

Multi-Thread Programming: Character Counter

차 호 현(#32224560), outcider112@dankook.ac.kr

Undergraduate Student in Moble Systems Engineering, Dankook University

Index

1. 서론: 스레드란 무엇인가?
2. 병렬 프로그래밍의 한계점과 개선
3. 프로젝트 빌드 방법
4. 프로젝트 빌드 버전 (1, 2, 3)
5. 실험 결과
6. 결론

Free day: 2 day left

스레드란 무엇인가?

운영체제는 하나의 물리적인 CPU를 다수의 가상 CPU로 확장하여 마치 여러 개의 프로그램이 동시에 실행하는 듯한 착시를 만든다. 운영체제 상에서 동작 중인 프로그램을 프로세스라고 말하며, 이러한 프로세스는 CPU 타임과 메모리 주소 공간, 그리고 시스템 콜을 통해 하나의 가상 컴퓨터 상에서 프로그램이 동작하는 것처럼 보이게 한다.

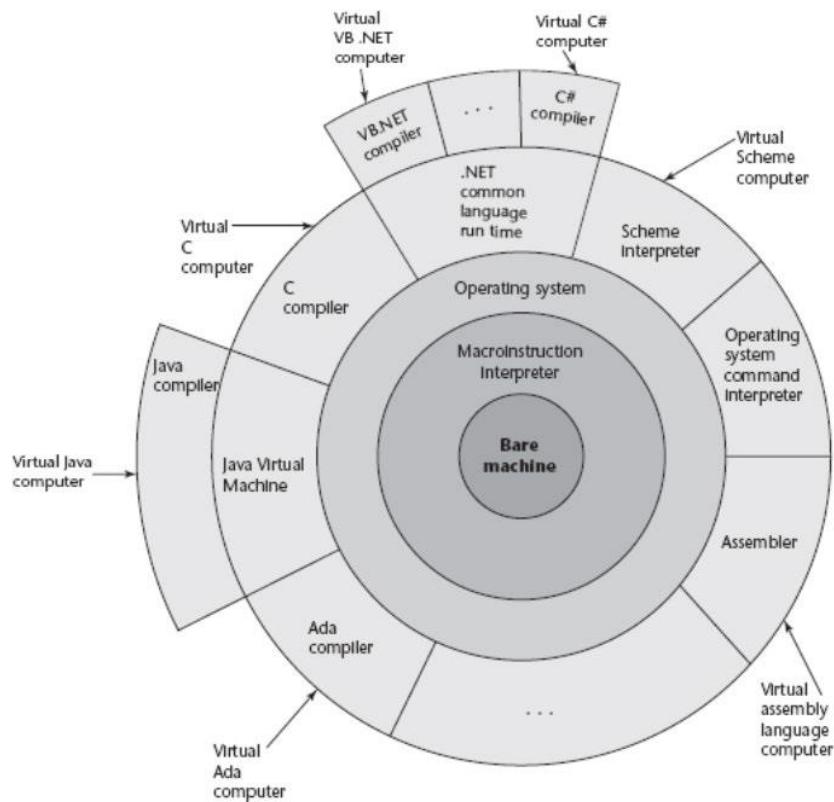


그림 1. 프로그래밍 언어 별 계층적 인터페이스¹

하나의 프로그램은 실행 모듈(흔히 부르는 “프로그램”)을 저장 장치로부터 로드하고, 프로세스가 실행 시간에 들어가면, 프로세서의 PC 값을 프로그램의 실행 첫 주소로 초기화 후 인출-실행 사이클을 시작한다. OS는 이러한 프로세스에게 CPU 시간을 제공하여, 하나의 CPU를 가지고 여러 프로그램 흐름을 처리하는 것처럼 보이게 한다.

¹ 프로그래밍 언어론 제 12판, 하상호 역

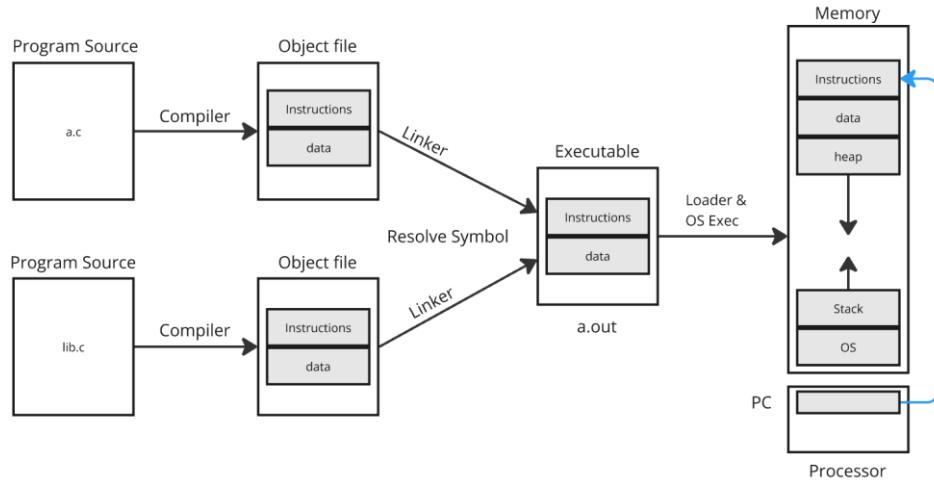


그림 2. 프로그램이 머신에 로드되는 과정

하지만, 프로세스를 이용한 병렬 처리에는 두 가지 한계점이 존재한다. 프로세스를 통한 병렬 처리의 한계점은 각 프로세스 간에 고유한 메모리 공간을 할당 받는 것이다. 프로세스는 실행 모듈을 로드할 때, 실행 모듈의 elf 포맷을 읽으면서 프로세스 하나가 사용할 메모리 레이아웃을 생성한다. 이러한 메모리 주소 공간을 벗어나서 프로세스가 정보를 교환하기 위해서는, 프로세스 간의 통신 (IPC)를 구현해야 하므로 메모리를 통해 정보를 교환하는 것보다 높은 오버헤드를 가질 수 있다.

따라서, 멀티-프로세스 컴퓨팅이 아닌, 상대적으로 경량인 스레드(Thread)를 이용하여 손쉬운 정보 교환과 상대적으로 적은 메모리를 바탕으로 동작할 수 있다.

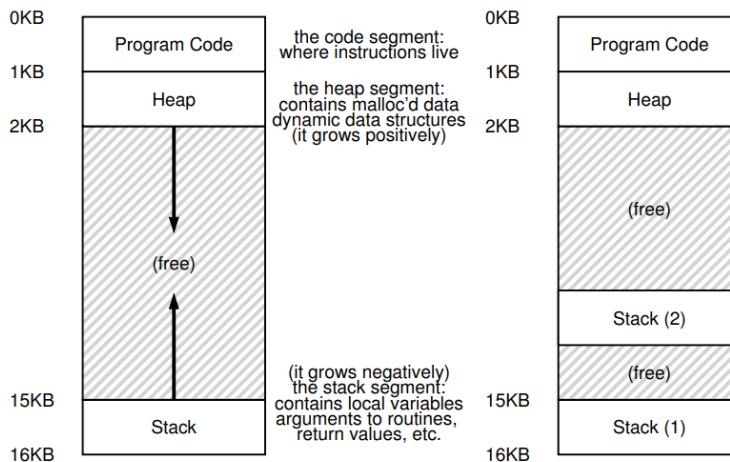


그림 3. 멀티스레드 프로그램의 메모리 레이아웃²

^{2, 3} 운영체제 아주 쉬운 세가지 이야기 제 2 판, 원유집 역,

스레드는 프로세스의 상태와 유사하여, 스레드만의 PC 와 레지스터 파일들을 가진다. 이에 따라서, 두 개의 스레드가 하나의 프로세서(CPU)에서 실행 중이면 문맥 교환을 통해서 실행하고자 하는 스레드를 교체한다. 이에 문맥 교환 시 가지고 있는 레지스터 상태를 저장하기 위해 프로세스가 PCB(Process Control Block)을 사용하듯이, 스레드의 상태를 저장하기 위해서 TCB(Thread Control Block)이 필요하다.

앞서 언급하였듯, 프로세스는 고유한 주소 공간을 할당 받는다. 하지만 멀티 스레드 프로그램은 멀티 프로세스와 달리 스레드 각각이 고유한 주소 공간을 가지지 않고 하나의 공유되는 주소 공간을 가진다. 그리고 각 스레드는 앞서 언급하였던 TCB 와 스택을 가져, 스레드가 독립적으로 실행되면서 필요한 여러 루틴을 호출할 수 있다.

다시 말해 최종적으로 스레드는 하나의 메모리 공간을 공유하면서도, 각각의 스레드는 스케줄이 되어 CPU에서 프로세스처럼 실행될 수 있는 단위를 의미한다.

스레드를 이용하여 일을 수 있는 이점은 두 가지로, 멀티 프로세서 상에서 병렬 처리를 수행하여 높은 처리량을 달성하기 위한 목적이 첫번째, IO 와 같은 상대적으로 느린 작업 혹은 Blocking 으로 인한 프로그램 처리 중단이 발생하는 동안, 다른 프로세스의 처리를 수행하여 CPU 를 효율적으로 사용하기 위함이다.³

실제로 단일 CPU 상에서의 다중 스레드 프로그램을 동작시키는 것은 실제로 동시성을 달성한 병렬 처리가 아니다. 가령, Job A 와 B 가 있을 때 CPU 타임 T를 T1 과 T2 로 할당하여 실행하는 것은 마치 두 스레드가 동시에 실행되는 것처럼 보일 수 있지만, 실제 처리량은 단일 CPU 만큼만 일을 수행하거나 문맥 교환으로 인한 CPU 시간을 일부 점유하여 오히려 단일 코어 상에서 다중 스레드 프로그램은 효율적이지 않다.

반대로 다중 CPU 상에서는 두 개의 스레드가 물리적으로 다른 프로세서에서 동작하여 CPU 1 과 2 의 CPU 시간을 모두 사용할 수 있다. 이에 따라 문맥 교환의 비용까지 감안하여 약 2 배의 성능 향상을 모색할 수 있게 되는 것이다.

병렬 프로그래밍의 한계점과 동기화

스레드를 통해서 하나의 메모리 주소 공간 내에서 여러 실행 흐름을 가지게 된다. 각각의 실행 흐름은 TCB를 통해서 스케줄러의 의해 관리된다. 하지만, 운영체제의 스케줄링으로 인하여 공유 자원 접근 문제가 발생한다.

운영체제의 스케줄러는 프로그램의 실행 중, 언제 어떤 구문을 실행하다가 스케줄러로 인하여 문맥 교환이 발생될지 모른다. 이러한 상황에서 두 개의 스레드가 공유되는 메모리에 접근한다고 가정한다.

```
void *my_thread(void* arg){  
    printf("%s\n", (char*) arg);  
}  
  
void main(){  
    pthread_t p1, p2;  
  
    int rc;  
  
    printf("main");  
  
    pthread_create(&p1, NULL, my_thread, "A");  
    pthread_create(&p1, NULL, my_thread, "B");  
  
    pthread_join(p1, NULL);  
    pthread_join(p2, NULL);  
}
```

다음과 같이 스레드 프로그램을 작성하였을 때, `pthread_create`를 통해 스레드를 생성한다. 스레드가 생성된 후에는 운영체제의 스케줄링 큐에 존재하며, 스케줄러에게 프로그램 처리가 될 때까지 대기 상태에 진입한다. 운영체제의 스케줄러는 언제 해당 프로그램을 처리할지, 실행 중 언제 스케줄 아웃될지 알 수 없다고 가정한다. 이러한 특징에 따라서, 프로그램에 따라 스레드 A를 먼저 호출하였으나, 스레드 B가 먼저 `printf` 구문을 처리할 수도 있다. 또한, 현대의 CPU는 단일 프로세서 코어를 통한 성능 향상의 한계를 맞이하여, CPU 내에 다중 프로세스 코어를 통해 물리적으로 서로 다른 CPU 상에서 프로그램을 실행할 수 있다.

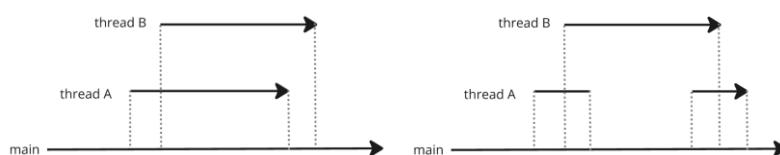


그림 4. 이상적인 실행 흐름과 스케줄러로 인해 스레드 A가 늦게 처리되는 흐름

또한 스케줄링 순서가 다른 문제에 함께, 공유 자원에 대해서 경쟁이 발생하는 문제가 발생한다. 기계어 단계에서, 두 개의 실행 흐름이 있다고 가정한다.

X 의 주소 = \$sp + 0

x += 1 x += 2

$(\$sp + 0) = 100$

lw \$t0, 0(\$sp)	lw \$t0, 0(\$sp)
addi \$t1, \$t0, 1	addi \$t1, \$t0, 2
sw \$t1, 0(\$sp)	sw \$t1, 0(\$sp)

이상적인 처리 결과는 $x += 1$ 를 수행하는 스레드로 인하여 101, $x += 2$ 를 수행하는 스레드로 인하여 103이다. (역 또한 같다)

하지만, 소스 코드와 달리 기계어는 하나의 소스 코드에 대해서 여러 기계 명령어에 걸쳐서 실행된다. 그런데, 이 과정에서 스레드의 처리가 다음과 같은 흐름으로 발생하였을 때는 의도한 결과 값인 103의 값이 나타나지 않는다.

lw \$t0, 0(\$sp)	
	lw \$t0, 0(\$sp)
	addi \$t1, \$t0, 2
addi \$t1, \$t0, 1	
sw \$t1, 0(\$sp)	
	sw \$t1, 0(\$sp)

이 경우, addi \$t1, \$t0, 1으로 인하여 t1의 값이 101이 되었다. 그리고 해당 변수의 주소에 101을 저장하였으나, 값 저장 이전 시점에 접근한 명령어로 인하여, 101이 아닌 102의 값이 메모리에 저장된다. 이러한 상황을 경쟁 상태 (race condition)이 된다.

따라서, 공유자원에 접근할 때, 모든 스레드가 동일한 값을 가지고 올 수 있도록 하는 것을 동기화라고 한다. 각 스레드 마다 동기화를 달성하기 위해서는 프로그램 명령어에 대해서 특정 메모리를 수정하는 코드 영역을 임계 영역이라고 정의해야 한다. 그리고 임계 영역에 대해서 동시에 처리될 수 없도록 상호 배제를 구현해야 한다.

상호 배제는 하나의 스레드가 임계 영역 내의 코드를 실행 중일 때 다른 스레드가 실행할 수 없도록 보장하는 것이다. 이러한 상호 배제는 Dekker와 Peterson 알고리즘 혹은 하드웨어 지원을 통해 구현할 수 있다.

Mutex

뮤텍스는 멀티 스레드 프로그램에서 공유 자원에 대해서 상호 배제를 구현하는 동기화 기법으로, 여러 스레드가 공유 자원에 접근하여 발생할 수 있는 경쟁 상태를 방지할 수 있다.

뮤텍스에는 두 가지 동작이 존재하는데, 락(잠금)과 언락(해제)이다. 어떠한 스레드에서 임계 영역으로 접근할 때 뮤텍스 락을 획득한다. 이 때, 다른 스레드에서 락을 획득하려고 시도하는 경우, 무한정 대기하게 된다.

```
pthread_mutex_lock(&lock); // 락을 획득하지 못하는 경우, 해당 스레드는 스케줄 아웃됨  
/* some of the code */ // 락을 획득한 경우 해당 코드 실행  
pthread_mutex_unlock(&lock);
```

이 때, 뮤텍스는 락을 얻지 못한 스레드를 스케줄 아웃하고 대기 상태로 전환한다. 이러한 구현의 정반대의 개념이 **Spinlock**으로 다음과 같이 while문을 통해 현재 락을 획득하였는지 지속적으로 검사한다. 이러한 방식은 해당 스레드에 배정된 CPU 시간을 전부 소비한다는 점에서 컴퓨팅 자원을 낭비하게 된다.

```
while (*lock == 1) {  
    // 잠금이 해제될 때까지 계속 대기  
}
```

프로젝트 빌드 방법

프로젝트는 2024-os-hw2의 32224560_hohyeon 브랜치에 존재한다.

구현된 멀티 스레드 프로그램은 cmake를 기반으로 빌드되며, makefile은 cmake를 호출하도록 한다.

```
make
```

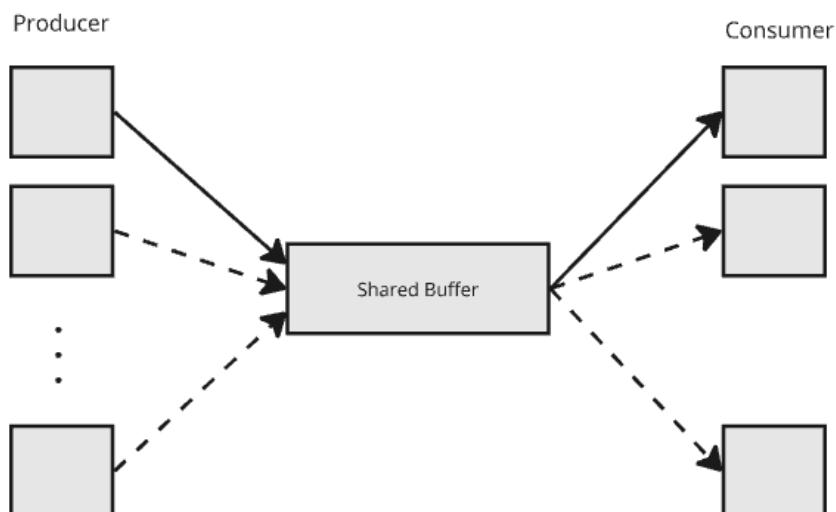
빌드가 종료되고 나면, 다음 경로에서 다음 메시지와 함께 빌드 결과를 확인할 수 있다.

```
=====
🎉 success to build
execute file location: ./build/prod_cond (single buffer)
execute file location: ./build/single_share_queue (single share queue)
execute file location: ./build/shared_queue (multi share queue)
execute file location: ./build/char_stat (char_stat : verifier)
=====
```

프로그램을 각각 테스트하기 위해서 다음과 같이 호출한다.

```
./build/prod_cond 테스트_파일 [생산자 수] [소비자 수]
./build/single_shared_queue 테스트_파일 [생산자 수] [소비자 수]
./build/multi_shared_queue 테스트_파일 [생산자 수] [소비자 수]
// multi shared queue는 반드시 생산자 수와 소비자 수를 일치 한다.
```

프로젝트 빌드 1 *(단일 버퍼 생산자-소비자)



프로그램은 단일 버퍼 (단일 포인터)에 대해서 읽고 쓸 수 있도록 프로그램을 구성하였다.

```
typedef struct sharedobject {
    FILE *rfile;
    int linenum;
    char *line;
    pthread_mutex_t lock;
    int eof;
    int full;
} so_t;
```

생성자와 소비자 간에 공유하는 변수가 위와 같을 때, 생성자의 프로그램과 소비자의 프로그

램은 다음과 같다.

생성자의 임계 구역 상하로 pthread_mutex_lock / pthread_mutex_unlock이 설정되어 있다. 생성자는 임계 구역에 진입 후, 만약 버퍼가 비어 있다면, 파일로부터 한 줄을 읽고, 버퍼를 채운다. 그러나, 생성자가 락을 획득하고 진입하였을 때, 버퍼가 채워져 있지 않은 경우, pthread_cond_wait을 통해서 컨디션 변수를 통해 신호를 받을 때까지 대기한다. 그러나, 신호를 받았을 때, 반복문을 통해서 버퍼에 대한 조건이 옳은지 확인한다.

```
void *producer(void *arg) {
    /* 프로그램 초기화 구문 */
    while (1) {
        pthread_mutex_lock(&so->lock);
        while(so->full == 1 && so->line) {
            pthread_cond_wait(&prod_cv, &so->lock);
        }

        read = getdelim(&line, &len, '\n', rfile);
        if (read == -1) {
            so->full = 1;
            so->eof = 1;
            so->line = NULL;
            pthread_cond_broadcast(&cond_cv);
            pthread_mutex_unlock(&so->lock);
            break;
        }

        // shared object job
        so->linenum = i;
        so->line = strdup(line);
        i++;
        so->full = 1;
        pthread_cond_broadcast(&cond_cv);
        pthread_mutex_unlock(&so->lock);
    }
    /* 스레드 종료 처리 */
}
```

마찬가지로 소비자 또한 버퍼가 비어 있는지 확인 후, 버퍼를 비우고 임계 구역을 벗어나서 프로그램을 처리한다. 이 때, 임계 구역을 벗어나면서 pthread_cond_wait에 cond_cv 변수에 대해 대기 중인 모든 스레드를 깨우기 위해서 pthread_cond_broadcast를 호출한다. Pthread_cond_signal이라는 함수가 존재하지만, 다중 생성자와 다중 소비자가 존재한다고 가정하는 상황에서 동작할 수 있도록 모든 스레드를 깨운다.

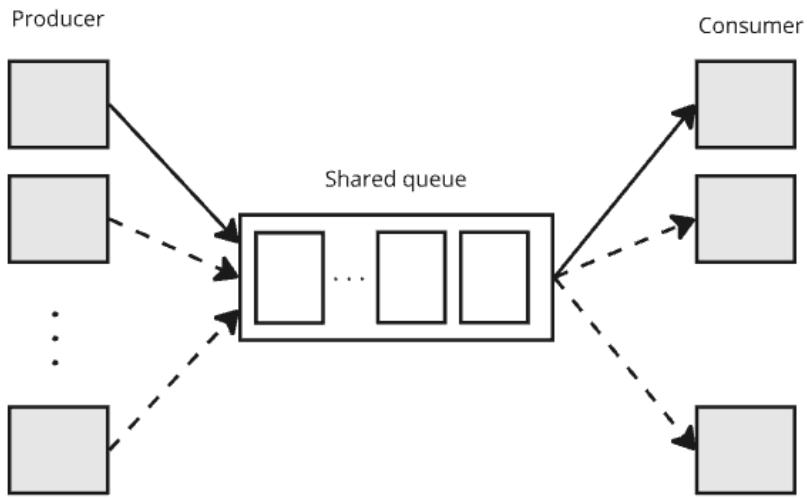
소비자에서는 get_char_stat_from_line이라는 함수를 임계 구역 외부에서 실행한다. 이는 임계 구역 내에서 처리하지 않도록 문자열 배열을 힙 영역에 선언 후, 단순히 해당 배열을 가리

키는 포인터를 전달받아 소비자의 임계 구역 외부에서 처리할 수 있도록 하여 불필요한 락의 점유를 방지한다.

```
void *consumer(void *arg) {
    /* 프로그램 초기화 구문 */
    while (1) {
        //check the condition for mutex lock and Termination condition
        pthread_mutex_lock(&so->lock);
        while(so->full == 0) {
            pthread_cond_wait(&cond_cv, &so->lock);
        }
        line = so->line;
        if (so->eof) {
            pthread_cond_broadcast(&prod_cv);
            pthread_mutex_unlock(&so->lock);
            break;
        }
        so->full = 0;
        pthread_cond_broadcast(&prod_cv);
        pthread_mutex_unlock(&so->lock);

        get_char_stat_from_line(line, local_stat, local_stat2);
        free(line);
    }
}
```

프로젝트 빌드 2 *(단일 공유 큐 생산자-소비자)



두 번째 구현은 단일 포인터 버퍼를 사용하는 대신, 포인터 큐를 구현하였다. 해당 방식은 단일 포인터 버퍼에 대해서 다중 생산자 – 다중 소비자가 접근하고자 경쟁하지만 한 번에 한 생산자-소비자만 접근할 수 있기 때문에, 실제로 락을 얻지 못하고 대기하거나, 락을 얻었지만 버퍼가 비어 있어 `wait`을 호출하게 되어 과도하게 문맥 전환을 하는 문제가 있다. 따라서 이러한 문제를 해결하기 위해, 단일 포인터 큐를 사용하여 구현하였다. 이전 코드에서의 변화점이라면 다음과 같다.

```
void *producer(void *arg) {
    /*프로그램 초기화 구문*/
    while (1) {
        pthread_mutex_lock(&so->lock);
        if(so->eof) {
            pthread_mutex_unlock(&so->lock);
            break;
        }
        read = getdelim(&line, &len, '\n', rfile);
        i++;
        if(read == -1) {
            so->eof = 1;
            so->line = NULL;
            fclose(rfile);
            pthread_mutex_unlock(&so->lock);
            break;
        }
        pthread_mutex_unlock(&so->lock);
        enqueue(queue, line);
        line = NULL;
        len = 0;
    }
    pthread_mutex_lock(&queue->lock);
```

```

queue->eof++;
pthread_cond_broadcast(&queue->not_empty);
pthread_mutex_unlock(&queue->lock);
}

```

- 버퍼가 차있는 상태인지 검사 수행 X
- 임계 영역 밖에서 enqueue 수행

이 경우, 임계 영역 밖에서 enqueue를 수행하는 이유는 큐 자체가 병렬 접근 시 상호 배제를 수행할 수 있도록 구현된 큐이다.

기본적인 큐와 동일하나, 구조체에 뮤텍스와 컨디션 변수가 존재한다.

```

typedef struct _shared_queue {
    node_t *head;
    node_t *tail;
    pthread_mutex_t lock;
    pthread_cond_t not_empty;
    int count;
    int eof;
    int producer_count;
}shared_queue_t;

```

Enqueue 과정에서는 단순히 head와 tail에 대해서 lock을 수행하며, 임계 구역을 벗어나면서 queue가 비어있지 않다는 것을 컨디션 변수로 전달한다.

```

void enqueue(shared_queue_t *queue, char *data) {
    node_t *new_node = malloc(sizeof(node_t));
    new_node->line = data;
    new_node->next = NULL;

    pthread_mutex_lock(&queue->lock);

    if (queue->tail == NULL) {
        // 큐가 비어있는 경우
        queue->head = new_node;
        queue->tail = new_node;
    } else {
        queue->tail->next = new_node;
        queue->tail = new_node;
    }
    queue->count++;

    pthread_cond_signal(&queue->not_empty);
    pthread_mutex_unlock(&queue->lock);
}

```

Dequeue 연산에서는 동일하게 락을 얻고 수행하며, 큐가 비어 있거나 더 이상 enqueue가

되지 않는 경우에 대해서 검사하도록 한다. 만약 eof라면, 더 이상 enqueue가 수행되지 않는 것으로 간주하고 dequeue를 종료한다.

```
char *dequeue(shared_queue_t *queue) {
    pthread_mutex_lock(&queue->lock);

    while (queue->head == NULL && queue->eof == 0) {
        pthread_cond_wait(&queue->not_empty, &queue->lock);
    }

    if (queue->head == NULL && queue->eof > 0) {
        pthread_mutex_unlock(&queue->lock);
        return NULL;
    }

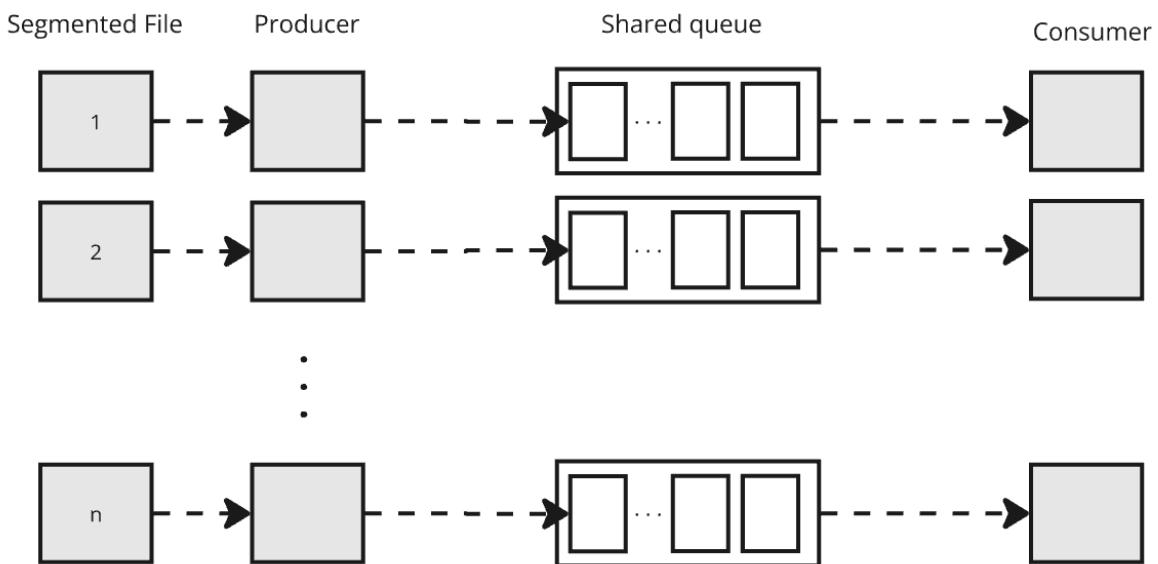
    node_t *temp = queue->head;
    char *data = temp->line;
    queue->head = queue->head->next;

    if (queue->head == NULL) {
        queue->tail = NULL;
    }
    queue->count--;

    pthread_mutex_unlock(&queue->lock);
    free(temp);

    return data;
}
```

프로젝트 빌드 3 *(다중 공유 큐 N:N 생산자-소비자) *최적



세 번째 구현은 하나의 파일에 대해서 접근하는 것이 아닌, 각 생산자마다 분할된 크기의 파일에 대해서 접근하여 파일 읽기 쓰기에 대한 병목을 감소시키고, 생산자와 소비자를 1:1로 구성한 경우다. 이 경우, 소비자와 생산자의 수가 다른 경우, 프로그램의 동작하지 않으므로, 프로그램 구동 시 주의한다. 해당 구현은 하나의 공유 큐에 대해서 락을 얻으려는 경쟁을 완화하고, 또 다른 공유 자원인 파일 디스크립터를 개별적으로 두어 이에 대한 병목을 제거한다.

파일 디스크립터를 개별적으로 두고, 각 생산자마다 파일으로부터 읽어야 할 범위를 지정해 준다. 이 때, 생산자의 수 만큼 나눠진 영역에 대해서 단순하게 처리하는 경우, 실제 통계와 프로그램의 결과가 상이해질 수 있다. 이는 파일의 범위를 나누면서 단어 내에서 범위 경계가 나뉘지는 경우 단어의 길이에 대해서 잘못된 통계를 반환할 수 있으므로, 문장의 끝에서 처리 할 수 있도록 파일의 범위를 재조정한다.

```

/* ai generated acknowledgement */
for(int i = 0; i < Nprod; i++) {
    shared_queue_init(&queue_v[i]);

    prod_arg[i].file = fopen(argv[1], "r");

    // 시작 위치 설정
    prod_arg[i].start_pos = i * f_segment_size;
    if (i != 0) { // 첫 번째 프로듀서가 아닌 경우에만 경계 조정
        fseek(prod_arg[i].file, prod_arg[i].start_pos - 1, SEEK_SET);
        int c;
        // 이전 위치에서 '\n'을 만날 때까지 포인터 이동
        while ((c = fgetc(prod_arg[i].file)) != '\n' && c != EOF) {
            prod_arg[i].start_pos = ftell(prod_arg[i].file);
        }
    }
}
  
```

```
        }
        // 다음 줄의 첫 번째 문자로 이동
        prod_arg[i].start_pos = ftell(prod_arg[i].file);
    }

    // 끝 위치 설정
    if (i == Nprod - 1) {
        // 마지막 프로듀서는 파일 끝까지 읽음
        prod_arg[i].end_pos = file_size;
    } else {
        prod_arg[i].end_pos = (i + 1) * f_segment_size;
        fseek(prod_arg[i].file, prod_arg[i].end_pos, SEEK_SET);
        int c;
        // 다음 '\n'을 만날 때까지 포인터 이동
        while ((c = fgetc(prod_arg[i].file)) != '\n' && c != EOF) {
            prod_arg[i].end_pos = ftell(prod_arg[i].file);
        }
    }

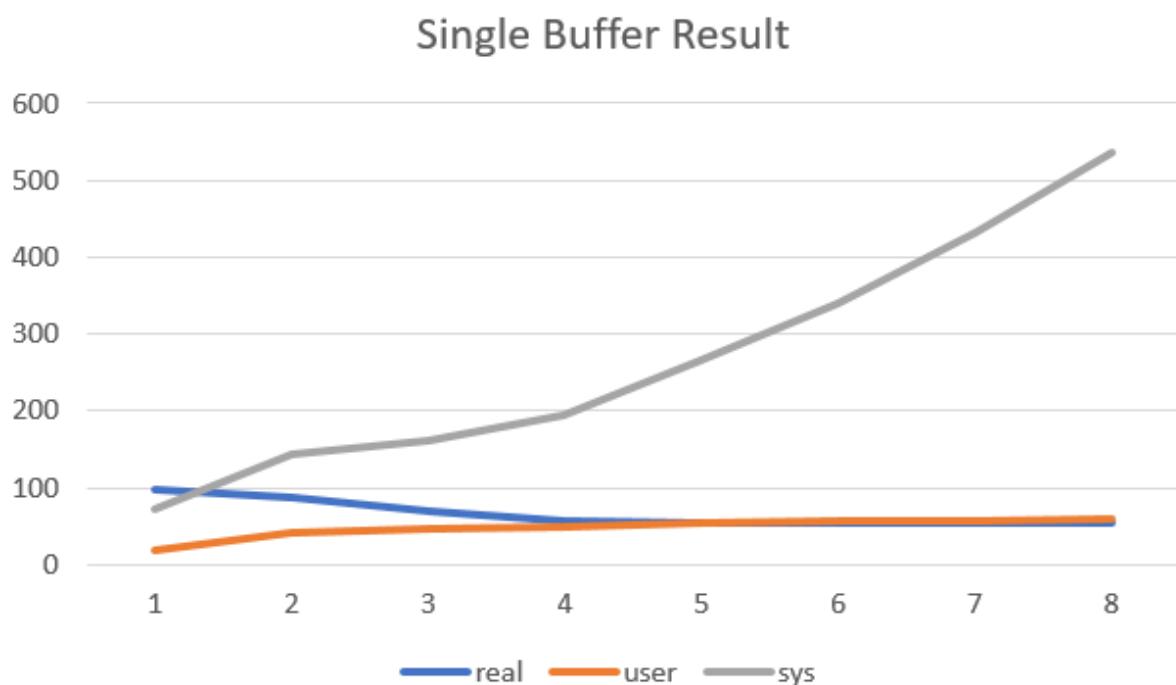
    prod_arg[i].queue = &queue_v[i];
}
```

실험 환경 및 테스트 결과

HW

- CPU: AMD 7950X, Physical 16 Core, Virtual 32Core
- RAM: 128GB
- SSD: 1TB

구현 1

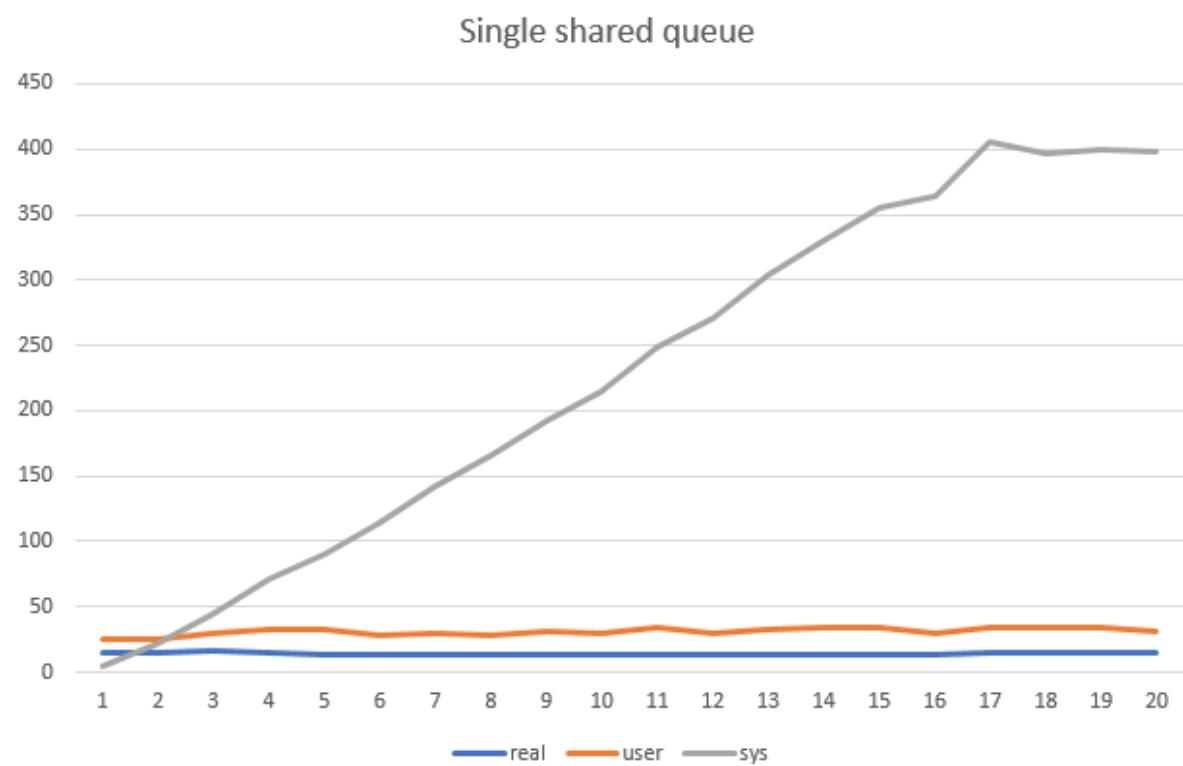


구현 1에 대한 결과, 동일 생산자-소비자 개수로 간략히 성능 측정한 결과, 생산자-소비자 수의 증가에 따라서 실제 처리 시간은 유의미하게 감소하지 않았다. 그러나, 시스템이 점유한 CPU 시간은 스레드의 수가 늘어날수록 가파르게 증가하였는데, 이는 생산자와 소비자가 최대 1개의 문자열만 전달할 수 있기 때문에, 다른 스레드가 broadcast를 통해 깨어났음에도 락을 얻지 못하거나, 버퍼가 차있지 않기 때문에 다시 대기 상태로 전환되어 잡은 문맥 교환이 발생된다.

	real	user	sys
1	97.58	18.787	72.21
2	87.512	42.716	143.309
3	70.075	47.603	161.103
4	57.384	49.818	194.164
5	54.092	54.473	264.799

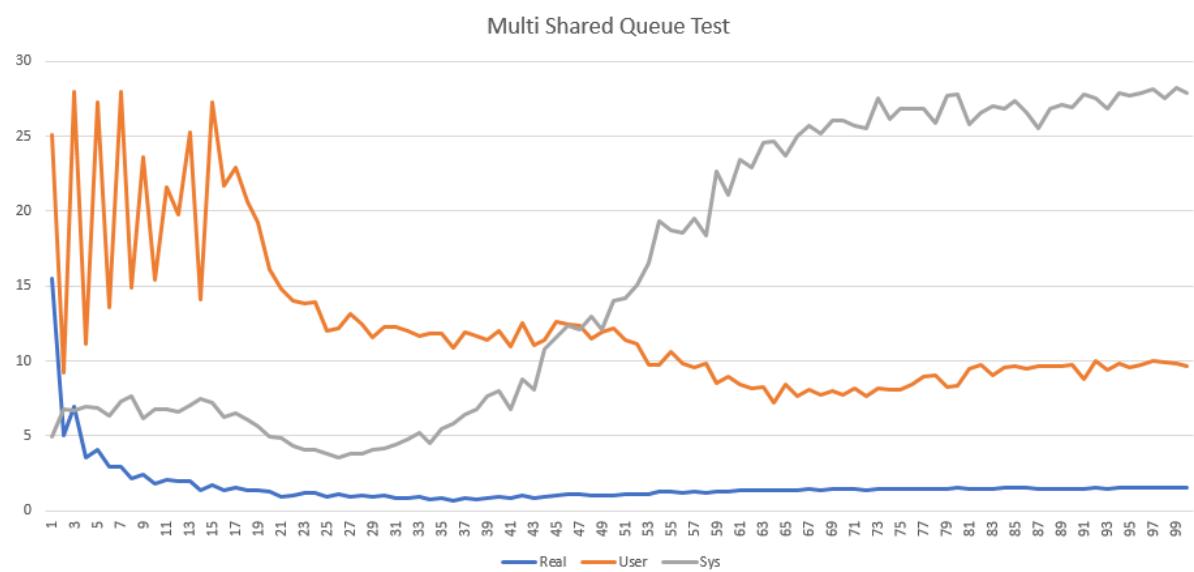
또한 실제 소요된 시간에 대해서도, 유의미한 감소가 발생되지 않았다.

구현 2



상대적으로 단일 포인터 버퍼에 비해, 낮은 처리 시간을 보여준다. 그러나, 스레드의 수가 증가할수록 시스템의 CPU 점유 시간이 증가하였다. 이는 하나의 파일 디스크립터에 대해서, 병렬 작업을 수행하고 있으나, 실질적으로는 하나의 생산자와 하나의 소비자로만 동작하는 것과 다른 점이 없다.

구현 3



파일을 분리하고 각 생산자 – 소비자 간의 단일 버퍼를 두어 처리하도록 구현하였다. 이에 따라서, 파일 접근에 따른 병목을 제거하여 생산자의 효율적인 읽기를 수행할 수 있었다. 현재 구현한 것 중 가장 최적의 상태이며, 다음과 같이 가장 최소의 처리 결과를 얻을 수 있었다.

```
./build/multi_shared_queue FreeBSD9-orig.tar 41 41

real 0m0.799s

user 0m10.949s

sys 0m6.807s
```

그러나, 여전히 스레드의 수가 증가함에 따라서 시스템의 CPU 점유 시간이 증가한다. 이는 문맥 교환과 락 경합, I/O 작업의 증가에 따라 발생하는 것으로 고려된다.

결론

이렇게 멀티 스레드 프로그램을 구현하였으나, 실질적으로 CPU를 제대로 활용하지 못하고 있다. 실제 `char_stat.c`가 `FreeBSD9-orig.tar`를 실행하는데 실제 시간 1.8초, user와 system 시간 각각 1.7초 0.082초를 기록하며 본인이 구현한 멀티스레드 프로그램에 비하여 큰 성능 개선이 없다. 특히나, 단일 버퍼 혹은 단일 큐를 통한 멀티 스레딩은 스레드 간 경합으로 인하여 성능 개선이 어려웠다. 또한 경쟁 상태를 회피하고 올바른 결과를 얻기 위해, 작은 프로그램 수정과 AI의 도움을 받았다. AI의 도움을 받은 프로그램 구문은 `/* ai generated coded acknowledgement */`라고 주석을 작성해 두었다.