 읽는다. 읽기 실패 시, done 플래그를 1로 설정하고 모든 소비자에게 신호를 보낸다. 읽은 줄을 메모리 할당(strdup)하여 버퍼에 저장하고, full 수를 증가시킨다. 마지막으로 pthread_cond_broadcast로 소비자에게 신호를 보내고, 뮤텝스를 해제한다.

즉, producer 함수는 파일에서 줄을 읽어 버퍼에 저장하고, 소비자에게 데이터가 준비되었음 알리는 역할을 한다.

```

72 void *consumer(void *arg) {
73     so_t *so = arg;
74     int count = 0;
75     char *line;
76     int *ret = malloc(sizeof(int));
77
78     while (1) {
79         pthread_mutex_lock(&so->lock);
80
81         // 데이터가 없으면 대기
82         while (so->full <= 0 && !so->done) {
83             pthread_cond_wait(&so->cond_consumer, &so->lock);
84         }
85
86         if (so->full <= 0 && so->done) { // 생산자가 완료되었고 버퍼가 비어 있으면
87             pthread_mutex_unlock(&so->lock);
88             break;
89         }
90
91         line = so->line[so->nextout]; // 소비할 줄 가져오기
92         if (line == NULL) {
93             pthread_mutex_unlock(&so->lock);
94             break;
95         }
96
97         // 소비한 줄 출력 및 메모리 해제
98         printf("Cons %x: [%02d:%02d] %s", (unsigned int)pthread_self(), count, so->nextout, line);
99         free(line);
100        so->line[so->nextout] = NULL;
101        so->nextout = (so->nextout + 1) % BUFFSIZE;
102        so->full--;
103        count++;
104
105        pthread_cond_broadcast(&so->cond_producer); // 생산자에게 신호를 보냄
106        pthread_mutex_unlock(&so->lock);
107    }
108
109    *ret = count;
110    pthread_exit(ret);
111 }

```

그림 10 소비자 함수

consumer 함수는 버퍼에서 줄을 가져와 소비하고 출력하는 역할을 한다. 이 또한 producer 함수와 비슷하게 뮤텝스를 잠근 후, 버퍼가 비어 있으면 대기를 하고, 버퍼에 내용이 찾으면 줄을 읽어 출력한다. 생산자가 종료된 후 버퍼가 비어 있으면 루프를 종료한다. 소비한 줄을 출력한 후 메모리

를 해제하고, nextout을 업데이트한다. 이후 pthread_cond_broadcast로 생산자에게 신호를 보낸다.

요약하자면, 소비자 함수는 버퍼에서 줄을 읽어 출력하고, 메모리를 해제하며, 생산자가 종료되면 스레드를 종료하는 역할을 한다.

```

113 int main (int argc, char *argv[]) {
114     int *ret;
115     int i;
116     FILE *rfile;
117     int Nprod, Ncons;
118     pthread_t prod[100];
119     pthread_t cons[100];
120
121     if (argc < 4) {
122         printf("usage: ./multi_cons <readfile> #Producer #Consumer\n");
123         exit(0);
124     }
125
126     so_t *share = malloc(sizeof(so_t));
127     memset(share, 0, sizeof(so_t));
128     rfile = fopen(argv[1], "r");
129
130     if (rfile == NULL) {
131         perror("rfile");
132         exit(0);
133     }
134
135     Nprod = atoi(argv[2]);
136     Ncons = atoi(argv[3]);
137
138     share->rfile = rfile;
139     for (i = 0; i < BUFFSIZE; i++) share->line[i] = NULL;
140
141     // 동기화 프리미티브 초기화
142     pthread_mutex_init(&share->lock, NULL);
143     pthread_cond_init(&share->cond_producer, NULL);
144     pthread_cond_init(&share->cond_consumer, NULL);
145     share->full = 0;
146     share->done = 0;
147     share->nextin = 0;
148     share->nextout = 0;
149
150     // 생산자 스레드 생성
151     for (i = 0; i < Nprod; i++) {
152         pthread_create(&prod[i], NULL, producer, share);
153     }
154
155     // 소비자 스레드 생성
156     for (i = 0; i < Ncons; i++) {
157         pthread_create(&cons[i], NULL, consumer, share);
158     }
159
160     // 스레드 조인 및 결과 출력
161     printf("\n\n");
162     for (i = 0; i < Ncons; i++) {
163         pthread_join(cons[i], (void**)&ret);
164         printf("main: consumer_%d joined with %d\n", i, *ret);
165         free(ret);
166     }
167     for (i = 0; i < Nprod; i++) {
168         pthread_join(prod[i], (void**)&ret);
169         printf("main: producer_%d joined with %d\n", i, *ret);
170         free(ret);
171     }
172
173     // 정리
174     fclose(rfile);
175     free(share);
176     pthread_mutex_destroy(&share->lock);
177     pthread_cond_destroy(&share->cond_producer);
178     pthread_cond_destroy(&share->cond_consumer);
179     pthread_exit(NULL);
180     return 0;
181 }

```

그림 11 메인 함수 전문

메인 함수는 프로그램의 시작점이며, 인자 수를 확인하고, so_t 구조체를 동적으로 할당한다. 파일을 열고 인자로 받은 생산자와 소비자 수를 설정한다. 동기화 프리미티브를 초기화하고, 생산자와 소비자 스레드를 생성하는 역할을 한다. 이제 메인 함수는 모든 소비자 스레드가 종료될 때까지 대기하고, 그 결과를 출력하는 역할을 하며 마지막으로 모든 자원을 해제하고 프로그램을 종료하는 역할을 한다.

이 프로그램은 멀티스레딩 환경에서 생산자와 소비자가 어떻게 상호작용하는지를 보여준다. 스레드 안정성을 보장하기 위해 뮤텝스와 조건 변수를 사용하여 데이터 무결성을 유지한다.

4-4. word_count_cons.c 파일

word_count_cons.c 파일은 멀티 스레딩을 사용하여 파일에서 데이터를 읽고, 읽은 데이터를 소비하며, 각 문자의 통계를 계산하는 생산자-소비자 패턴을 구현한 프로그램이다. 파일에서 줄을 읽고, 그 줄들을 소비하는 소비자 스레드가 알파벳의 빈도를 세는 기능을 포함하고 있다. 생산자-소비자 패턴은 위의 multi_cons.c 파일과 같기에 알파벳의 빈도를 세는 기능을 위해 추가된 코드만을 설명하도록 하겠다.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <pthread.h>
6
7  #define BUFFSIZE 4096
8  #define ASCII_SIZE 256
9
10 typedef struct sharedobject {
11     int nextin;
12     int nextout;
13     FILE* rfile;
14     char* line[BUFFSIZE];
15
16     pthread_mutex_t lock;
17     pthread_cond_t cond_producer; // 생산자 조건 변수
18     pthread_cond_t cond_consumer; // 소비자 조건 변수
19     int full; // 버퍼에 채워진 항목 수
20     int done; // 생산자 종료 플래그
21     int stat[ASCII_SIZE]; // 알파벳 통계
22 } so_t;

```

그림 12 전처리기 지시문 및 구조체 정의

해당 부분에서 추가된 부분은 ASCII_SIZE 라는 알파벳 문자 수를 정의하는 매크로이다. so_t 구조체는 같으며, 알파벳의 빈도를 셀 때 필요한 배열이 포함된다.

생산자 스레드 함수는 동일한 코드를 사용하기 때문에 코드 설명은 넘어가도록 하겠다.

```

99 // 소비한 줄의 문자 수 카운트 /* multi_cons에서 추가된 코드 */
100 for (char *ptr = line; *ptr != '\0'; ptr++) {
101     if (*ptr < ASCII_SIZE) {
102         so->stat[(unsigned char)*ptr]++;
103     }
104 }

```

그림 13 소비자 스레드 함수 중 추가된 구문 (소비한 줄의 문자 수 카운트)

consumer 함수는 데이터가 없으면 대기하고, 줄을 소비할 때 해당 줄에 포함된 문자 빈도를 계산하여 stat 배열에 저장한다. 소비한 줄을 출력하고 메모리를 해제한 후, 생산자 스레드에 신호를 보내게 된다.

```

122  /* multi_cons에서 추가된 코드 */
123  void print_statistics(so_t *so) {
124      printf("*** print out distributions *** \n");
125      printf(" #ch  freq \n");
126
127      int sum = 0;
128      for (int i = 0; i < ASCII_SIZE; i++) {
129          sum += so->stat[i];
130      }
131
132      for (int i = 0; i < 30; i++) {
133          int num_star = so->stat[i] * 80 / sum; // 막대 차트 생성을 위한 예시
134          printf("[%3d]: %4d \t", i + 1, so->stat[i]);
135          for (int j = 0; j < num_star; j++) {
136              printf("#");
137          }
138          printf("\n");
139      }
140
141      printf("      A      B      C      D      E      F      G      H      I      J      K      L      M      N      O      P
142      printf("%8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d\n",
143             so->stat['A'] + so->stat['a'], so->stat['B'] + so->stat['b'], so->stat['C'] + so->stat['c'], so->stat['D'] + so->stat['d'], so->stat['E'] + so->stat
144             so->stat['F'] + so->stat['f'], so->stat['G'] + so->stat['g'], so->stat['H'] + so->stat['h'], so->stat['I'] + so->stat['i'], so->stat['J'] + so->stat
145             so->stat['K'] + so->stat['k'], so->stat['L'] + so->stat['l'], so->stat['M'] + so->stat['m'], so->stat['N'] + so->stat['n'], so->stat['O'] + so->stat
146             so->stat['P'] + so->stat['p'], so->stat['Q'] + so->stat['q'], so->stat['R'] + so->stat['r'], so->stat['S'] + so->stat['s'], so->stat['T'] + so->stat
147             so->stat['U'] + so->stat['u'], so->stat['V'] + so->stat['v'], so->stat['W'] + so->stat['w'], so->stat['X'] + so->stat['x'], so->stat['Y'] + so->stat
148             so->stat['Z'] + so->stat['z']);
149  }

```

그림 14 통계 출력함수

print_statistics 함수는 소비한 알파벳의 빈도를 출력한다. 각 알파벳에 대한 빈도를 기반으로 차트를 생성하게 된다. 해당 코드는 char_stat.c 파일에서 일부 발췌했다.

```

179  // 통계 배열 초기화 /* multi_cons에서 추가된 코드 */
180  memset(&share->stat, 0, sizeof(share->stat));
181
182  // 동기화 프리미티브 초기화
183  pthread_mutex_init(&share->lock, NULL);
184  pthread_cond_init(&share->cond_producer, NULL);
185  pthread_cond_init(&share->cond_consumer, NULL);
186  share->full = 0; // 버퍼의 초기 상태 설정
187  share->done = 0; // 초기 종료 상태 설정
188  share->nextin = 0; // 다음 입력 위치 초기화
189  share->nextout = 0; // 다음 출력 위치 초기화
190
191  // 생산자 스레드 생성
192  for (i = 0; i < Nprod; i++) {
193      pthread_create(&prod[i], NULL, producer, share);
194  }
195
196  // 소비자 스레드 생성
197  for (i = 0; i < Ncons; i++) {
198      pthread_create(&cons[i], NULL, consumer, share);
199  }
200
201  // ... (중략) ...
212  printf("Consumer %d consumed %d lines.\n", i, *ret);
213  free(ret); // Consumer result 메모리 해제
214  }
215
216  fclose(rfile);
217
218  // 통계 출력
219  print_statistics(share);

```

그림 17 main 함수 중 통계 배열을 초기화하는 부분과 통계를 출력하는 부분 일부

main 함수는 생산자 및 소비자 스레드를 생성하고, 각각 스레드가 완료될 때까지 대기한다. 모든 소비자 및 생산자 스레드가 완료된 후에는 통계를 출력하고 자원을 해제한다.

5. Result

이렇게 제작된 프로그램 3가지를 이용해서 여러 출력물을 보여주도록

하겠다.

```
assam_bokyeong21@assam:~/hw2$ ls
Makefile multi_cons.c prod_cons.c samples word_count_cons.c
assam_bokyeong21@assam:~/hw2$ make
gcc -pthread -Wall -Wextra -o prod_cons prod_cons.c
prod_cons.c: In function 'main':
prod_cons.c:93:20: warning: unused variable 't' [-Wunused-variable]
   93 |     int rc;    long t;
       |                   ^
prod_cons.c:93:9: warning: variable 'rc' set but not used [-Wunused-but-set-variable]
   93 |     int rc;    long t;
       |     ^~
gcc -pthread -Wall -Wextra -o multi_cons multi_cons.c
gcc -pthread -Wall -Wextra -o word_count_cons word_count_cons.c
word_count_cons.c: In function 'consumer':
word_count_cons.c:101:22: warning: comparison is always true due to limited range of data type [-Wtype-limits]
   101 |         if (*ptr < ASCII_SIZE) {
       |                     ^
assam_bokyeong21@assam:~/hw2$ ls
Makefile multi_cons multi_cons.c prod_cons prod_cons.c samples word_count_cons word_count_cons.c
assam_bokyeong21@assam:~/hw2$
```

그림 17 Makefile을 실행한 모습

```
assam_bokyeong21@assam:~/hw2$ ./prod_cons samples/char_stat.c
main continuing
Cons_70f66640: [00:00] #include <stdio.h>
Cons_70f66640: [01:01] #include <stdlib.h>
Cons_70f66640: [02:02] #include <unistd.h>
Cons_70f66640: [03:03] #include <string.h>
Cons_70f66640: [04:04]
Cons_70f66640: [05:05]
Cons_70f66640: [06:06] #define MAX_STRING_LENGTH 30
Cons_70f66640: [07:07] #define ASCII_SIZE 256
Cons_70f66640: [08:08]
Cons_70f66640: [09:09]
Cons_70f66640: [103:103]
Cons_70f66640: [104:104]     if (line != NULL) free(line);
Cons_70f66640: [105:105]     // Close the file
Cons_70f66640: [106:106]     fclose(rfile);
Cons_70f66640: [107:107]
Cons_70f66640: [108:108]     return 0;
Cons_70f66640: [109:109] }Prod_71767640: 110 Lines
Cons: 110 Lines
main: consumer_0 joined with 110
main: producer_0 joined with 110
```

그림 18 prod_con 파일을 실행한 모습

```
assam_bokyeong21@assam:~/hw2$ ./multi_cons
usage: ./multi_cons <readfile> #Producer #Consumer
assam_bokyeong21@assam:~/hw2$ ./multi_cons samples/hw2.txt
usage: ./multi_cons <readfile> #Producer #Consumer
assam_bokyeong21@assam:~/hw2$ ./multi_cons samples/hw2.txt 1 2

Prod_b3798640: [01:01] HW2: Multi-threaded word count
Prod_b3798640: [02:02]
Prod_b3798640: [03:03] Due date: Oct. 19th
Prod_b3798640: [04:04]
Prod_b3798640: [05:05] The second homework is about multi-thread programming with some synchronization. Thread is a unit of execution; a thread has execution context, which includes the registers, stack. Note that address space (memory) is shared among threads in a process, so there is no clear separation and protection for memory access among threads.
Cons_b2796640: [00:00] HW2: Multi-threaded word count
Cons_b2796640: [01:01]
Cons_b2796640: [02:02] Due date: Oct. 19th
Cons_b2796640: [03:03]
Cons_b2796640: [04:04] The second homework is about multi-thread programming with some synchronization. Thread is a unit of execution; a thread has execution context, which includes the registers, stack. Note that address space (memory) is shared among threads in a process, so there is no clear separation and protection for memory access among threads.
Prod_b3798640: [06:06]
Prod_b3798640: [07:07] The example code includes some primitive code for multiple threads usage. It basically tries to read a file and print it out on the screen. It consists of three threads: the main thread is for admin info second thread serves as a producer: reads lines from a file and puts the line string on the shared buffer third thread serves as a consumer: gets strings from the shared buffer
Cons_b2796640: [19:30]
Cons_b2796640: [20:31] http is a program that shows the execution of threads in the system.
Cons_b2796640: [21:32]
Cons_b2796640: [22:33] Measure & compare the execution time for different # of threads
Cons_b2796640: [23:34]
Cons_b2796640: [24:35] Happy hacking! Seehwanmain: consumer_0 joined with 11
main: consumer_1 joined with 25
main: producer_0 joined with 36
```

그림 19 multi_cons 파일을 실행한 모습 (생산자 1, 소비자 2)

```
Cons_837fe640: [03:23]
Cons_837fe640: [04:24] You can download some example input source code from the Link: [http://mobile-os.dankook.ac.kr/data/FreetsD9-orig.tar] or
Cons_837fe640: [05:25] you can use /opt/FreetsD9-orig.tar from our server.
Cons_837fe640: [06:26]
Cons_837fe640: [07:27] Please make a document so that I can follow it to build/compile and run the code. It would be better if the document included an introduction and some important implementation details of your program structure.
Cons_837fe640: [08:28]
Cons_837fe640: [09:29] To measure the execution time, consider using clock_gettime() or gettimeofday().
Cons_837fe640: [10:30]
Cons_837fe640: [11:31] http is a program that shows the execution of threads in the system.
Cons_837fe640: [12:32]
Cons_837fe640: [13:33] Measure & compare the execution time for different # of threads
Cons_837fe640: [14:34]
Cons_837fe640: [15:35] Happy hacking! Seehwanmain: consumer_0 joined with 20
main: consumer_1 joined with 16
main: producer_0 joined with 5
main: producer_1 joined with 24
main: producer_2 joined with 7
```

그림 20 multi_cons 파일을 실행한 모습 (생산자 3, 소비자 2)

```

Prod_4b423640: [21262890:498]         errno = ENAMETOOLONG;
Prod_4b423640: [21262881:499]         return (-1);
Prod_4b423640: [21262882:500]         }
Prod_4b423640: [21262883:501]         len = offsetof(struct sockaddr_un, sun_path[1]);
Prod_4b423640: [21262884:502]
Prod_4b423640: [21262885:503]         if (connect(fd, (void *)&sun, len) < 0) {
Prod_4b423640: [21262886:504]             close(fd);
Prod_4b423640: [21262887:505]             fd = -1;
Prod_4b423640: [21262888:506]         }
Prod_4b423640: [21262889:507]     }
Prod_4b423640: [21262890:508]
Prod_4b423640: [21262891:509]     /*
Prod_4b423640: [21262892:510]     * handle the open flags by shutting down appropriate directions
Prod_4b423640: [21262893:511]     */
Prod_4b423640: [21262894:512]     if (fd >= 0) {
Prod_4b423640: [21262895:513]         switch(flags & O_ACCMODE) {
Prod_4b423640: [21262896:514]             case O_RDONLY:
Prod_4b423640: [21262897:515]                 if (shutdown(fd, SHUT_WR) == -1)
Prod_4b423640: [21262898:516]                     warn(NULL);
Prod_4b423640: [21262899:517]                 break;
Prod_4b423640: [21262900:518]             case O_WRONLY:
Prod_4b423640: [21262901:519]                 if (shutdown(fd, SHUT_RD) == -1)
Prod_4b423640: [21262902:520]                     warn(NULL);
Prod_4b423640: [21262903:521]                 break;
Prod_4b423640: [21262904:522]             default:
Prod_4b423640: [21262905:523]                 break;
Prod_4b423640: [21262906:524]         }
Prod_4b423640: [21262907:525]     }
Prod_4b423640: [21262908:526]     return(fd);
Prod_4b423640: [21262909:527] }
Prod_4b423640: [21262910:528]
Prod_4b423640: [21262911:529] #endif
Prod_4b423640: [21262912:530] Prod_4b423640: [21262913:531] # $FreeBSD: release/9.0.0/bin/cat/Makefile 198148 2009-10-15 18:17:29Z ru $
Prod_4b423640: [21262914:532]
Prod_4b423640: [21262915:533] PROG= cat
Prod_4b423640: [21262916:534]
Prod_4b423640: [21262917:535] .include <bsd.prog.mk>
Prod_4b423640: [21262918:536] .Producer done.
Cons_4ac22640: [10409863:46] .\" Redistribution and use in source and binary forms, with or without
Cons_4ac22640: [10409864:47] .\" modification, are permitted provided that the following conditions
Cons_4ac22640: [10409865:48] .\" are met:

```

그림 21 word_count_cons 실행 중간 모습 (생산자 1, 소비자 2)

```

Cons_4a421640: [10852921:579]
Cons_4a421640: [10852922:580] .include <bsd.prog.mk>
Cons_4a421640: [10852923:581] Consumer 0 consumed 10409994 lines.
Consumer 1 consumed 10852924 lines.
*** print out distributions ***
#ch freq
[ 1]: 0
[ 2]: 2919
[ 3]: 2908
[ 4]: 3308
[ 5]: 3022
[ 6]: 2983
[ 7]: 2968
[ 8]: 3072
[ 9]: 3857
[10]: 13755177
[11]: 21211038
[12]: 2885
[13]: 7940
[14]: 22686
[15]: 2853
[16]: 3011
[17]: 2995
[18]: 2894
[19]: 2917
[20]: 2925
[21]: 2948
[22]: 2878
[23]: 2959
[24]: 2891
[25]: 2878
[26]: 2858
[27]: 2992
[28]: 4143
[29]: 2992
[30]: 2992
A B C D E F G H I J K L M N O P Q R S T U V
26327129 8637394 19823849 16795371 48396112 13224608 6957413 9073934 26098475 643571 2928934 15582860 10063744 23020999 21108612 11704796 926347 23600474 24218888 30945280 10929599 4616392 3
557992 11733166 4360875 1046831

```

그림 22 word_count_cons 실행 완료 모습 (생산자 1, 소비자 2)

6. Build Environment

3개의 파일을 실행하기 위해서는 GCC 컴파일러가 필요하며, 해당 프로그램은 Makefile을 사용하여 빌드되기 때문에 프로그램이 위치한 디렉토리

내에서 다음과 같은 명령어를 사용하면 프로그램이 실행된다.

\$make / \$make all : 3개의 모든 파일의 실행 파일을 빌드

\$make clean : 모든 실행 파일 정리

각 파일을 실행하기 위한 명령어를 알려주겠다.

```
./prod_cons <readfile>
```

```
./multi_cons <readfile> #Producer #Consumer
```

```
./multi_cons <readfile> #Producer #Consumer
```

7. Lesson

7-1. Conclusion

이번 프로젝트를 통해 멀티스레딩 기법을 사용하여 여러 소비자가 동시에 자원을 소비하는 문제를 성공적으로 해결할 수 있었다. 스레드 동기화와 상호 배제를 적절히 구현함으로써 데이터의 일관성을 유지하고, 경합 상태를 피할 수 있었다. 특히 뮤텍스와 세마포어와 같은 동기화 메커니즘을 활용해 스레드 간의 충돌을 방지하는 방법을 직접 경험함으로써, 운영체제에서의 병렬 프로그래밍의 중요성과 복잡성을 더 깊이 이해하게 되었다.

7-2. Review

저번 과제보다 난이도가 급격하게 올라가서 해결하는데 애를 쓴 것 같다. 특히 다중 컨슈머 지원 부분을 해결할 때 초반에 전혀 감이 오지 않았다. 프로그램 알고리즘을 짜고, 여러 스레드들간의 동기화 문제를 잘 잡았다고 생각했는데 프로그램 실행할 때마다 오류에 직면했다. 특히 동기화가 정말 중요하다는 것을 매우 매우 뼈저리게 깨달았다. 중반부에 세그멘테이션 오류가 났는데, 이것 잡으려고 많은 시간을 쏟아부은 것 같다.

덕분에 멀티스레딩 구현에서 중요한 것들이 무엇인지 정말 온몸으로 느

졌고, 오랜만에 코딩하면서 희열, 분노 등 다양한 감정을 느낀 것 같다.

7-3. Using ChatGPT

- 과제요 요구 사항을 정리하는 부분에서 ChatGPT의 도움을 받았다.
- 처음 제공 받은 `prod_con.c` 파일의 코드 분석하는 부분에서 사용했다. 추가로 `pthread` 라이브러리 사용법 등 조건 변수, 함수 사용법에 대해 모를 때 사용했다.
- 멀티 컨슈머 기능을 구현할 때 세그멘테이션 오류를 잡을 때 사용했다. `gdb`를 이용해 디버깅 정보를 ChatGPT에게 입력했고, 이를 해석하고 어느 부분에서 오류 났는지 알기 위해 사용했다.