과제 2번

183005 임채승

#1. 생산자-소비자롤 구성된 응용프로그램 만들기

thraed를 생성하기 전에 초기화를 하고, 다 쓴 다음 destory를 해준다.

```
pthread_mutex_init(&critical_section, NULL); // init mutex
...
sem_init(&semWrite, 0, 4); // 세마포어 write를 4로 초기화
sem_init(&semRead, 0, 0); // 세마포어 read를 0으롤 초기화
...
sem_destory(&semWrite);
sem_destory(&semRead);
```

생산자에서 `void mywrite(int n)`가 로직 부분입니다. 여기서 생산하는 부분에서는 `pthread_mutex`로 다른 쓰레드에서 값을 접근하게 만들면 안됩니다.
생산하는 부분이므로 세마포어 write를 기다려야 하며(쓰려면 기다려야 합니다), 만약 큐에 데이터를 추가했다면, 세마포어 read를 생성하면 됩니다(read를 쓸 수 있게 생성합니다). 코드로 다음과 같이 설명 할 수 있습니다.

```
void mywrite(int n)
{
  sem_wait(&semWrite);

  pthread_mutex_lock(&critical_section);
  queue[wptr++] = n;
  if (wptr == N_COUNTER)
    wptr = 0;
  pthread_mutex_unlock(&critical_section);

  sem_post(&semRead);
}
```

소비자에서 `void myread()`가 로직 부분입니다. 생산자와 마찬가지로 데이터를 다룰 때 `pthread_mutex`를 이용해서 다른 쓰레드가 데이터를 다루지 못하도록 합니다.
소비하는 부분이므로 세마포어 read를 기다려야 하며(읽으려면 기다려야 합니다) 만약 큐에서 데이터를 읽어왔다면, 세마포어 write를 생성하면 됩니다(생산자에서 쓸 수 있게 해줘야 합니다).

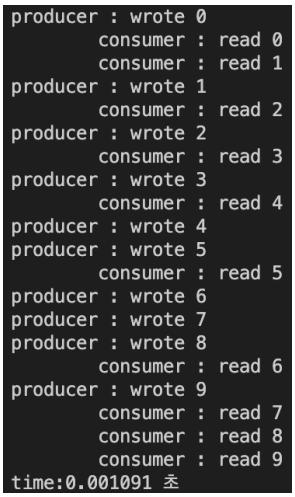
```
int myread()
{
  int n;
  sem_wait(&semRead);

  pthread_mutex_lock(&critical_section);
  n = queue[rptr++];
  if (rptr == N_COUNTER)
    rptr = 0;
  pthread_mutex_unlock(&critical_section);

  sem_post(&semWrite);

  return n;
}
```

실행이 되면 다음과 같이 내용이 나왔습니다.



#2. 소프트웨어로 문을 만드는 방법

Dekker알고리즘

처음 만들어진 소프트웨어 해결방법입니다. 돌아가는 방식은 임계영역으로 들어가기 위해서 자신의 flag를 true로 만든 다음, 다른 것이 임계영역에 있는지 검사합니다. 만약에 자신의 차례가 아니라 다른 쓰레드의 차례라면 자신의 flag를 false로 임시 취소를 한 뒤, 다음 차례를 기다립니다. 그 이후 다시 flag를 true로 만들어 임계영역에 들어갑니다. 모두 마친 후, 자신의 turn을 다른 쓰레드에게 넘겨주고, 자신의 임계영역은 끝났음을 나타내기 위해 flag를 false로 둡니다. 단점은 오직 2개의 쓰레드만 관리 되고, busy wait를 할 수 있습니다.

```
while(1) {
  ...
  flag[i] = true; // 임계영역에 들어가려고 시도
```

```
  while(flag[j]) { // Pj가 임계영역에 있는지 조사
    if(turn == j) { // Pj가 들어갈 기회라면
      flag[i] = false; // 일단 진입 취소
      while(turn == j); // 순서를 기다림
      flag[i] = true; // 재진입 시도
    }
  }
  // 임계영역 (critical section)
  ...
  turn = j; // 진입 순서 양보
  flag[i] = fasle; // 임계역역 사용 완료 지점
  ...
}
```

Peterson알고리즘

Dekker 알고리즘에서 좀 더 간단한게 줄인 버젼 입니다. 자신은 들어가려고 시도한 다음, 바로 진입 순서를 상대방에게 줍니다. 만약 자기 차례가 아니라 다른 쓰레드의 차례라면 대기하다가, 다른 쓰레드가 끝마칠때 임계영역에 들어갑니다. 작업을 다했다면 flag를 false로 바꿉니다. 쓰레드를 여러개 2개 이상 쓸 수 있지만, busy wait를 할 수 있습니다.

```
while(1) {
  ...
  flag[i] = true; // 임계영역에 들어가려고 시도
  turn = j; // 상대방에 진입 기회 양보
  while(flag[j] && turn == j); // 상대방이 진입하려고 한다면 대기
  // 임계영역(critical section)
  ...
  flag[i] = false; // 임계영역 사용 완료 지점
  ...
}
```

Lamport알고리즘

분산처리 환경에 유용한 알고리즘입니다. 각 작업마다 번호를 부여받고 낮은 번호 즉 먼저 온 작업부터 처리합니다. 프로세스들은 한계 대기 조건을 만족하는 것이고, 또한 반드시 들어온 순서대로 수행되어야 하는 것은 아닙니다. 위 Dekker알고리즘과 다르게 2개 이상의 쓰레드에서 쓸 수 있고, Peterson알고리즘과 다르게 busy wait도 없습니다.

```
while(1) {
  ...
  choosing[i] = true; // 번호표 받을 준비
  // 다음 번호를 생성하여 할당
  number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
  choosing[i] = false; // 번호표를 받았음.
  for(j = 0; i < n; i++) { // 모든 프로세스에 대한 번호표 비교 루프
    while(choosing[j]); // 비교할 Pj가 번호표 받을 때까지 대기
    while(number[j] && number[j], j) < (number[i], i));
```

```
    // Pj가 번호표를 갖고 있고 Pj의 번호표가 Pi의 번호표보다
    // 작거나 또는 번호표가 같은 경우 j가 i보다 작다면
    // Pj의 종료(number[j] = 0)까지 대기
  }
  // Critical Section
  ...
  number[i] = 0; // 사용완료로 번호표 취소
  ...
}
```

Pi의 임계영역 진입조건은 모든 프로세스(j(0 ≤ j < n_에 대한 검사를 종결한 시점 즉, 어떤 프로세스도 Pi의 번호표보다 작은 번호표를 갖고 있지 못한 상태입니다. 이 알고리즘은 상호배제, 한계대기, 진행 조건을 모두 만족하는 알고리즘입니다.

Step 2)

Lamport 알고리즘방식을 선택하고 싶습니다. 그 이유는 Dekker알고리즘이나, Peterson알고리즘은 2개로만 표현이 되는데, Lamport는 그보다 더 많은 쓰레드를 관리할 수 있습니다. 알고리즘을 다음과 같이 코드로 쓸 수 있습니다. 추가로 bool을 쓰기 위해 `#incldue<stdbool.h>`도 추가 하였습니다.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define THREAD_COUNT 10
// 초기화 하는 부분입니다.
bool choosing[THREAD_COUNT] = {false};
int number[THREAD_COUNT] = {0};

// 먼저 배열 중 가장 큰 값을 구합니다.(우선순위를 뒤에 붙이기 위해서)
int number_max(int* numbers, int count) {
  int max = 0;

  for (int i = 0; i < count; i++){
    if (max < numbers[i]) max = numbers[i];
  }

  return max;
}

void lock(int thread) {
  // 선택하기 전에 true로 다른 쓰레드에서 들어가는 것을 막습니다.
  choosing[thread] = true;
  // 우선순윅 가장 큰 것으로 넣습니다.
  number[thread] = number_max(number, THREAD_COUNT) + 1;
  // 선택이 끝났기 때문에 다른 쓸레드에서 number에 데이터를 넣는 것을 허용합니다.
  choosing[thread] = false;
  for (int j = 0; j < THREAD_COUNT; j++) {
    // choosing을 선택하고 있는 경우는 기다립니다.
```

```
    while(choosing[j]);
    // 우선순위대로 실행 할 수 있도록 기다립니다. (0은 우선순위가 없는 것이다)
    while(number[j] && (number[j] < number[thread]));
  }

  return;
}

// 다 썼다면 0으로 초기화 합니다.
void unlock(int thread) {
  number[thread] = 0;

  return;
}
```

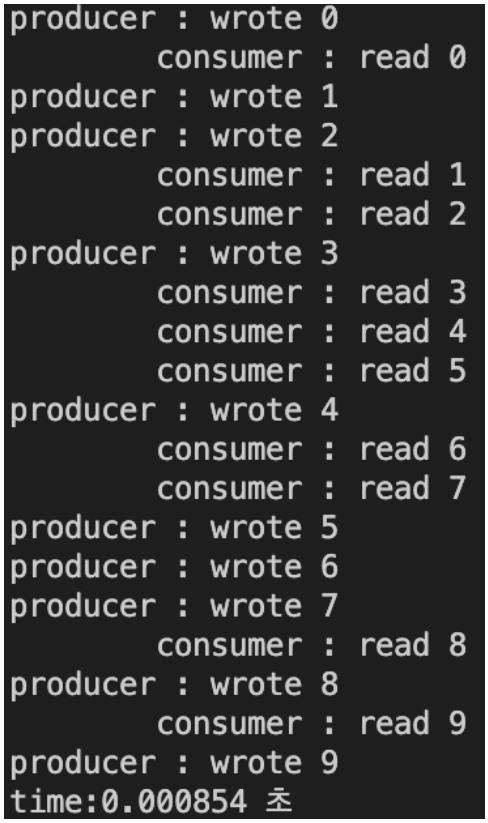
Step3)

위의 코드를 procon2.c에 넣은 다음 pthread_mutex대신 넣어둡니다.
`void mywrite(int n)`과 `void myread()`를 다음과 같이 고칩니다.

```
...
lock(1);
queue[wptr++] = n;
if (wptr == N_COUNTER)
  wptr = 0;
unlock(1);
...
```

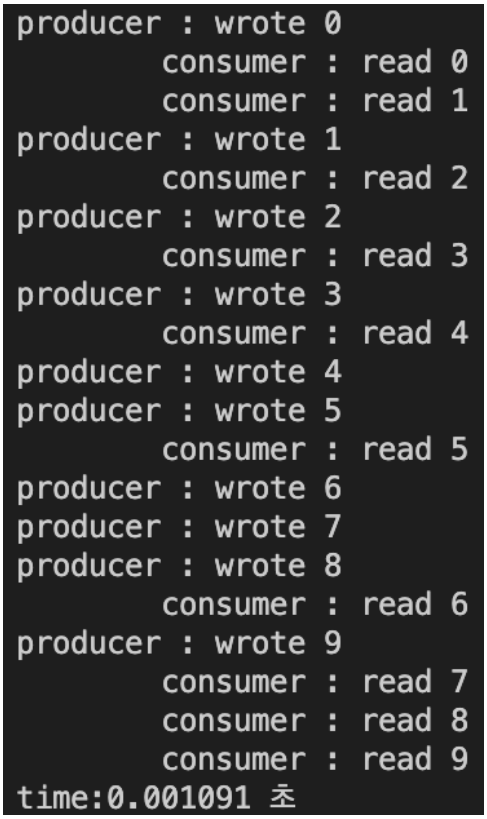
```
...
lock(0);
n = queue[rptr++];
if (rptr == N_COUNTER)
  rptr = 0;
unlock(0);
...
```

실행을 하면 다음과 같이 결과값이 잘 나옵니다.

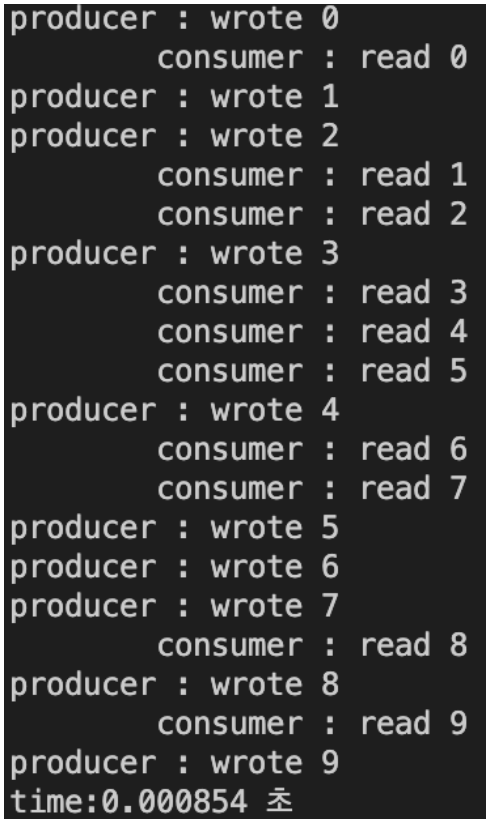


Step 4)

기존에 있던 Mutex와 Lamport알고리즘을 이용한 lock 방식의 차이입니다. time을 측정할 때 wait에 영향을 받는 time을 구하는 코드를 작성했으나, 성능 차이를 비교한다는 점에서 코드를 수정하지 않았습니다.

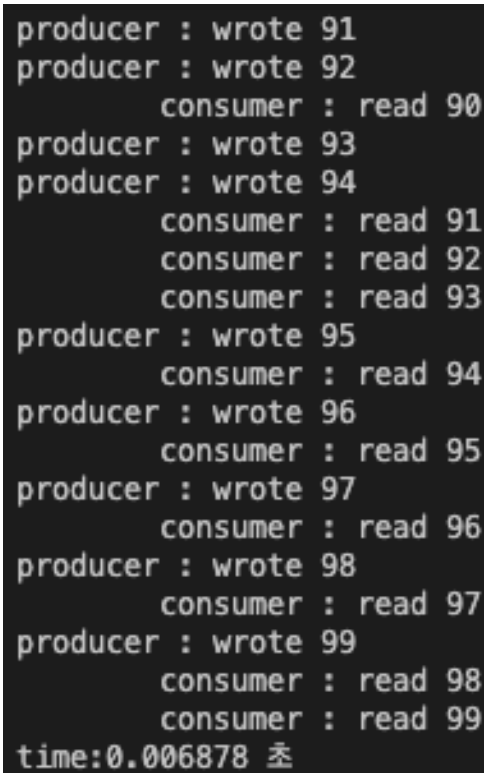


Mutex

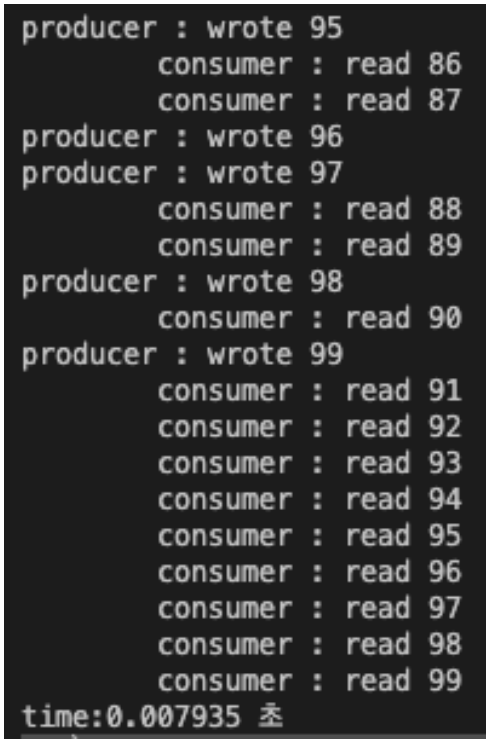


Lamport알고리즘

단순히 for문을 10번 반복 했을 때, 위 사진에서 비교하면 속도차이는 Mutex보다 Lamport가 좀더 빠르게 나왔습니다. 평균적으로 Lamport가 좀 더 빠르게 나왔습니다. Mutex를 잘 만들었을 것이라고 생각하고 개수를 늘렸습니다.

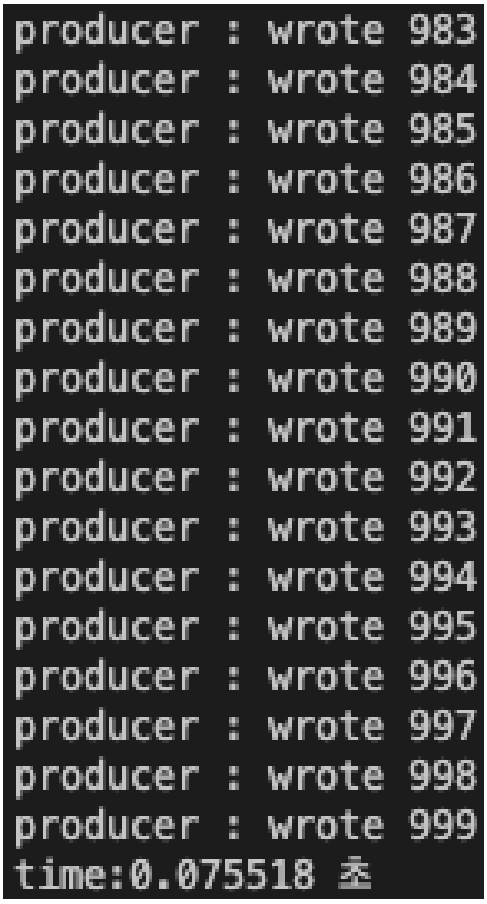


Mutex

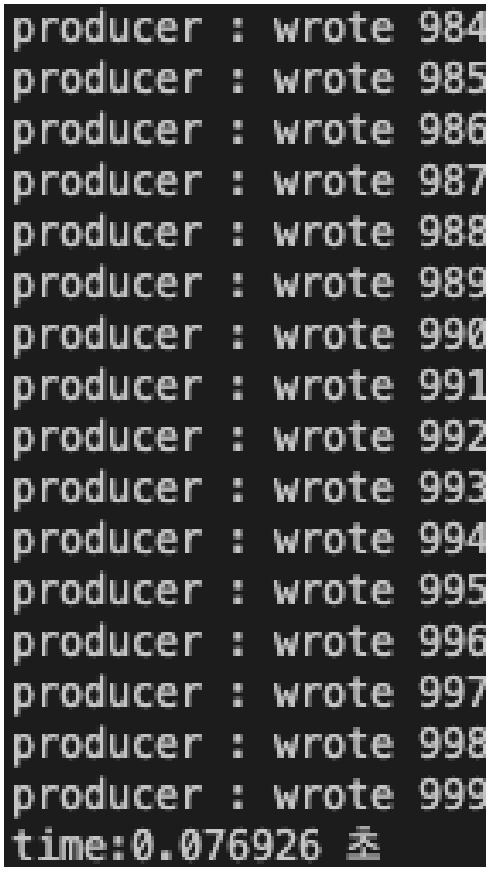


Lamport알고리즘

여기서 양을 100개 즉 10배로 늘려보았습니다. 속도차이는 위와 같이 확연하게 차이가 나왔습니다. Mutex가 Lamport보다 약 0.001초 더 빠르게 나왔습니다.



Mutex

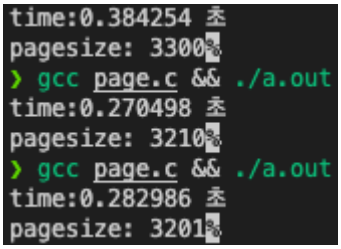


Lamport 알고리즘

위와 같이 1000개 즉 100배 늘리면 약0.001초 나왔습니다. 100개를 한 결과와 차이가 나지 않는 것으로 보아, Lamport로 구현된 내용과, Mutex로 구현된 내용의 성능차이는 크게 나지 않은 것 같습니다.

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

페이지 크기는 컴퓨터가 만든 페이지가 넘어가는 순간부터 커진다고 하였습니다. 제 맥북의 페이지 측정 속도는 약 0.2초 미만이므로 페이지를 측정할 때마다 0.25초 이상일 때로 생각을 하고 코드를 짜 측정해 보았습니다. 결과는 다음과 같이 pagesize가 3300일때로 측정이 되었습니다.



2n일꺼라고 생각한 저는 코드로 직접 page를 찾아보았습니다(맥북에서는 `#include <unistd.h>`를 넣은 다음 `getpagesize()`로 구합니다). 이에 답은 제 예상과 같이 2n인 4096이 나오게 되었습니다.

그러면 page fault가 나서 생기는 오류라고 생각하고 4090~부터 4100까지 검사해보았습니다.

```
time:0.139720 초
pagesize: 4093
--------------
time:0.166009 초
pagesize: 4094
--------------
time:0.629430 초
pagesize: 4095
--------------
time:0.950739 초
pagesize: 4096
--------------
time:0.609390 초
pagesize: 4097
--------------
time:0.183406 초
pagesize: 4098
--------------
time:0.137949 초
pagesize: 4099
```

결과는 위와과 같이 4094까지 잘 검사하다가 갑자기 4095때는 3~4배 가까이 시간초가 올라가고, 4096때는 8~9배 가까지 시간초가 띄는 결과를 볼 수 있었습니다. 다시 4097때는 4095때와 같이 3~4배 가까이 시간초 차이가 나며, 그 이후는 다시 원상 복귀가 되는 것을 알 수 있었습니다. 이에 추측을 하던데 2n의 배수면 page fault가 많이 나는 것이 아닐까 생각이 되어 실험을 하게 되었습니다. (4096이나 그 근처를 많이 지나갈수록 page fault 가 많이 일어남)

```
--------------
time:0.989258 초
pagesize: 2048
--------------
time:0.977525 초
pagesize: 4096
--------------
time:0.930676 초
pagesize: 6144
--------------
```

```
time:1.042952 초
pagesize: 1024
--------------
time:1.011010 초
pagesize: 2048
--------------
time:1.028605 초
pagesize: 3072
```

```
time:0.463652 초
pagesize: 512
--------------
time:0.974279 초
pagesize: 1024
--------------
time:0.499240 초
pagesize: 1536
--------------
time:0.981364 초
pagesize: 2048
```

```
time:0.449486 초
pagesize: 256
--------------
time:0.504747 초
pagesize: 512
--------------
time:0.449672 초
pagesize: 768
--------------
time:0.942967 초
pagesize: 1024
--------------
time:0.452978 초
pagesize: 1280
--------------
time:0.575938 초
pagesize: 1536
```

```
time:0.232009 초
pagesize: 128
--------------
time:0.431940 초
pagesize: 256
--------------
time:0.215710 초
pagesize: 384
--------------
time:0.451294 초
pagesize: 512
--------------
time:0.186824 초
pagesize: 640
--------------
time:0.439782 초
pagesize: 768
```

차근차근 2048부터 128까지 나추면서 실험을 한 결과입니다. 점차 시간초가 줄어드는 것을 볼 수 있습니다. 즉 할당할 때 page와 완벽하게 크기가 같다면 page fault가 많이 날 수 있다는 이야기로 결론이 납니다. 좀 더 해석을 하자면 2048일때는 4096과 비교할 때 2번 중 한 번, 6144 도 또한 2번중 1번 에러가 나기 때문에 시간초가 오래 걸리고 그에 반면에 512는 점차 4096을 맞추지 못하는 것도 확인 할 수 있습니다.