전남대학교 '운영체제' - 박태준 교수

Assignment #2. Synchronization and memory

아래 문제들을 해결 한 후 풀이(해설)과 함께 이클래스를 통해 제출하세요.

- 즉, 소스파일 + 레포트가 제출되시면 됩니다.

- 레포트 양식은 자유. 단, 너무 불필요하게 길지는 않게!

- 파일형식은 pdf로 제출 바랍니다!

- 코딩은 큰 주제와 맥락을 벗어나지 않는 선에서, 자유롭게. 창의롭게!

- 저는 구글 검색을 적극 권장해드립니다. 검색도 능력입니다! (너무 대놓고 베끼진 마시고...)

- chatgpt는 그닥...추천 못드리겠습니다... 지난 과제에서 잘 안되셨던 분들의 원인 중 하나가 chatgpt를 참조한 경우가 더럿 있더라구요.

*주의: 본 과제를 위한 코드들은 **xNIX 계열 (Linux 또는 MacOS)에서** 실행가능한 코드들 입니다. 윈도우 사용자 분들은

1. WSL (Windows subsystem for Linux)
2. VMware 또는 VirtualBox등의 가상화 환경

등에서 본 과제를 하시면 되겠습니다. (지난시간과 다르게 헤비한 작업이 아니니, 어디서 하셔도 무방할겁니다!)

기한: 6월 16일 금요일 까지!!!

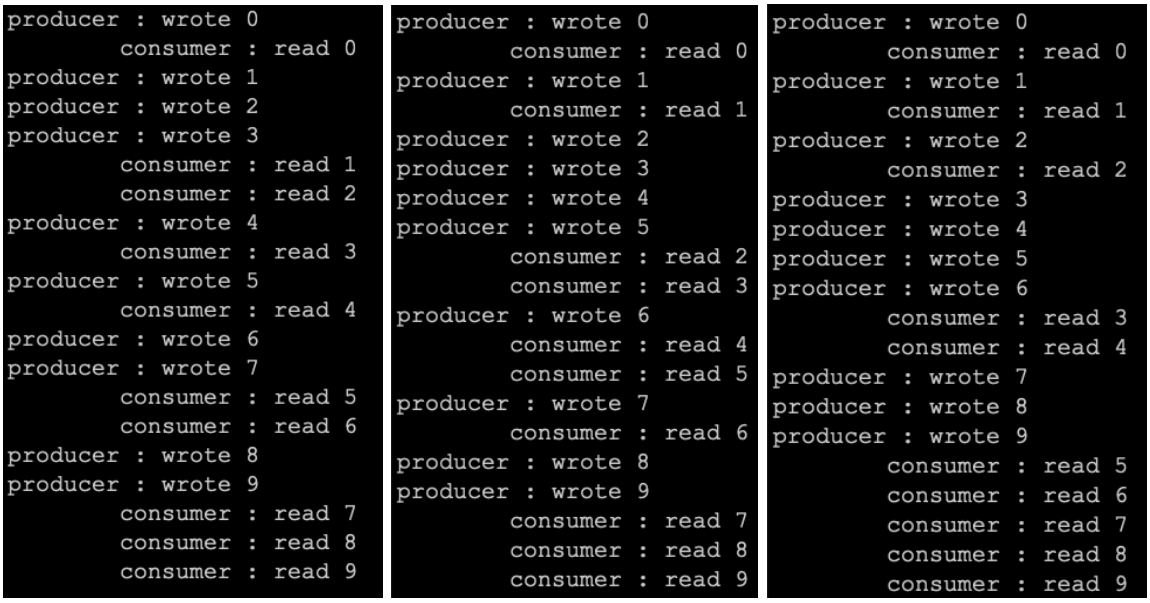
#1. 생산자-소비자로 구성된 응용프로그램 만들기

- 음... 커널 컴파일 과제... 많은 분들이 어려움을 겪으셨던 듯 합니다... 아무래도 리눅스가 아직 친하지 않거나, 시스템 레벨이 친하지 않거나, 또는 컴파일이 한시간 씩이나 걸리는 경험을 처음 해보시거나 등등, 이런 저런 이유로 어려웠던듯 합니다! 그래서 이번에는 아주 정석적인 문제를 드리고자 합니다. 바로 생산자-소비자 문제입니다. 그 중에서도, **"1개의 생산자 스레드와 1개의 소비자 스레드로 구성되는 간단한 응용프로그램을 작성"**을 해보는 겁니다. 조건은 다음과 같습니다.

- 생산자 스레드
  - 0~9까지 10개의 정수를, 랜덤한 시간 간격으로, 공유버퍼에 쓴다.
- 소비자 스레드
  - 공유버퍼로부터 랜덤한 시간 간격으로, 10개의 정수를 읽어 출력한다.

- 공유버퍼
  - 4개의 정수를 저장하는 원형 큐로 작성
  - 원형 규는 배열로 작성
- 2개의 세마포어 사용
  - semWrite : 공유버퍼에 쓰기 가능한 공간(빈 공간)의 개수를 나타냄
  - 초기값이 4인 counter 소유
  - semRead : 공유버퍼에 읽기 가능한 공간(값이 들어 있는 공간)의 개수를 나타냄
  - 초기값이 0인 counter 소유
- 1개의 뮤텍스 사용
  - pthread_mutex_t critical_section
  - 공유버퍼에서 읽는 코드와 쓰는 코드를 임계구역으로 설정
  - 뮤텍스를 이용하여 상호배제

이상의 조건을 만족시키는 프로그램을 실행을 하면 다음과 같은 결과가 나옵니다. 물론 랜덤한 시간으로 작동되는 것이라서, 결과는 매 실행마다 조금씩 다를 수 있습니다.



```
producer : wrote 0
        consumer : read 0
producer : wrote 1
producer : wrote 2
producer : wrote 3
        consumer : read 1
        consumer : read 2
producer : wrote 4
        consumer : read 3
producer : wrote 5
        consumer : read 4
producer : wrote 6
producer : wrote 7
        consumer : read 5
        consumer : read 6
producer : wrote 8
producer : wrote 9
        consumer : read 7
        consumer : read 8
        consumer : read 9
```

```
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
producer : wrote 3
producer : wrote 4
producer : wrote 5
        consumer : read 2
        consumer : read 3
producer : wrote 6
        consumer : read 4
        consumer : read 5
producer : wrote 7
        consumer : read 6
producer : wrote 8
producer : wrote 9
        consumer : read 7
        consumer : read 8
        consumer : read 9
```

```
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
producer : wrote 4
producer : wrote 5
producer : wrote 6
        consumer : read 3
        consumer : read 4
producer : wrote 7
producer : wrote 8
producer : wrote 9
        consumer : read 5
        consumer : read 6
        consumer : read 7
        consumer : read 8
        consumer : read 9
```

이렇게 하면 여러분들이 많이 혼동스러우실 것 같습니다! 스켈레톤, 'procon.c'를 제공해드립니다! procon.c 파일에서 [Write here]을 중심으로 완성해주시면됩니다. 혹여, 빈칸채우기가 방해되신다면 처음부터 편하신대로 작성하셔도 됩니다!

- **Step 1)** 'os06_syncronization' 강의자료에서 생성자-소비자 문제를 다시한번 보세요.
- **Step 2)** 주어진 조건에 맞는 간단한 생성자-소비자 문제 애플리케이션을 만들어보세요.

- **Step 3)** 함께 제공된 procon.c 파일은 본 문제의 스켈레톤 코드입니다. [Write here]를 중심으로 작성하시면 더욱 쉬우실 겁니다.
- **기대 결과물)** 내용이 반영된 procon.c, 이에 대한 설명 또는 레포트
- **주의사항)** 요상하게 macOS에서 Semaphore가 정상적으로 작동하지 않는 경우가 확인되고 있습니다. 가능하면 Windows의 WSL이나, 별도의 Linux 환경에서 작성하시는 것을 추천드립니다.
- **검색 키워드)** "생산자-소비자 문제"

#2. 소프트웨어로 문을 만드는 방법

- 두번째 문제는 수업때 잠깐 언급드렸던, 소프트웨어 적으로 동기화를 구현하는 방법입니다. Dekker 알고리즘, Peterson 알고리즘, Lamport 알고리즘등이 있는데요, 사실 오늘날에는 잘 쓰이지는 않는 것으로 알고있습니다. 특히 대부분 Busy-waiting에 기반한 방식이라 성능상에서 손해가 크거든요; 그래도 이 친구들의 구조를 파악해보는 것은 동기화라는 개념을 이해하는데 있어서, 그리고 그로부터 향후 유사한 문제를 해결하데 있어서 크게 도움이 됩니다. 예를 들어, 수업시간에 배운 하드웨어적인 방법들은 대부분 단일 CPU, 단일 장치에서만 유효한 방법일 뿐, 분산 환경등 서로 다른 장치간에는 활용하기 어려우므로 분명 다른 방법이 필요할겁니다!

그런 의미에서, 소프트웨어 기반 동기화 방식에 대해 간단히 조사를 해보시고, 그중 하나를 직접 구현하시어 #1번의 생산자-소비자 문제에서 pthread_mutex 대신 적용해보세요. 인터넷에 정말 많은 자료들이 있으므로, 그리 어렵진 않으실겁니다!

- **Step 1)** 소프트웨어 기반 동기화 방식(알고리즘)**들**에 대해 간단한 조사를 해보시고, 그에 대한 설명을 작성해주세요.
- **Step 2)** 조사한 동기화 방식들 중 마음에 드시는(?) 한가지를 골라 직접 구현해보세요 (고른 이유도 간단하게 설명!).
- **Step 3)** Step 2에서 구현한 알고리즘을 #1의 문제에서 pthread_mutex 대신 활용해보세요.
- **Step 4 - Bonus)** 오리지널 pthread_mutex와 여러분의 software lock이랑 프로그램의 성능 비교를 해보는 것도 좋겠네요!
- **기대 결과물)** 소프트웨어 기반 동기화에 관한 내용들, 내용이 반영된 procon2.c, 이에 대한 설명 또는 레포트

- **검색 키워드)** "운영체제 동기화 알고리즘 예제"
- **참고사항)** 아마 실제로 만들어 보시면 동기화가 제대로 안 이루어 질 수도 있습니다. 이것은 여러분들이 잘 못하신게 아니라, 컴퓨터가 잘못해서 그런거니 전혀 걱정안하셔도 됩니다; 최신 CPU에서 성능을 극대화 하기 위해 명령어들의 실행 순서를 마음대로 바꿔버려서 동기화 코드가 순서대로 작동하지 않아서 그런겁니다 (이를 비순차실행 이라고 합니다.)

  이런 현상을 막기 위해서, 여러분들이 작성할 lock/unlock함수의 도입부분과 끝부분에 asm ("mfence") 라는 명령어를 추가하면 됩니다. 비순차실행을 막아버리는 명령어에요!

  아래 예제는 해당 어셈블리 명령어를 #define을 통해 재정의하여 추가한 예제를 나타냅니다. 이렇게 하시면 동기화가 잘 될겁니다 :)

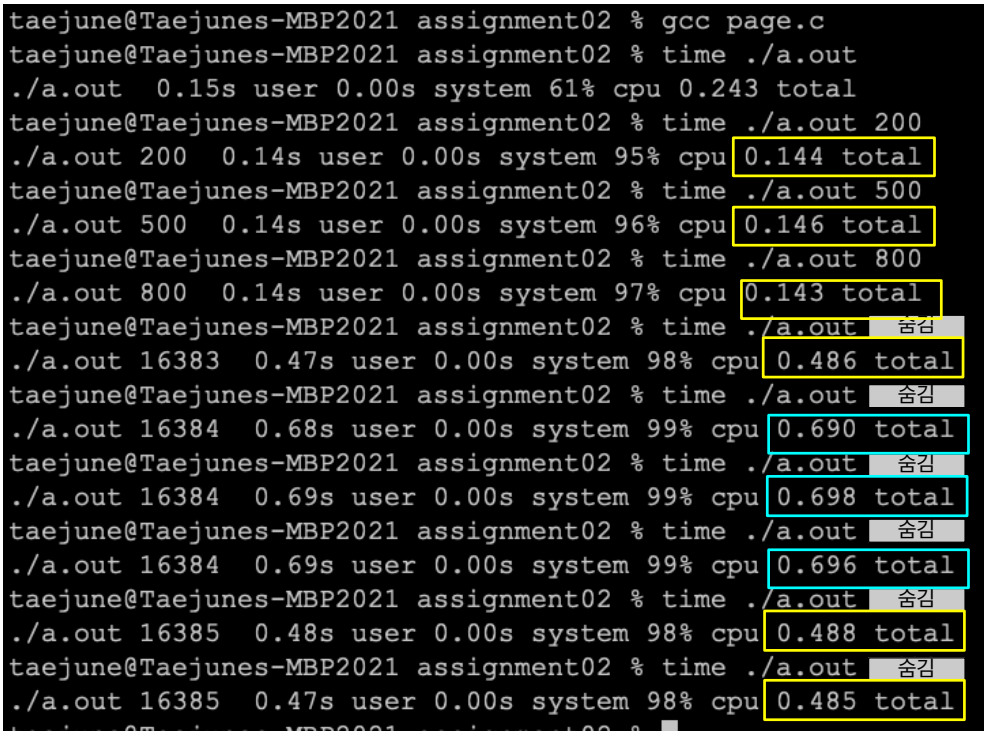
```
// memory barrier for ensuring sequencial processing.
#define barrier() asm ("mfence")

void software_lock(){
  barrier();  // <-- add here

  while(...){
    blabla
  }

  barrier();  // <-- add here
  return;
}
```

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

(여기서 설명하는 page fault의 경우 곧 수업시간에서 배울 demand paging 에서 나옵니다.)

- 사실 메모리쪽 과제는 어떻게 내어드려야 할지 고민이 많았습니다. 이왕 공부하시는거, 여러분들께 실질적으로 도움이 되는 문제를 내드리고 싶은데... 요즘은 메모리라는 영역을 극도로 추상화해서 사용자에게 보여주다보니, 이런 문제를 고안해 내는 것이 영 쉽지 않았습니다. 따로 debugger 등을 활용하지 않는 이상 직접적으로 마주칠 일이 잘 없거든요. 생각해보세요. 당장 여러분들이 친숙하실 JAVA, Python등에서는 메모리를 직접적으로 다룰 일이 잘 없거니와, Rust와 같은 최신 언어는 애초에 Memory-safety가 주요 feature중 하나입니다. 그렇다고, 메모리를 살펴보기 위해 debugger를 직접 사용하자니... 아마 제가 오래오래 살게될거 같습니다. :)

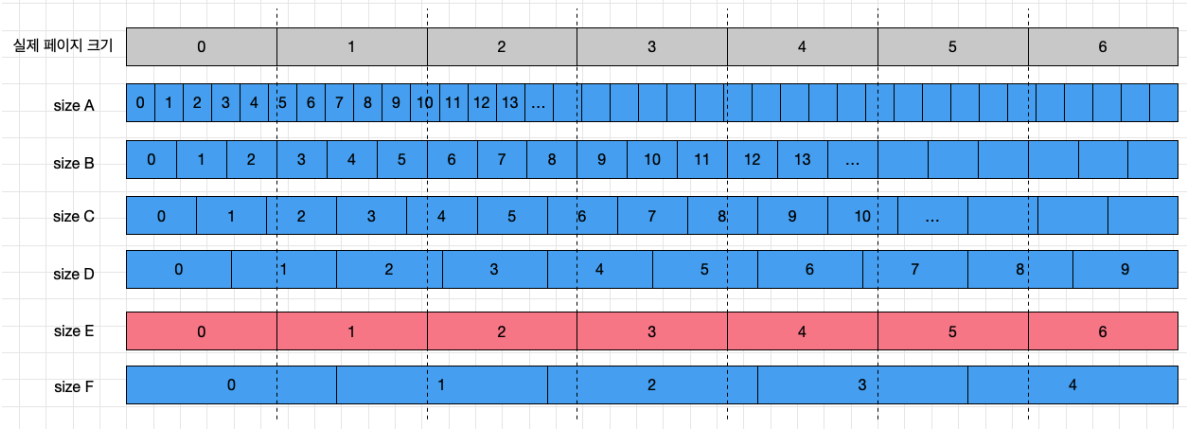
그래서, 간단하게 여러분 컴퓨터의 페이지 하나의 크기를 '직접' 알아보는 내용을 다뤄보고자 합니다. 첨부된 page.c는 큰 heap영역을 할당받고, 특정 주소들을 매 'pagesize' 변수의 크기만큼 점프를 뛰면서 약 100회 참조하는 코드입니다. 예를 들어, 처음 할당받은 주소가 123410000이고, pagesize 변수값이 100이라면, 12341000, 12341100, 12341200, 12341300, ... 와 같은 위치로 참조를 하는거죠. 이러한 동작을 TESTLOOP 횟수 만큼 시행을 하는 겁니다. 이 프로그램을 컴파일하고, 실행시킬때 "time" 명령어를 앞에 붙여서 실행을 하면 프로그램의 실행시간을 간단하게 측정할 수 있습니다. 예를 들어, 컴파일 된 실행파일의 이름이 a.out 이라면, "time ./a.out". 추가로, pagesize의 값을 매번 컴파일해서 바꾸는 것이 아니라, 실행할때 인자로 줄 수도 있어요. 예를 들어, "time ./a.out 1000" 이라고 입력을 하면 pagesize의 값이 1000으로 설정되는겁니다. 이런 방식으로 pagesize의 값을 변경해가며, 프로그램의 실행 시간을 측정하다보면 '특정 값'에서 유난히 실행 시간이 길어지는 pagesize가 있습니다. 아래 그림처럼요.



이 크기가, 높은확률로 여러분 컴퓨터에 설정된 페이지의 크기(의 배수배)입니다. 이 값을 찾는 것이 여러분의 과제 입니다! 조금 코딩이 어려우신 분들은 수동으로 찾으실 수 있으실 것이며, 코딩을 잘하시는 분들은 코드를 고치거나 스크립트를 짜서 자동으로 찾을 수 도 있겠네요!

이 문제에 대한 부연 설명을 아래 그림에 빗대어 설명드리자면, 실제 페이지 크기가 회색일때, pagesize를 A-F까지 바꿔가는겁니다. 이때, A-D는 실제 페이지 사이즈보다 작으니, 한 페이지 안에 여러 값들이 있으므로 전체 page fault의 횟수가 더 작을겁니다. 하지만 실제 페이지 크기와 pagesize의 크기가 똑같다면? 매 접근마다 page fault가 날테니, 프로그램 실행시간이 더 길어질

겁니다! F의 경우는 실제 페이지크기보다 더 커진경우인데, 그래도 서로 걸쳐 있는 부분이 있으니 E보다는 또 page fault가 덜 나겠네요!



이 문제의 의의는 여러분들께서 변수 또는 구조체의 크기를 특정한 크기의 배수 단위로 할당하는 것만으로 프로그램의 성능을 효율적으로 끌어올릴 수 있다는 것을 직접 눈으로 보여드리는 겁니다. 꼭 페이지 크기와 관련된 대한 사항이 아니더라도, 여러 프로그램들이 데이터들을 특정 크기의 배수단위로 할당함으로서 최적화를 달성합니다. 이런 것들을 padding이라고 부릅니다. 실제 예를 드리자면, 아래는 리눅스 커널 코드 중 하나인데, 보시면 padding이라는 의름의 의미 없는 15바이트가 구조체 끄트머리에 할당되어있는 것을 보실 수 있죠.

```
68  struct tnetd7200_clock {
69          u32 ctrl;
70          u32 unused1[3];
71  #define DIVISOR_ENABLE_MASK 0x00008000
72          u32 mul;
73          u32 prediv;
74          u32 postdiv;
75          u32 postdiv2;
76          u32 unused2[6];
77          u32 cmd;
78          u32 status;
79          u32 cmden;
80          u32 padding[15];
81  };
```

어쨋든, 한번쯤은 이런 것을 직접 눈으로 확인 해 보는게 여러분들에게 도움이 되지 않을까 생각이 듭니다 :) 마지막으로, 리눅스에서는 실제 컴퓨터의 page size를 확인하는 명령어가 있습니다. 그 명령어가 무엇인지도 찾아보시고, 여러분이 찾은 값과 일치하는지 비교를 해보시는 것도 좋겠습니다!

- **Step 1)** page.c 코드를 컴파일하고 실행시켜보세요. 실행시킬때 time 명령어와 함께 실행하여 실행 시간을 측정해보세요 (e.g., time ./a.out)
- **Step 2)** 컴파일된 파일은 pagesize 변수를 인자값으로 받습니다. 값을 변경해가며 실행시간의 변화를 확인해보세요 (e.g., time ./a.out 1024)

- **Step 3)** 입력 값이 특정 값에 이르면 (i.e., 실제 page size) 갑자기 실행시간이 확 증가합니다! 이를 통해 여러분 컴퓨터의 페이지 크기를 확인해보세요. 혹시 시간 값의 차이가 잘 보이지 않거나, 실행에 이상이 있으시면 #define된 각종 값들을 변경해보세요.
- **Step 4)** 리눅스(or macOS)에서 여러분 컴퓨터에 설정된 페이지 크기를 확인하는 명령어가 무엇인지 찾아보시고, 여러분이 찾아낸 값과 일치하는지 비교해보세요. 거꾸로 실제 값을 확인해놓고 값을 찾아보는것도 좋은 방법이 되겠네요. 예를 들어 실제 값이 1000이라면, 999, 1000, 1001 이렇게 실행시키면서요!
- **기대 결과물)** 찾아낸 pagesize에 대한 실행시간 캡쳐, 명령어를 통해 확인한 실제 page 크기 캡쳐, 관련 설명이 담긴 레포트
- **검색 키워드)** "check linux page size"