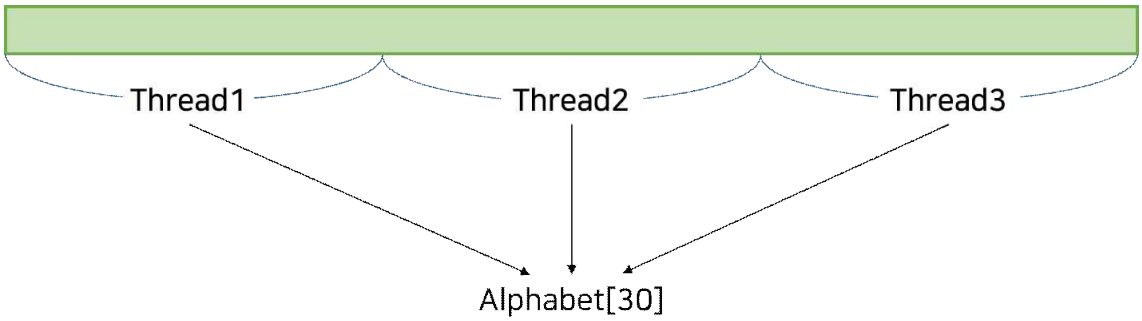
운영체제 과제4-1

201921166 정의철

1. 설계 개요



스레드 3개가 각각 파일의 1/3씩 구역을 할당받아, 단어의 첫 글자를 result 배열에 반영하도록 하였습니다. 파일의 크기를 N이라고 했을 때, Thread1은 (0, N/3), Thread2는 (N/3, N*2/3), Thread3는 (N*2/3, N)의 구역을 할당받습니다.

N/3 또는 N*2/3의 구역이 단어의 사이일 경우, 공백이 되도록 구역을 재조정합니다.

result[30] 배열의 result[i], result[i+1], · · · 은 각각 독립적인 공간이므로 배열의 공간 수에 맞추어 mutex도 30개 선언하였습니다. 만약 mutex가 1개 밖에 없다면, Thread가 3여도 알파벳 카운트는 1개 thread밖에 진행하지 못하므로 single thread 보다도 못한 성능을 낼 것입니다.

이후 pthread_create() 함수를 통해 thread를 3개 생성하고, join하여 작업이 끝나기를 기다린 뒤에 알파벳 별로 결과값을 출력합니다.

2. 코드를 이용한 각 기능에 대한 설명

가. 파일의 길이(N) 찾기

```c
FILE *input_fp = fopen("./input1.txt", "r");
fseek(input_fp, 0, SEEK_END);
int file_size = ftell(input_fp);
```

파일의 길이는 fseek와 ftell을 통해 찾습니다.

feesk(input_fp, 0, SEEK_END); 구문은 input_fp의 파일 커서를 파일의 맨 끝으로 보내고, ftell(input_fp); 를 통해 현재 input_fp가 가리키고 있는 파일 커서의 위치를 반환하므로 파일의 총 길이를 알 수 있습니다.

나. 파일 커서를 공백으로 밀어내기

각 thread별로 부여 받은 구역. 예를 들어 (0, N/3)에서 N/3 부분이 문자를 가리키고 있을 때, 공백을 가리키도록 파일 커서를 밀어내는 작업입니다.

```c
fseek(input_fp, thread_offset_arr[1], SEEK_SET);
while (fgetc(input_fp) != 0x20)
{
    thread_offset_arr[1]++;
}
```

fseek를 통해 input_fp의 커서 위치를 thread_offset_arr[1](= N/3)으로 보냅니다. 그 위치가 공백(=0x20)이 아니라면, 커서가 공백을 가리킬 때 까지 thread_offset_arr을 1식 증가시키는 방법을 사용하였습니다.

다. N이 공백을 가리키지 않게 당겨오기

Thread3이 맡게 되는 구역은 (N*2/3, N)입니다. 이 때, N이 공백을 가리키고 있다면 fscanf의 특성 때문에 파일 마지막에 위치한 단어가 중복되어 입력됩니다. 예를 들어 파일의 마지막 문자열이 “woah ”처럼 N의 위치에 ‘h’가 아닌, 공백이 있을 경우에는 중복 문제가 발생합니다.

```c
size_t end_point_cursor = file_size - 1;
fseek(input_fp, end_point_cursor, SEEK_SET);

while (fgetc(input_fp) == 0x20) {
    fseek(input_fp, --end_point_cursor, SEEK_SET);
}

thread_offset_arr[THREAD_NUM] = end_point_cursor;
```

따라서 N이 공백이 아닌, 'h'를 가리킬 수 있도록 해주는 연산을 수행합니다.

라. 스레드 생성 및 thread_arg 구조체

```
typedef struct _thread_arg
{
    size_t origin;
    size_t end;
} thread_arg;
```

스레드를 생성할 때에 void * 타입의 데이터만 argument로 전달 가능합니다. 하지만, 스레드 함수에 넘겨주어야 하는 값은 origin(처리할 시작점)과 end(처리할 끝점) 두 개 이므로, 이를 thread_arg 구조체로 넘겨줄 수 있도록 했습니다.

```
for (size_t i = 0; i < THREAD_NUM; i++)
{
    thread_arg *temp = malloc(sizeof(thread_arg));
    temp→origin = thread_offset_arr[i];
    temp→end = thread_offset_arr[i + 1];
    pthread_create(&pid_arr[i], NULL, count_runner, (void
*)temp);
}
```

스레드에 넘겨주어야 할 인자를 thread_arg 구조체로 래핑하고, pthread_create() 함수를 통해 스레드를 생성해 주었습니다.

pthread_create() 함수에서 두 번째 자리에 들어가는 인자는 const pthread_attr_t *__restrict__ __attr 인데, 따로 지정해줄 attribute는 없으므로 NULL을 넘겨주도록 하였습니다.

마. 병렬 처리 및 동기화

```c
void *count_runner(void *arg)
{
    thread_arg t_arg = *(thread_arg *)arg;
    char temp[50];
    FILE *fp = fopen(FILE_PATH, "r");

    fseek(fp, t_arg.origin, SEEK_SET);

    while (ftell(fp) < t_arg.end)
    {
        fscanf(fp, "%s", temp);
        save_alphabet(temp[0]);
    }

    fclose(fp);
    free(arg);
}

void save_alphabet(char c)
{
    pthread_mutex_lock(&mutex_arr[c - 'a']);
    result[c - 'a']++;
    pthread_mutex_unlock(&mutex_arr[c - 'a']);
}
```

void *count_runner(void *arg) 함수는 pthread의 start routine으로서, 스레드 3개에 의해서 병렬 실행됩니다. 텍스트 파일을 open 한 뒤에, 파일 커서를 orgin으로 옮기고 end에 도달할 때 까지 계속해서 문자열을 읽어 들입니다. fscanf는 공백의 개수에 상관없이 공백이 하나 이상 존재할 경우에 무조건 chunking 하므로, temp[0]은 계속해서 단어의 앞 글자를 담게 되어있습니다. save_alphabet을 호출하여 temp[0]을 넘기면, 해당 알파벳에 해당하는 mutex를 lock하고 (++) 증가연산을 한 뒤에 unlock합니다.

3. 개발 history 및 소감

이번 과제에서는 스레드를 통한 병렬처리와 동기화 기법을 사용하는 것이 포인트였습니다.

스레드는 생성과 회수에 생각보다 큰 리소스가 필요합니다. 따라서 잘못 구현하면 오히려 싱글 스레드 보다 성능이 떨어지는 결과를 가져올 수도 있습니다. 이를 명심

하여 최적화를 해보려 노력했던 것 같습니다. 먼저, file read를 각 스레드에서 처리하도록 하였습니다. 구현의 편의를 위해서 file 내용을 모두 읽어와 이를 전역변수에 담고, 각각의 스레드에서 전역변수를 통해 데이터를 처리하는 방법은 메모리 공간적인 측면에서나, 퍼포먼스 측면에서나 싱글 스레드에 비해 나은 점이 없고, 오히려 싱글 스레드에 비해 시간복잡도가 2배나 더 컸습니다. 따라서 각 스레드에서 파일을 직접 여는 방법을 채택했습니다.

그리고 mutex를 26개 선언하였습니다. critical section에서 처리해야할 데이터의 종류는 int result[26]. 즉 int형 데이터 26개 이므로 mutex가 26개 있어야 동기화 퍼포먼스를 끌어올릴 수 있습니다. 만약 mutex가 한 개만 선언되었다면, 서로 영향이 없는 데이터를 동시에 저장할 때에도 mutex에 의해 waiting이 발생하므로 싱글 스레드 프로그램과 비슷하거나 오히려 더 못한 퍼포먼스를 낼 것입니다. 따라서 mutex를 26개 선언하는 방식을 채택했습니다.

기존에 자바 스레드를 사용하는 방법만 배워보았지, POSIX Thread는 처음 사용해보았습니다. 자바가 C 보다 고수준 언어여서, 자바 스레드를 사용할 때에는 사용자 입장에서 신경 쓸 것이 별로 없고, 자바8 부터는 람다식을 이용해서 한 줄만으로도 스레드를 생성할 수 있습니다. 하지만 C로 스레드를 사용해보면서, Low level에서의 스레드 작동을 이해할 수 있어 좋은 경험이 되었습니다. 또한 mutex lock, unlock 부분을 빼고 실행하니 결과 값이 다르게 나오는 것을 실습해보아 동기화기법의 중요성을 깨달을 수 있었습니다.

운영체제 과제4-2

1. 임계구역 문제 해결 3가지 요건

임계구역 내의 코드가 유효한 동작을 하기 위해서는 다음 3가지 요건을 만족해야 한다.

가. Mutual Exclusion

하나의 프로세스가 임계 구역에 들어가 있다면 다른 프로세스는 들어갈 수 없어야 한다.

나. Progress

임계 구역에 들어간 프로세스가 없는 상태에서, 들어가려고 하는 프로세스가 여러 개 있다면 어느 것이 들어갈지를 적절히 결정해주어야 한다.

다. Bounded Waiting

다른 프로세스의 기아를 방지하기 위해, 한 번 임계 구역에 들어간 프로세스는 다음 번 임계 구역에 들어갈 때 제한을 두어야 한다.

2. 문제 해결 요건 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);
```

프로세스 P_i의 구조

가. Mutual Exclusion

위의 코드에서 turn = j일 경우, P_i는 while문을 통해 busy waiting을 하므로, 임계 구역에 접근할 수 없다. turn 변수는 오직 임계 구역의 실행이 끝난 뒤에만 값이 재 할당되므로, 하나의 프로세스가 임계 구역에 들어가 있다면 다른 프로세스는 들어

갈 수 없고, while문에서 busy waiting을 하게 된다. 따라서 Mutual Exclusion 요건을 만족한다.

나. Progress

flag[i]와 flag[j]가 둘 다 TRUE인 경우, turn 변수에 의해서 임계 구역에 진입할 프로세스가 정해진다. 따라서 P_i와 P_j 두 프로세스가 모두 임계 구역 진입을 원하는 경우, turn 변수에 의해서 임계 구역에 진입할 프로세스가 결정 되고, 임계 구역에 진입하지 못하는 프로세스는 flag를 FALSE로 변경하여 상대 프로세스가 임계 구역에 진입하도록 하고 자신은 busy waiting 한다. 임계 구역에 들어간 프로세스가 없는 상태에서, 들어가려고 하는 프로세스가 여려 개 있다면 turn 변수에 의해서 어느 것이 들어갈지가 적절히 결정되므로, Progress 요건을 만족한다.

다. Bounded Waiting

한 프로세스가 임계 구역을 벗어나게 되면 turn 변수가 자신 이외의 프로세스를 가리키게 하므로, 기아가 발생하지 않는다. 따라서 다른 프로세스의 기아를 방지하기 위해, 한 번 임계 구역에 들어간 프로세스는 다음 번 임계 구역에 들어갈 때 제한이 있으므로, Bounded Waiting 요건을 만족한다.

위의 Dekker's algorithm은 임계 구역 문제 해결 3가지 요건을 만족하므로, 동기화 기능을 충분히 수행해 낼 수 있다.