운영체제 과제 4

201921162 송민기

1-1 코드 설계

주어진 텍스트 파일의 단어의 첫 알파벳의 개수를 세는 문제이기 때문에 만약 텍스트 파일의 길이가 매우 길 경우에 하나의 스레드만을 이용한다면 오랜 시간이 걸릴 수가 있다. 따라서 이것의 효율을 높이기 위해서 세 개의 스레드를 이용하여 병렬처리를 구현하였고, 알파벳의 총 개수를 저장하는 배열에 임의로 세 개의 스레드가 순서 없이 계속해서 들어온다면 혼선이 생길 수 있기 때문에 이것에 대해서는 배열을 저장하는 부분을 Mutex Lock을 이용하여 임계구역으로 처리해주었다.
코드의 플로우를 간단하게 설명하면 다음과 같다.

1. main() 함수에서 파일 입출력을 통하여 input.txt 파일을 읽어 온 후 큰 공간을 가진 sentence[] 배열에 저장을 한다.
2. 전체 텍스트의 길이를 카운트 한 후 삼등분하여 각 스레드의 구조체에 시작과 끝을 저장시킨다.
3. 단어 중간에서 끊길 수 있기 때문에 공백을 각 스레드의 끝으로 지정한다.
4. Mutex Lock 생성, 스레드 생성, count() 함수 실행
5. count() 함수에서는 각 스레드의 시작과 끝까지 카운트 작업을 수행한다.
6. Mutex Lock 에 의하여 임계구역 설정
7. count() 모두 완료 후 join, 결과 값 출력

1-2 코드의 각 기능 설명

```
FILE *fp;
    fp = fopen(filename, "r");
    while (!feof(fp))
    {
        fgets(sentence, MAX, fp);
    }
    length = strlen(sentence);
```

파일 입출력을 통하여 input.txt(=filename)을 읽어 온다. 파일의 끝이 나올 때 까지 fgets를 통하여 FILE 구조체를 통해 파일 입출력 스트림에서 문자열을 읽어 온다. 읽어온 문자열은 sentence[] 배열에 저장한다. strlen을 통하여 파일의 문자열의 길이를 length 에 저장시킨다.

```
    int thread1_size = length / 3;
    int thread2_size = thread1_size * 2;
    int thread3_size = length;

    fseek(fp, thread1_size, SEEK_SET);
    while (fgetc(fp) != ' ')
    {
    }
    thread1_size = ftell(fp);

    fseek(fp, thread2_size, SEEK_SET);
    while (fgetc(fp) != ' ')
    {
    }
    thread2_size = ftell(fp);

    fclose(fp);
```

thread_size를 통하여 세 개의 쓰레드가 작업을 수행할 범위를 구하기 위해 length를 삼등분 합니다. 하지만 예를 들어 텍스트가 “abc defghi qwer” 이렇게 있다면 3등분 시키면 스레드 1에는 "abc d", 스레드 2 에는 "efghi", 스레드3 에는 “qwer" 가 되어 단어의 맨 앞 알파벳을 정확하게 확인하기가 어려워진다. 따라서 fseek와, ftell을 이용하여 위치를 조정하였다.
fseek를 통해서 스트림 위치 지정자의 위치를 조정한다.

fseek(파일 스트림, 끝 부분 까지의 거리 , 출발점)

while문을 통하여 fgetc 로 스트림을 읽어 들이고 공백이 나올 때까지 지속한다. 범위 끝 부분에 공백이 나오면 while문을 종료하여 자신의 현재 위치를 알려주는 ftell()을 통하여 자신의 위치, 즉 조정된 범위를 스레드의 길이로 저장한다.

```
for (int i = 0; i < 26; i++)
    {
        pthread_mutex_init(&mutex[i], NULL);
    }
    thread thread1_com = {0, thread1_size};
    pthread_create(&thread1, NULL, count, &thread1_com);
    thread thread2_com = {thread1_size, thread2_size};
    pthread_create(&thread2, NULL, count, &thread2_com);
```

```
    thread thread3_com = {thread2_size, thread3_size};
    pthread_create(&thread3, NULL, count, &thread3_com);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    for (int i = 0; i < 26; i++)
    {
        pthread_mutex_destroy(&mutex[i]);
    }
```

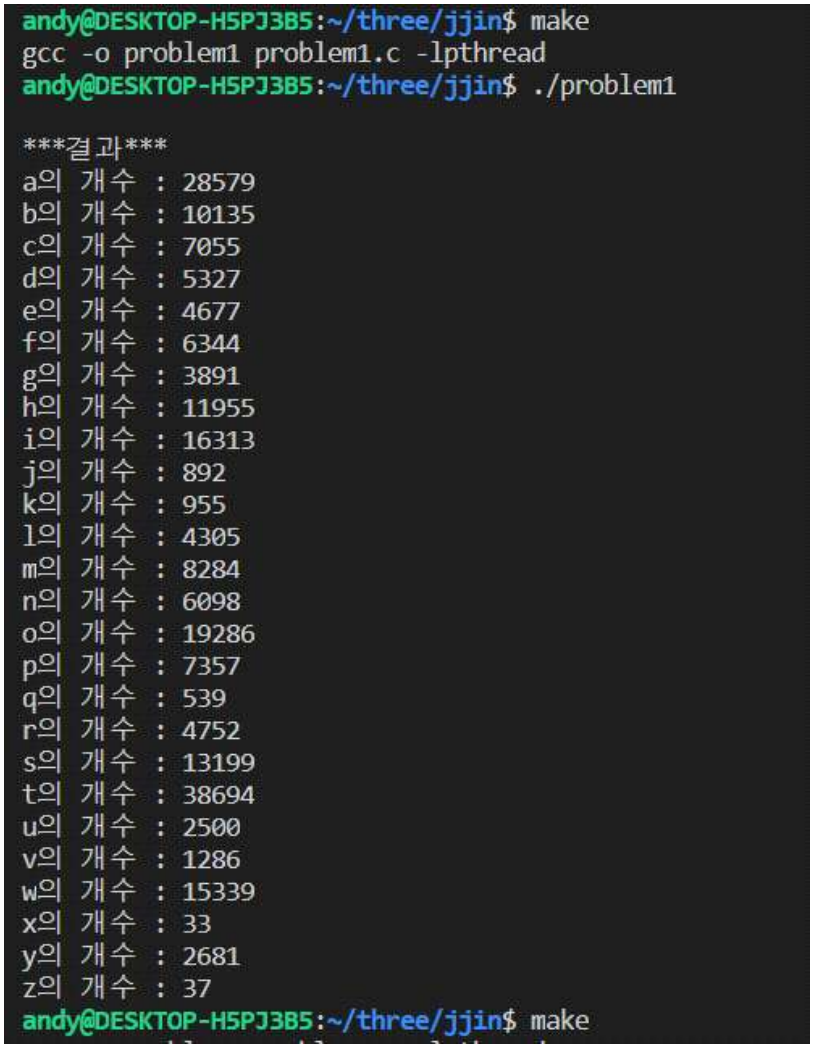
mutex는 a부터 z를 저장할 때마다 각각 스레드의 접근을 제한하도록 26개의 mutex를 생성하였고, mutex_init을 통하여 초기화한다. 스레드 구조체에 스레드가 작업을 수행할 범위를 저장하고 스레드를 생성하고 count() 함수를 실행시킨다.
pthread_create(&thread변수 , NULL, 실행할 함수, 인자)를 통해 스레드가 생성되며 순차적으로 세 개의 스레드가 생성되기 때문에 병렬적으로 처리할 수 있다. 작업을 마친 스레드는 join을 통해 값을 반환, mutex_destroy를 통해 mutex를 모두 폐기한다.

```
void *count(void *args)
{
    thread arg = *(thread *)args;
    for (int c = arg.start; c < arg.end; c++)
    {
        if (c == 0)
        {
            if (sentence[c] != ' ')
            {
                pthread_mutex_lock(&mutex[sentence[c] - 'a']);
                result[sentence[c] - 'a']++;
                pthread_mutex_unlock(&mutex[sentence[c] - 'a']);
            }
            else if (sentence[c] == ' ')
            {
                continue;
            }
        }
        else
```

```
        {
            if (sentence[c - 1] == ' ' && sentence[c] != ' ')
            {
                pthread_mutex_lock(&mutex[sentence[c] - 'a']);
                result[sentence[c] - 'a']++;
                pthread_mutex_unlock(&mutex[sentence[c] - 'a']);
            }
        }
    }
}
```

count() 함수에 대한 설명이다. for 문을 각 스레드에 저장된 범위만큼 실행시킨다. c번째 일 때, sentence[c]은 문자이고, c-1 번째 즉 sentence[c-1]이 공백이면 이것은 단어의 시작을 의미하는 것이기 때문에 result[sentence[c]-'a']에 +1한다. 만약 텍스트의 맨 처음일 경우 c-1 번째가 없기 때문에 따로 if 분기를 나눠서 수행한다. mutex_lock을 통해 만약 첫 문자가 'a' 일 경우 mutex[a]를 가지는 다른 프로세스의 진입을 막고, 작업을 마치면 mutex_unlock을 통해 다른 프로세스의 접근을 허용한다.

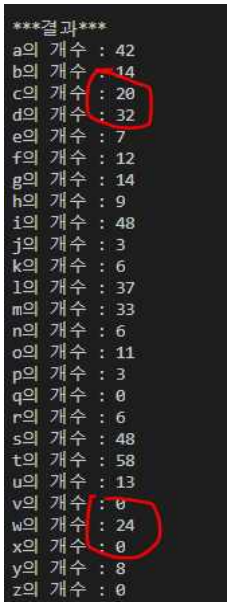
```
andy@DESKTOP-H5PJ3B5:~/three/jjin$ make
gcc -o problem1 problem1.c -lpthread
andy@DESKTOP-H5PJ3B5:~/three/jjin$ ./problem1

***결과***
a의 개수 : 42
b의 개수 : 14
c의 개수 : 21
d의 개수 : 32
e의 개수 : 7
f의 개수 : 12
g의 개수 : 14
h의 개수 : 9
i의 개수 : 48
j의 개수 : 3
k의 개수 : 6
l의 개수 : 37
m의 개수 : 33
n의 개수 : 6
o의 개수 : 11
p의 개수 : 3
q의 개수 : 0
r의 개수 : 6
s의 개수 : 48
t의 개수 : 58
u의 개수 : 13
v의 개수 : 0
w의 개수 : 24
x의 개수 : 0
y의 개수 : 8
z의 개수 : 0
andy@DESKTOP-H5PJ3B5:~/three/jjin$
```

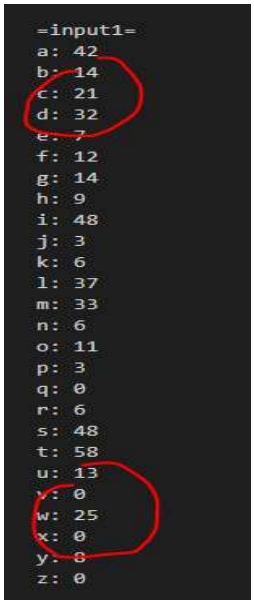


input.txt와 input2.txt를 이용해 프로그램을 실행시켰을 때의 결과이다.

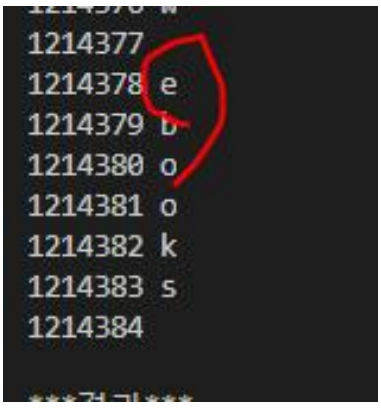
1-3 개발 HISTORY 및 소감

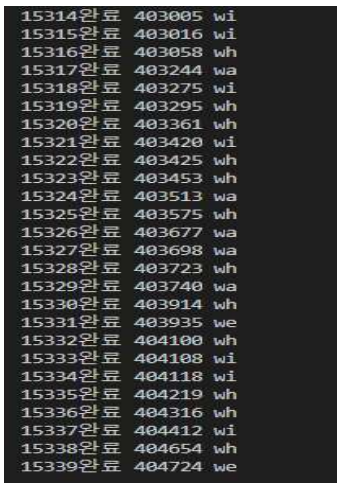
1. 처음에는 mutex를 한 개만 생성하여 코드를 작성하였다. 하지만 mutex가 한 개 있다면 모든 result[]++ 작업을 수행할 때마다 프로세스의 접근이 제한되기 때문에 알파벳 별로 나누어서 알파벳 개수만큼의 mutex를 만들어 각 알파벳에 대한 접근에 제한을 걸어둔다면 효율이 좋아질 것이라고 생각하여 26개의 mutex를 생성하였다.





2. 위의 사진을 보면 친구들과 답을 비교해 보았을 때의 결과이다. 첫 번째 것이 내 것이고, 두 번째 것이 친구의 것이다. 카운팅 하는 데에 오류가 있다는 것을 알고, 두 부분의 연관성이 무엇일까 생각을 한 결과 처음 단어와 끝 단어의 첫 문자라는 것을 알 수 있었다. 실제로 코드를 보니 위에서 설명했던 count() 함수의 if(c==0)일 때를 작성하지 않아 맨 처음 단어가 카운팅 되지 않았다. 하지만 맨 끝 단어는 도무지 무엇이 잘못되었는지 찾을 수 없었다..





위 사진을 보면 input2.txt의 맨 끝 단어인 ebook도 제대로 읽어오는 것을 확인할 수 있다. 수 많은 디버깅을 하다가 직접 input.txt의 맨 끝 단어의 이니셜인 w를 모두 세어 보았고, 24개인 것을 확인하여 친구의 실수라는 것을 알 수 있었다. 동시에 디버깅을 제대로 하지 않으면 오류 수정이 힘들기 때문에 디버깅의 중요성도 크게 깨달았다.

3. 과제 1을 하면서 아쉬운 점이 있다면 다음과 같이 파일 입출력을 통해 텍스트를 불러 읽어오는 배열을 설정한 것이다.

```
#define MAX 999999999
char sentence[MAX];
```

999999999개의 배열의 크기를 가지는 sentence이기 때문에 만약 텍스트의 길이가 999999999개를 초과할 경우 오류가 발생을 하게 된다. 하지만 인터넷의 '글자 수 세기' 사이트에서 input2.txt를 실행시켜본 결과 수초가 걸린다는 것을 확인했다. 반면 내가 작성한 코드는 훨씬 빠른 속도로 알파벳 분류까지 하는 것을 확인하였다. 물론 이것이 자바스크립트와 c언어의 차이일 수 있겠지만, 배열을 위와 같이 설정한다고 해서 크게 문제가 되지는 않을 것이라 생각했다.

또한 main 함수에서 파일 입출력 시 텍스트 파일을 읽고, count() 함수에서 for문을 통해 텍스트를 다시 한 번 읽기 때문에 같은 문장을 두 번 읽게 된다. 만약 텍스트가 매우 길어진다면 시간도 배가 될 것이다. 따라서 count 함수에서 필요한 크기만큼의 텍스트만을 불러와서 작업을 수행한다면 이상적일 것이라고 생각하였다. 하지만 방법이 딱히 떠오르지 않아 성공하지는 못하였다. 기회가 된다면 이 부분을 해결해 봐야겠다.

2-1 임계구역 문제

공유 데이터를 사용하기 위해 경쟁하는 n 개의 프로세스들을 가정하면 임계영역이란 각프로세스에서 공유 데이터를 조작하는 명령을 포함하는 코드 영역을 가리킨다. 즉 공유 데이터를 둘이상의 프로세스가 동시에 접근하면 안 되는 것이다. 한 프로세스가 임계 영역을 수행 중이면 다른 프로세스는 자신의 임계 영역의 코드를 수행할 수 없어야 한다. 이것을 임계 구역 문제라고 한다.

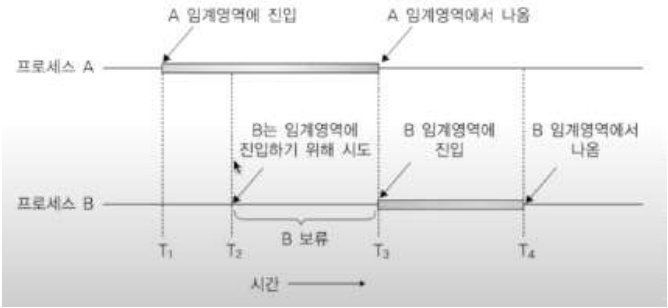
구조는 다음과 같다.

do{

진입영역 -> 임계 영역으로 진입 허가 요청 코드
critical 영역-> 순서대로 한 번의 프로세스가 실행
출구 영역-> 임계 영역 탈출 시 권한 반납 코드

}while

세 가지 조건을 충족해야 문제가 발생하지 않는다. 세 가지 조건은 진입 상호 배제, 진행, 한계 대기가 있다. 아래 그림은 상호배제와 관련된 그림이다.

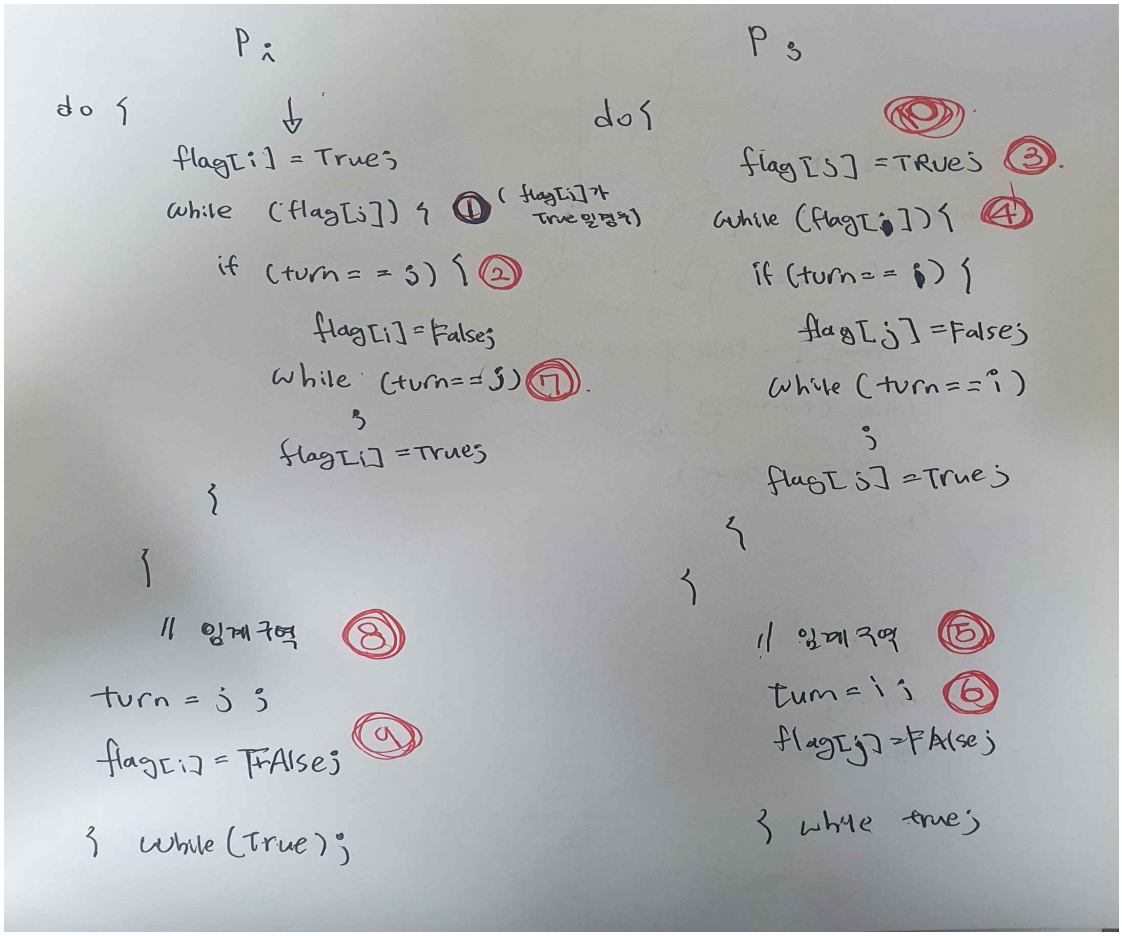


위의 그림은 상호배제와 관련된 그림이다.A가 임계구역을 접근 중일 때 B가 임계 구역에 접근하지 못한다는 것을 나타낸다. 이처럼 상호배제 조건 만족을 위해서는 임계 구역에 하나의 프로세스만이 접근 가능해야한다는 것을 알 수 있다. 상호배제가 임계구역에 접근 중인 프로세스가 있을 때 해당하는 문제라면 진행은 임계구역에 접근 중인 프로세스가 없을 때 언제든지 프로세스가 접근할 수 있게 허용해 줘야한다는 것이다. 마지막으로 한계 대기는 임의의 프로세스가 자신의 임계 영역에 진입 요청을 한 후 허가되기 전까지 다른 프로세스들이 자신의 임계 영역에 접근하는 회수에 한계가 있어야한다는 것이다. 즉 어떤 프로세스도 임계 영역에 들어가는 것이 무한정 연기, 기아상태, 교착상태가 발생하면 안 된다는 것이다.

2-2 동기화 문제 해결 증명

```
do {
        flag[i] = TRUE;

        while (flag[j]) {
              if (turn == j) {
                    flag[i] = FALSE;
                    while (turn == j)
                          ; // do nothing
                    flag[i] = TRUE;
              }
        }

        // CRITICAL SECTION

        turn = j;
        flag[i] = FALSE;

        // REMAINDER SECTION
} while (TRUE);
```

위의 코드는 동기화 문제를 해결해주는 코드이다. 위에서 설명했듯이 상호배제, 진행, 한계 대기 3가지 조건을 만족해야 한다. turn은 현재 수행할 차례의 프로세스를 나타내고 flag는 각 프로세스가 수행을 가능한 지를 나타낸다.



각 프로세스에서 함수가 코드가 실행되는 순서를 번호로 써보았다. 만약 flag[j]가 TRUE 이고 j의 turn 일 조건에서 프로세스 I의 코드가 실행이 될 경우 flag[i] 가 TRUE 가 된다.

(위 사진 속 번호의 의미)

1. flag[j]가 참일 동안 while 문이 수행된다. 현재 True이므로 실행이 된다.
2. 만약 turn==j일 경우 flag[i]=false 가 된다. 실제로 그렇기 때문에 수행이 된다. 이후 turn==j 일 동안 while 문이 실행되기 때문에 프로세스 I는 while 문 안에 갇히게 된다.
3. 프로세스 j 가 진행된다.
4. flag[i]가 참일 경우 while 문이 실행되므로 현재 flag[i]가 false 이기 때문에 while문은 실행되지 않는다.
5. 프로세스 j가 임계구역을 수행한다.
6. tun==i, I가 수행할 차례로 바꾼다. 자신의 flag를 false로 바꾼다.
7. 프로세스 i가 turn 이 j가 아니기 때문에 while문 안에서 빠져나온다.
8. 프로세스 I가 임계구역을 수행한다.
9. 프로세스 I 가 자신의 flag를 false로 바꾸고 turn을 j로 바꾼다.
10. 프로세스 j가 다시 실행된다. flag[j]== true 이고 turn ==j, flag[i]==false 이기 때문에 3,4,5,6 과정을 진행하고 프로세스 I 도 프로세스 j와 같은 과정을 진행한다.

방금 조건은 flag[j]=True 일 때였지만 만약 flag[i]가 false 이고 turn 이 I 였다면 프로세스 I 가 멈춤 없이 실행될 것이고 임계구역을 수행중이기 때문에 프로세스 j 가 잠심 while 문에

갇혀 있게 될 것이다.

위의 임계구역 문제 해결 방법은 flag와 turn을 두어 만약 상대가 임계구역에 접근하는 중이라면 while 문에 머무르게 하여 임계 구역의 진행을 막았다. 이를 통해서 하나의 프로세스가 임계구역에 있을 때 다른 프로세스가 임계구역에 접근을 못하기 때문에 상호배제를 만족한다는 것을 알 수 있다. 또한 임계구역에서 수행을 마친 프로세스는 마지막에 자신의 flag를 false, turn을 상대 프로세스의 turn으로 바꾸어 주기 때문에 임계구역에 아무도 없을 때, 언제든지 접근할 수 있다. 이것으로 진행의 조건도 만족한다는 것을 알 수 있다. 또한 상대의 flag==true 일 때는 while 문 안에 머무르다가 상대의 수행이 끝나면 상대의 flag=flase 로 바뀌기 때문에 while 문 안에서 무한으로 대기하는 문제가 발생하지도 않게 된다. 따라서 마지막 조건인 한계 대기도 만족시키게 된다.