Operating Systems

& Adv. Mobile project HW2:

Multi-threaded word count

모바일시스템공학과 32217259 문서영

[목차]

1. 개요

첫 번째 과제에서는 fork 를 활용하여 멀티프로세스를 구현했습니다. 이번 과제에서는 파일의 알파벳 개수를 세는 작업을 **멀티스레딩 환경**에서 **Producer-Consumer 문제**를 활용하여 구현했습니다. 이 문제는 하나 이상의 프로듀서(Producer)와 소비자(Consumer)가 공유 자원을 통해 데이터를 주고받는 상황을 다루며, 동시 접근 문제를 처리하는 방법을 배우기 위한 중요한 개념입니다. 이를 해결하기 위해 이전 과제에서 사용했던 Process 와 Thread 의 개념을 비교하여 정리하였고, 동기화 문제에 대한 정의와 그 해결 방법을 알아보았습니다.

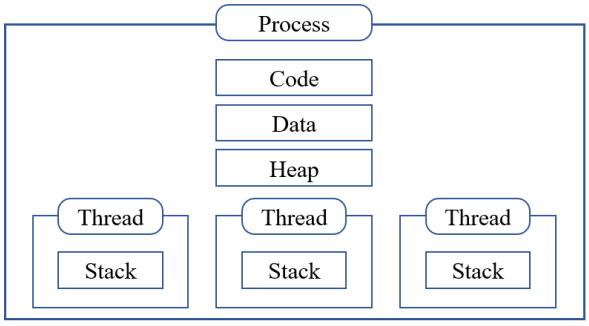
또한, 코드 작업을 통해 발생한 여러 동기화 문제에 대해 직접 확인하면서 해결하였고, 이를 통해 이론만으로 어렵다고 느껴진 작업들을 좀 더 수월하게 이해할 수 있었습니다.

2. 배경지식

1. Process and Thread

Thread 를 정의하기 전에 Process 에 대해 간단히 복습하고 넘어가려고 합니다. Process 는 실행 중인 프로그램의 instance 로 Heavy Weight Process 라고 부르며, process 를 실행하면 각 process 마다 address 가 부여되기 때문에 메모리 보호가 가능합니다. 여기서 운영체제가 Process 를 관리하는 방법이 있습니다. 먼저 Process Control Block(PCB)을 사용하여 Process 의 현재 상태를 일종의 "스냅샷"을 사용하여 정보를 저장하고, Scheduling 을 사용하여 CPU 시간을 공정하게 할당합니다. 또한 비 CPU resource 에 대해서는 보호 메커니즘을 사용하여 안전하게 자원을 사용하도록 합니다.

Thread 는 Light Weight Process 라고 불리며, 프로세스 내에서 실행되는 더 작은 단위(흐름의 단위)라고 부릅니다. Process 와 달리 동일 Address 안에 thread 가 여러 개 들어갈 수 있으며, 메모리 공간을 공유하게 됩니다. 하지만 각 Thread 마다 독립적으로 실행할 수 있습니다. 그렇기에 Process 보다 빠르고 효율적입니다.



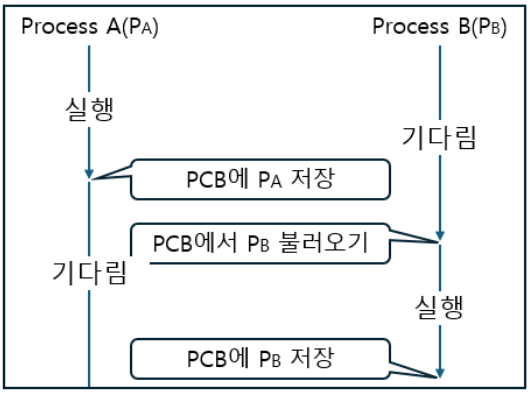
[A Process and three Threads Structure]

2. Synchronization problem

동기화(Synchronization)은 다수의 process 또는 thread 를 동시에 실행해도 공유 데이터의 일관성을 유지하는 것입니다. 하나의 process 내에서 공유되는 자원을 사용함으로써 메모리를 효율적으로 사용하고 thread 간 context switching 이 가볍다는 장점이 있지만, 동시에 자원을 공유하기 때문에 공유 자원 접근 순서를 정하여 문제가 발생하지 않도록 해야 하는 동기화 작업을 해야 한다는 단점이 존재합니다.

여기서 Context switching(문맥 교환)은 운영체제에서 여러 프로세스나 스레드를 실행할 때 CPU 가 현재 실행 중인 프로세스의 상태를 저장하고 새로운 프로세스로 전환하는 과정을 말합니다. 주요 과정으로는 먼저 현재 프로세스 상태를 PCB(Process Control Block)에 저장합니다. CPU 가 실행한 후 다음 프로세스의 상태 정보를 PCB 에서 가져와 CPU 에 새 프로세스의 레지스터 값과 프로그램

카운터 등이 다시 설정됩니다. 그 후 이전 프로세스의 상태를 저장하고 새로운 프로세스의 상태를 복원한 후, CPU 는 새로운 프로세스의 코드 실행을 시작합니다.



[Context Switching Image]

3. Critical section and Mutual Exclusion, Race Condition

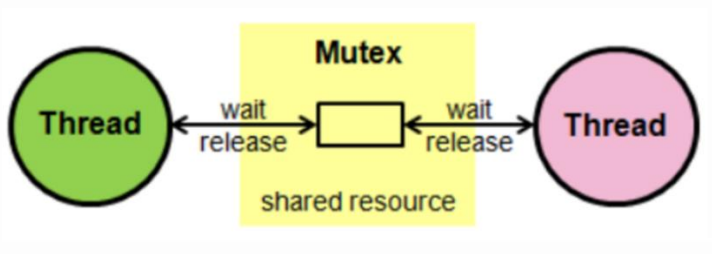
Critical section(임계 영역)은 공유 자원에 여러 스레드나 프로세스가 동시에 접근할 수 없도록 보호해야 하는 코드 영역입니다. 이 영역에서는 여러 스레드가 동시에 공유 자원에 접근하면 자원의 무결성이 깨질 수도 있기 때문에 하나의 스레드만 Critical section 에 들어가 작업을 수행하도록 해야 합니다. 이러한 특징을 **Mutual Exclusion(상호 배제)**이라고 하며, Critical section 을 보호하기 위한 방법으로는 Mutex(뮤텍스), Semaphore(세마포어), Condition Variable(조건 변수)와 같은 동기화 도구들이 있습니다.

Race Condition(경쟁 조건)은 여러 프로세스 또는 스레드가 동시에 실행되면서 공유 자원에 대한 작업 순서가 보장되지 않아 의도치 않은 결과가 발생하는 상황을 말합니다. Race Condition 은 여러 스레드가 동시에 실행될 때, 같은 공유 자원에 접근하면서, 실행 순서에 따라 결과가 나올 수 있는 경우에 발생하게 됩니다. 이러한 상황에서 실행 결과가 결정적이지 않기 때문에 프로그램이 매번 다르게 실행될 수 있습니다. 이는 특히 멀티 스레드 환경에서 코드가 매우 빠르게 실행이 될 때, 하나의 스레드가 작업을 끝내기 전에 다른 스레드가 그 자원을 사용하려고 할 때 발생합니다. (제가 작업한 producer-consumer 코드 작업을 할 때 가장 해결하기 어려웠던 부분인 것 같습니다.)

Race Condition 을 방지하기 위해 Critical Section 을 사용하여 공유 자원에 대한 동시 접근을 막도록 합니다. 또한 Mutex 나 Lock 과 같은 동기화 메커니즘을 사용하여 스레드가 순서대로 자원에 접근하도록 하면 데이터의 무결성을 보장할 수 있습니다.

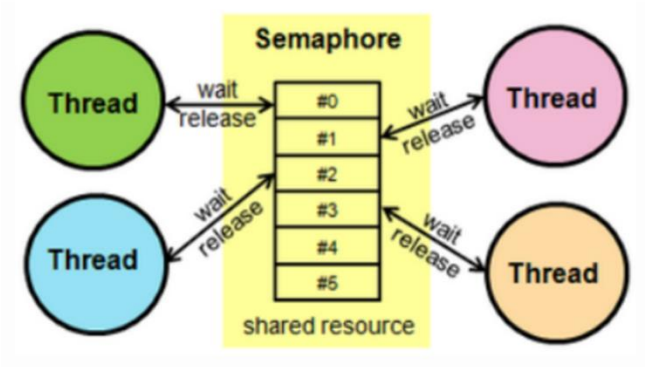
4. Mutex and Semaphore

Mutex(뮤텍스)는 Mutual Exclusion 의 약자로, 오직 하나의 스레드만 공유 자원에 접근할 수 있도록 하는 Lock 입니다. 공유 자원을 독점적으로 잠금하여 여러 스레드가 동시에 자원에 접근하지 못하도록 합니다. Lock 과 Unlock 의 상태를 가지는데, 특정 스레드가 lock 을 하였을 때 이 스레드만이 unlock 을 할 수가 있습니다. Mutex 의 장점은 한 번에 한 스레드만 자원을 사용할 수 있도록 하기 때문에 공유 자원에 대한 단일 스레드 접근을 보장합니다.



[Mutex 동작 방식]

Semaphore(세마포어)는 공유 자원에 접근할 수 있는 최대 스레드 수를 제한하는 count 기반 동기화 기법입니다. 동시에 여러 개의 프로세스 또는 쓰레드의 임계구역에 접근할 수 있도록 카운트(스레드 또는 프로세스의 허용 가능한 최대 수)를 갖고 있습니다. 여기서 만약 count 가 1 이라면 Mutex 가 됩니다. (뮤텍스: 공유 자원 1 개, 세마포어: 공유 자원 2 개 이상).



Semaphore

[Semaphore 동작 방식]

5. DeadLock

DeadLock(교착 상태)는 두 개 이상의 스레드 또는 프로세스가 서로 상호 의존적인 상태에 빠져 무한정 대기하며 더 이상 진행할 수 없는 상황을 말합니다. 이에 대한 조건으로는 총 4 가지가 있습니다. 가장 먼저 Mutual Exclusion(상호 배제)로, 자원은 나눠서 사용할 수 없고, 한 번에 한 스레드만 접근할 수 있습니다. 두 번째, Hold and Wait(점유 대기)는 자원을 점유한 상태에서 다른 자원을 요청하고 기다리는 상태입니다. 또한 Non-preemptive(비선점)은 자원을 점유한 스레드가

스스로 자원을 해제할 때까지 강제로 회수할 수 없고, 기다려야 합니다. 마지막 조건은 Circular Wait(순환 대기)로 여러 스레드가 서로 순환적으로 자원을 요청하며 대기하는 상태입니다.

6. Producer-Consumer problem (Bounded-Buffer problem)

Producer-Consumer Problem 은 멀티 스레드 프로그래밍에서 자주 등장하는 동기화 문제 중 하나로, 하나 이상의 생산자(Producer)와 소비자(Consumer)가 공유 자원을 통해 데이터를 주고받는 상황을 다루며, 공유 자원에 대한 동시 접근을 관리하는 방법을 배우기 위해 중요한 개념입니다.

생산자는 데이터를 생성하는 역할을 하며, 생산된 데이터를 공유 버퍼에 넣습니다. 소비자는 공유 버퍼에 있는 데이터를 소비하며, 버퍼에서 데이터를 꺼냅니다. 버퍼는 제한된 크기를 가졌기에 생산자가 버퍼에 넣을 때 버퍼가 가득 차 있으면 대기하고, 소비자가 버퍼에서 데이터를 꺼낼 때 버퍼가 비어 있으면 대기해야 합니다.

만약 버퍼가 꽉 차거나 비어 있으면 대기를 해야 한다는 문제가 발생하는데, 이를 해결하기 위한 방법이 동기화 메커니즘입니다. 주로 Mutex 와 Condition Variable, Semaphore 를 사용하여 해결할 수 있습니다.

3. 코드 설명(작성 과정 및 오류 설명, 출력값)

**** mutex_lock/unlock만 사용했을 경우 (오류)**

```
assam_seoyoung@assam:~/2024-os-hw2$ ./edit input_file 1 4
Prod_8fff7640: 28 lines
Cons: 0 lines
Cons: 0 lines
Cons: 0 lines
main continuing
Cons: 0 lines
main: consumer_0 joined with 0
main: consumer_1 joined with 0
main: consumer_2 joined with 0
main: consumer_3 joined with 0
main: producer_0 joined with 28
```

위와 같은 결과가 나왔습니다. producer는 line을 잘 읽었지만, consumer에 line을 전달할 때 오류가 발생하여 모두 0 line을 처리하는 것을 볼 수 있습니다. 이는 producer와 consumer 간의 동기화 문제라 볼 수 있습니다. producer가 모든 라인을 처리하기 전에 consumer가 읽고 데이터를 소비해야 하는데, 그 전에 producer가 계속 작업을 수행하여 consumer가 빈 데이터를 처리하는 문제가 발생하였습니다.

1. producer : consumer = 1 : 1

위의 문제를 해결하기 위해 조건 변수를 넣어주었습니다.

```c
typedef struct sharedobject {
    FILE *rfile;
    int linenum;
    char *line;
    pthread_mutex_t lock;
    pthread_cond_t cond; // 조건 변수: 특정 조건이 만족될 때 thread 대기/시작 사용
    int full;
} so_t;
```

[1:1 producer:consumer 의 공유자원 구조체]

또한 pthread_cond_wait, pthread_cond_signal 함수를 사용해 producer 와 consumer 가 서로 데이터가 생성 및 소비되었음을 알릴 수 있도록 하였습니다. 함수를 사용하여 공유 자원이 겹치지 않도록 적절하게 사용하니 1:1 관계는 잘 수행되었습니다.

```
assam_seoyoung@assam:~/2024-os-hw2$ ./edit input_file 1 1
main continuing
Cons_57953640: [00:00] Line 1
Cons_57953640: [01:01] Line 2
Cons_57953640: [02:02] Line 3
Cons_57953640: [03:03] Line 4
Cons_57953640: [04:04] Line 5
Cons_57953640: [05:05] Line 6
Cons_57953640: [06:06] Line 7
Cons_57953640: [07:07] Line 8
Prod_58154640: 8 lines
Cons: 8 lines
main: consumer_0 joined with 8
main: producer_0 joined with 8
```

[input_file 넣었을 경우]

기본 코드를 수정하여 알파벳 개수를 세는 작업을 수행하였습니다.

```
main: consumer_0 joined with 2073
main: producer_0 joined with 2073
A:      160
B:      160
C:      160
D:      160
E:      160
F:      160
G:      160
H:      160
I:      160
J:      160
K:      160
L:      160
M:      160
N:      160
O:      160
P:      160
Q:      160
R:      160
S:      160
T:      160
U:      160
V:      160
W:      160
X:      160
Y:      160
Z:      160
```

[input_file 알파벳 출력 결과]

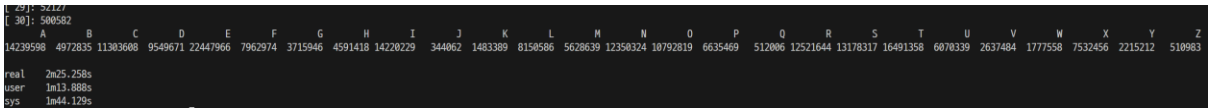
```
}assam_seoyoung@assam:~/2024-os-hw2$ time ./one_prod_cons /opt/FreeBSD9-orig.tar 1 1
main continuing
Cons: 21262918 lines
Prod_949ee640: 21262918 lines
main: consumer_0 joined with 21262918
main: producer_0 joined with 21262918
*** print out distributions ***
  &ch  freq
[  1]: 9077967  ************
[  2]: 7245746  **********
[  3]: 5473711  ********
[  4]: 5700289  ********
[  5]: 3208375  ****
[  6]: 7692940  ***********
[  7]: 2824641  ****
[  8]: 2585468  ***
[  9]: 1552922  **
[ 10]: 2170713  ***
[ 11]: 988332   *
[ 12]: 806050   *
[ 13]: 706301   *
[ 14]: 596018
[ 15]: 526804
[ 16]: 453856
[ 17]: 358912
[ 18]: 354497
[ 19]: 268067
[ 20]: 280063
[ 21]: 201742
[ 22]: 164614
[ 23]: 145650
[ 24]: 125262
[ 25]: 104175
[ 26]: 82598
[ 27]: 70950
[ 28]: 64521
[ 29]: 52127
[ 30]: 500582
        A       B       C       D       E       F       G       H       I       J       K       L       M       N       O       P       Q       R       S       T       U
        V       W       X       Y       Z
14239598  4972835 11303608  9549671 22447966  7962974  3715946  4591418 14220229   344862  1483389  8150586  5628639 12350324 10792819  6635469   512006 12521644 13178317 16491358  6070339
  2637484  1777558  7532456  2215212   510983

real    4m10.693s
user    1m51.417s
sys     2m36.538s
```

[FreeBSD9-orig.tar 넣었을 경우]

for문을 돌면서 문자열에서 알파벳의 개수를 측정하는 코드를 사용했습니다.

```
assam_seoyoung@assam:~/2024-os-hw2$ ./edit input_file 1 2
main continuing
Cons_f6c26640: [00:00] Line 1
Cons_f6c26640: [01:01] Line 2
Cons_f6c26640: [02:02] Line 3
```

하지만 input_file은 잘 작동하는 것 같았지만, FreeBSD9-orig.tar처럼 긴 코드에서는 **"./char_stat"**의 출력값과 달리 중복되는 값이 존재했습니다.



[./one_prod_cons char_stat.c 1 1]

위의 코드를 사용하지 않고, **"char_stat.c"**의 코드를 사용하여 consumer에 코드를 넣어주니 동작이 잘 되는 것을 확인할 수 있었습니다.

2. producer : consumer = N : N

이를 구현하기 위해 버퍼를 추가하여 FIFO 방식인 원형 큐를 사용하기로 하였습니다. 또한 구조체 자체를 배열로 선언하여 코드를 구성할지, 구조체 안에 buffer를 2차원 배열로 해야 속도가 향상될지 고민을 하였습니다. 만약 구조체를 배열로 선언하게 된다면 multi-thread 차원에서 프로듀서-컨슈머 사이에 데이터 공유가 어려워지고, 공유 자원이라는 의미가 사라질 것 같아서 구조체 내에서 char **buffer를 사용하여 동작하도록 구현하였습니다.

방식은 원형큐(circle queue)를 사용하여 in/out의 flag로 버퍼가 찼는지 쉽게 확인하도록 하였습니다.

```c
typedef struct sharedobject {
    FILE *rfile;
    int linenum;
    char *line;
    char **buffer;
    int buffer_size;
    int in; int out;
    int cnt;

    pthread_mutex_t lock;
    pthread_cond_t cond_prod; // 조건 변수: 특정 조건이 만족될 때 thread 대기/시작 사용
    pthread_cond_t cond_cons; // 조건 변수: 특정 조건이 만족될 때 thread 대기/시작 사용
    pthread_attr_t attr;
    int full; int done;
} so_t;
```

[공유 자원 구조체 수정]

그래서 공유 자원의 구조체를 약간 변형시켜 주었습니다.

하지만 동작이 되지 않아서 확인을 해보니 버퍼에 들어간 데이터의 양을 제대로 체크를 하지 않아 발

생하였다는 걸 알고 버퍼에 들어간 데이터 개수를 세는 **int cnt**를 추가하였습니다. in과 out은 데이터 출입을 관리하는 변수이고, cnt와는 다른 변수입니다. 만약 in과 out이 같다면, 또한 cnt가 buffer_size와 같다면 producer에서 full을 1로 설정하였습니다.

```
// 데이터가 소비될 때까지 기다림
while (so->full == 1 && so->done == 0) { // 언제 full?? -- 데이터가 큐에 다 채워졌을 때??
    pthread_cond_wait(&so->cond_prod, &so->lock);
}

so->linenum = i;
// so->line = strdup(line);        /* share the line -> line 복제하여 저장 */
so->buffer[so->in] = strdup(line);
so->cnt++;
so->in = (so->in + 1) % so->buffer_size;
i++;
if(so->in == so->out && so->cnt == so->buffer_size) so->full = 1; // 만약 in == out 이면
```

[원형 큐 작업]

모든 데이터를 다 읽어왔다면, done이라는 변수를 사용하여 producer의 작업을 완전히 끝내도록 하였습니다. 하지만 전혀 동작이 되지 않았고, 다시 코드를 수정하였습니다.

```
int stat[MAX_STRING_LENGTH];
int stat2[ASCII_SIZE];
int count = 0;

typedef struct sharedobject {
    char *lines[BUF_SIZE];
    int full[BUF_SIZE];
    int done;
    int in;
    int out;
    pthread_mutex_t lock;
    pthread_cond_t cond_prod;
    pthread_cond_t cond_cons;
} so_t;

so_t share;
FILE *rfile;
int Nprod, Ncons;
```

[밑의 FreeBSD9-orig.tar 1 1 출력 결과 후 수정 작업]

File은 공유자원에서 꺼냈고, 각 lines를 읽으면서 라인이 찼는지 확인하기 위해 full을 lines의 개수와 같게 하여 배열로 만들었습니다. Mutex_lock의 작업은 producer에서 파일에서 라인 읽어올 때, 공유자원의 lines에 넣어줄 때 걸어주었고, cosumer에서는 라인이 채워졌는지 확인할 때, 라인을 공유자원에서 꺼낼 때, 알파벳 통계작업을 진행할 때 걸어주었습니다.

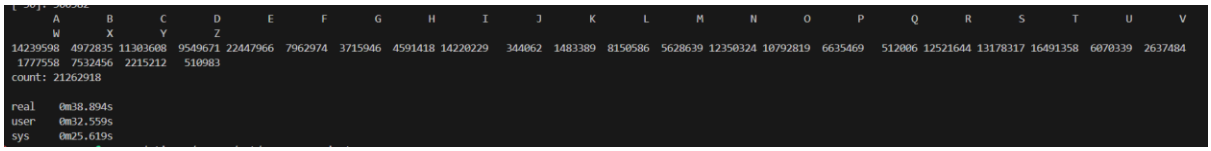
```
    A       B       C       D       E       F       G       H       I       J       K       L       M       N       O       P       Q       R       S       T       U       V
    W       X       Y       Z
24461614  8093992 18556186 16466922 39758123 13002010  6734619  8906383 25609995   590184  2836338 15330830  9889498 22722612 20803479 11540733   842309 23286746 23844047 30560895 10794017  4540040  34
50958 11301267  4242881   996555
count: 2126291 8

real    0m33.027s
user    0m32.917s
sys     0m19.668s
```

[**~$ time ./n_pc2 /opt/FreeBSD9-orig.tar 1 1** 출력 결과]

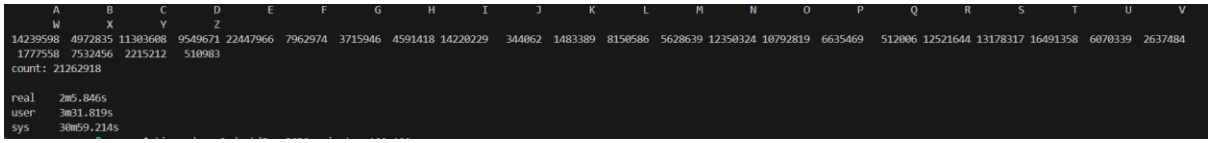
출력되는 결과값은 같게 나왔지만, A의 값만 봐도 파일의 전체 라인 개수를 넘어가고, char_stat.c의

출력값보다 2배 넘게 알파벳을 가져왔습니다. 이를 고치기 위해 Claude의 도움을 많이 받았습니다. Pthread_cond_broadcast는 동시에 대기하고 있던 producer, consumer가 전부 깨어날 수 있기 때문에 pthread_cond_signal로 다시 수정하였습니다. 불필요한 낭비를 줄이기 위해 broadcast는 producer 안에서 파일을 다 읽어왔을 때만 사용하였습니다.

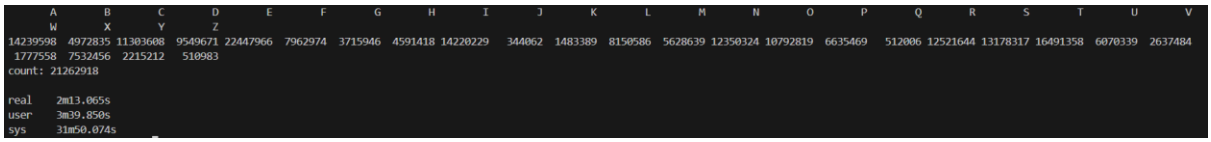
또한 라인의 중복 처리를 막기 위해 producer에서 getline으로 line을 읽어오기 전에 mutex_lock을 걸어주도록 하였습니다. consumer에서는 line을 가져와 알파벳 개수를 세는 작업에도 mutex_lock을 해주었으며, char_stat.c의 코드를 사용하였습니다.



[**~$ time ./n_pc2 /opt/FreeBSD9-orig.tar 1 1** 출력 결과]



[**~$ time ./n_pc2 /opt/FreeBSD9-orig.tar 30 30** 출력 결과]



[**~$ time ./n_pc2 /opt/FreeBSD9-orig.tar 100 100** 출력 결과]

위의 출력 결과는 line의 BUF_SIZE를 5로 설정했을 때의 결과인데, BUF_SIZE를 증가시킬수록 출력 시간이 줄어든다는 걸 확인하였습니다. 이 이유는 버퍼 사이즈가 작으면 버퍼가 빠르게 채워지기 때문에 lock을 많이 걸어 오버헤드가 증가하게 됩니다.

또한 1:1의 producer:consumer보다 n:m이 더 빠르게 동작할 거라고 생각했는데, 1:1의 작업이 가장 빨랐습니다. 이 문제는 위의 이유와 유사하게 lock을 더 많이 건다는 이유였고, lock contention, context switching, and I/O limitations의 이유로 다중 producer-consumer의 장점이 사라지게 되었습니다.

4. 출력 결과

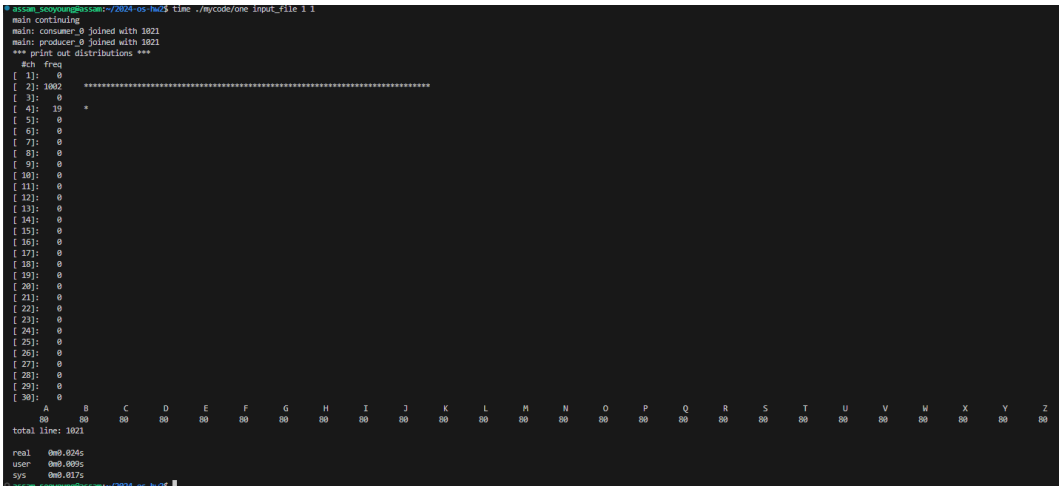
출력 방법

```
assam_seoyoung@assam:~/2024-os-hw2$ gcc ./mycode/one_prod_cons.c -o one
assam_seoyoung@assam:~/2024-os-hw2$ gcc ./mycode/n_prod_cons.c -o n_pc
```

위의 사진과 같이 one 과 n_pc 의 코드를 따로 생성할 수 있습니다.

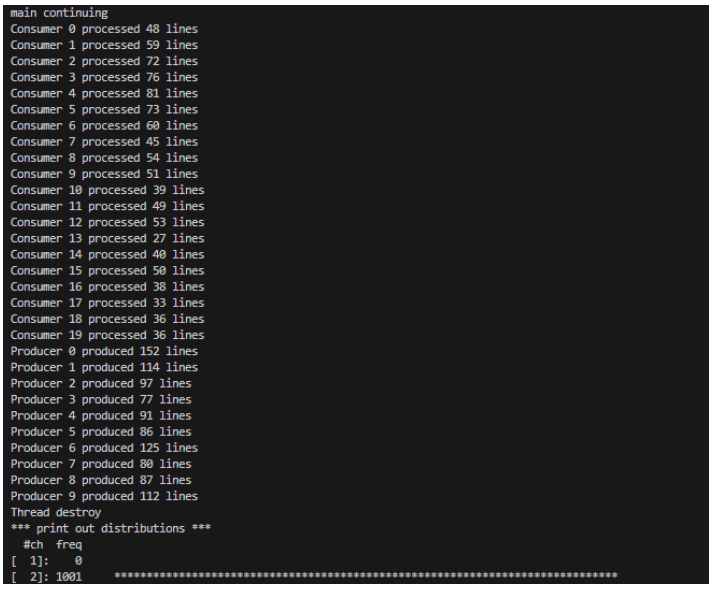
```
assam_seoyoung@assam:~/2024-os-hw2$ time ./mycode/one input_file 1 1
```

위의 코드에 input_file 을 예시로 넣어보면 밑의 사진과 같은 결과를 확인할 수 있습니다.

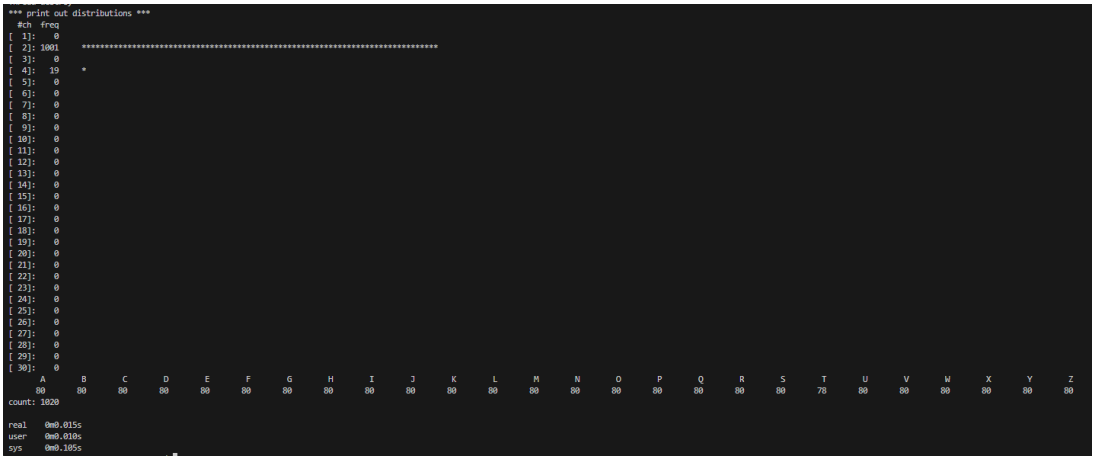



```
assam_seoyoung@assam:~/2024-os-hw2$ time ./mycode/n_pc input_file 10 20
```

위의 작업을 수행하면 밑과 같은 결과가 나옵니다.



각 Producer, consumer 가 수행한 라인의 개수가 나오고, 밑에는 결과의 출력값이 나옵니다.



5. 어려웠던 점 및 후기

Pthread_mutex 를 활용하여 producer-consumer problem 을 구현하니, 동시에 작업을 수행할 수 있다는 걸 눈으로 확인할 수 있어 좋았습니다. 1:1 의 관계까지는 큰 어려움 없이 구현을 했는데, n:m 의 관계인 다중 producer-consumer 에서는 큰 파일을 넣었을 때 알파벳 몇 개를 읽지 않거나 원하는 개수보다 더 많이 가져오는 중복이 발생하기도 하는 동기화 구현이 제대로 되지 않았습니다. 또한 개념이 추상적이어서 코드 작업을 하면서 이해하는 게 어려웠습니다. 그리고 broadcast 는 무조건 n:m 의 관계에서 사용해야 할 거라고 생각했는데, broadcast 가 대기하고 있는 모든 스레드를 깨우기 때문에 시간이 더 오래 걸릴 수도 있다고 하여 signal 로 바꾸니 동작이 되었습니다. 여기서 제가 알고 있던 것과 달라서 좀 신기하기도 했지만, 구현을 제대로 하고 있는지에 대한 의문도 남았습니다.

과제를 끝나고 나니 1:1 의 관계에 대한 시간이 다중 producer-consumer 의 관계보다 더 적게 나와 아쉬움이 많이 남았습니다. ChatGPT 나 Claude 의 많은 도움으로 문제를 해결할 수는 있었지만, 다음에는 다른 도구 없이 혼자서 해결해보고 싶은 마음이 많이 들었습니다.