2. Synchronization and memory

192293 고분자공학과 탁민경

1.생산자-소비자로 구성된 응용프로그램 만들기

- 1개의 생산자 스레드와 1개의 소비자 스 레드로 구성되는 간단한 응용프로그램을 작성

요구사항

- 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

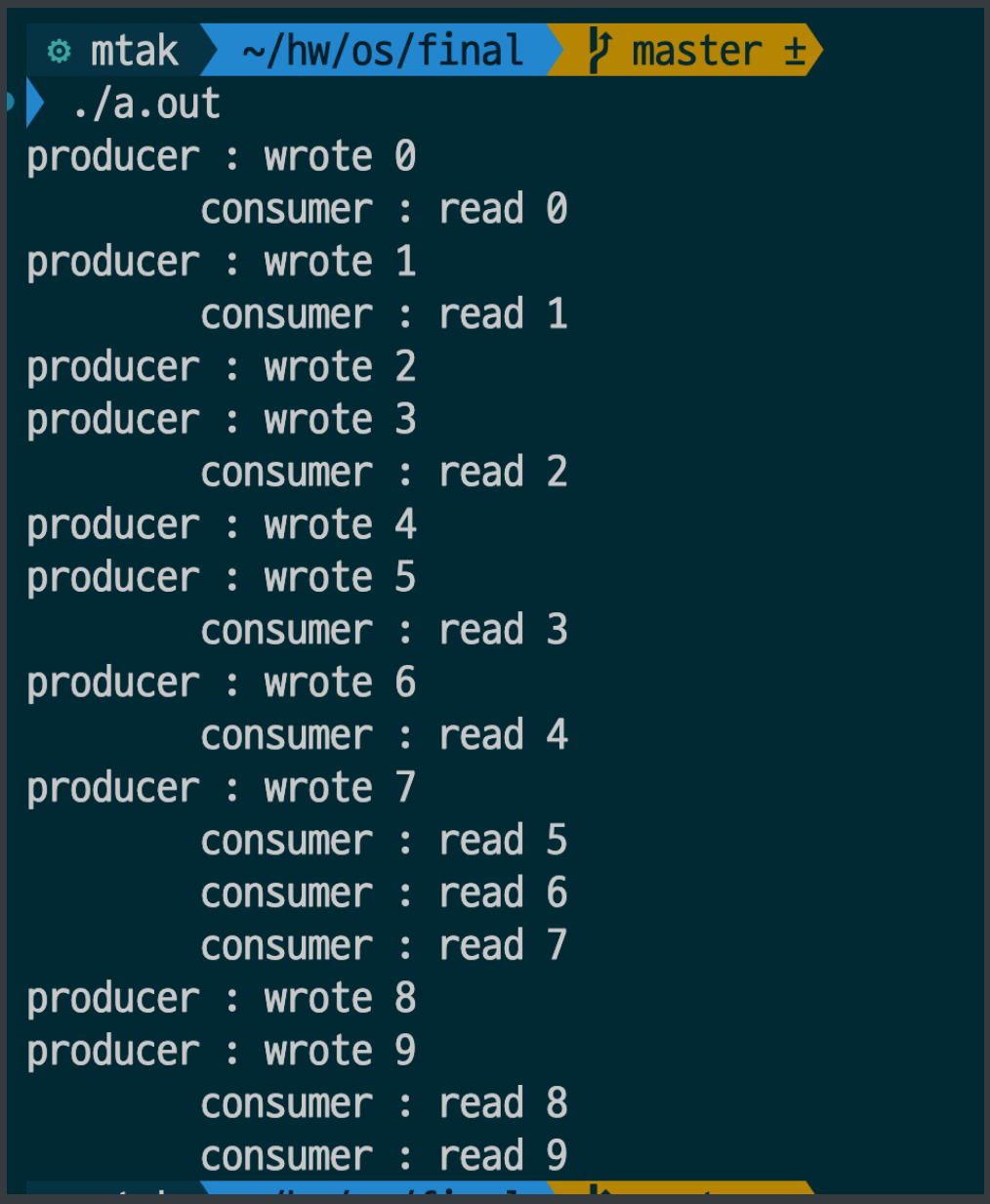
```
1  #include <stdio.h>
2  #include <pthread.h>
```

```c
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
#include <dispatch/dispatch.h>

#define N_COUNTER 4 // the size of a shared buffer
#define MILLI 1000  // time scale

void mywrite(int n);
int myread();

pthread_mutex_t critical_section;   // POSIX mutex
dispatch_semaphore_t semWrite, semRead; // POSIX semaphore
int queue[N_COUNTER];         // shared buffer
int wptr;               // write pointer for queue[]
int rptr;               // read pointer for queue[]

// producer thread function
void *producer(void *arg)
{
  for (int i = 0; i < 10; i++)
  {
    mywrite(i); /**** write i into the shared memory ****/
    printf("producer : wrote %d\n", i);

    // sleep m miliseconds
    int m = rand() % 10;
    usleep(MILLI * m * 10); // m*10
  }
  return NULL;
}

// consumer thread function
void *consumer(void *arg)
{
  for (int i = 0; i < 10; i++)
```

```
39    {
40      int n = myread(); /**** read a value from the shared memory ****/
41      printf("\tconsumer : read %d\n", i);
42
43      // sleep m miliseconds
44      int m = rand() % 10;
45      usleep(MILLI * m * 10); // m*10
46    }
47    return NULL;
48  }
49
50  // write n into the shared memory
51  void mywrite(int n)
52  {
53    dispatch_semaphore_wait(semWrite, DISPATCH_TIME_FOREVER);
54    pthread_mutex_lock(&critical_section);
55    queue[wptr] = n;
56    wptr = (wptr + 1) / N_COUNTER;
57    pthread_mutex_unlock(&critical_section);
58    dispatch_semaphore_signal(semRead);
59  }
60
61  // write a value from the shared memory
62  int myread()
63  {
64    int n;
65    dispatch_semaphore_wait(semRead, DISPATCH_TIME_FOREVER);
66    pthread_mutex_lock(&critical_section);
67    n = queue[rptr];
68    rptr = (rptr + 1) / N_COUNTER;
69    pthread_mutex_unlock(&critical_section);
70    dispatch_semaphore_signal(semWrite);
71    return n;
72  }
73
74  int main()
```

```
75  {
76    pthread_t t[2]; // thread structure
77    srand(time(NULL));
78
79    //wptr, rptr 초기화
80    wptr  = 0;
81    rptr = 0;
82    pthread_mutex_init(&critical_section, NULL); // init mutex
83
84    // init semaphore
85    semWrite = dispatch_semaphore_create(4);
86    semRead = dispatch_semaphore_create(0);
87
88    // create the threads for the producer and consumer
89    pthread_create(&t[0], NULL, producer, NULL);
90    pthread_create(&t[1], NULL, consumer, NULL);
91
92    for (int i = 0; i < 2; i++)
93      pthread_join(t[i], NULL); // wait for the threads
94
95    // destroy the semaphores
96    dispatch_release(semWrite);
97    dispatch_release(semRead);
98
99    pthread_mutex_destroy(&critical_section); // destroy mutex
100   return 0;
101 }
102
```

임계영역(write, read)에 들어가기 전에 mutex로 락을 해당 영역의 runtime이 겹치지 않게 해 주었고, write하기 전 semWrite wait로 다른 쓰레드가 write하지 못하도록 막았고, write한 이후 semRead signal로 대기하고 있던 read쓰레드를 깨웠다.
read thread는 read전 semRead로 다른 쓰레드가 read하지 못하게 막았고, read한 이후 semWrite signal로 대기하고 있던 write 쓰레드를 깨웠다.



2. 소프트웨어로 문을 만드는 방법

1. 소프트웨어 기반 동기화 알고리즘

01-1 데커 알고리즘

- 두 개의 프로세스 간 상호 배제를 보장하는 최초의 알고리즘
- **Flag, Turn** 활용 : Flag의 t/f 여부에 따라 while문 진입을 결정, while문 내부에서 turn에 따라 flag를 바꾸거나 cs 진입

- flag는 누가 CS(Critical Section)에 진입할 것인지 알리는 변수이고, turn은 누가 CS에 들어갈 차례인지 알리는 변수입니다.

```
while(1) {                  // 프로세스i의 진입 영역
  flag[i] = true;           // 프로세스i가 임계구역에 진입하기 위해 진입을 알림
  while(flag[j]) {          // 프로세스j가 임계구역에 진입하려고 하는지 확인
    if (turn == j) {        // 프로세스j가 임계구역에 있다면
      flag[i] = false;      // 프로세스i가 진입을 취소하고
      while (turn == j); // 프로세스i의 차례가 올 때까지 대기 함.
      flag[i] =true;        // 차례가 넘어왔다면 진입을 알림.
    }
  }
// Critical Section
  turn = j;                 // 임계구역의 사용이 끝났다면 차례를 프로세스j에게 넘김.
  flag[i] = false;          // 진입을 취소하여 임계구역 사용완료를 알림.
}
```

01-2 피터슨 알고리즘

- 데커 알고리즘보다 간단하게 구현할 수 있는 알고리즘
- while문의 밖에서 flag,turn을 확인하고, 상대에게 차례를 **양보**함 (=늦게 양보한 애가 나중에 들어감)

```
while(1) {                          // 프로세스i의 진입 영역
  flag[i] = true;                   // 프로세스i가 임계구역에 진입하기 위해 진입을 알림.
  turn = j;                         // 프로세스j에게 진입을 양보함.
                                    //  (두 프로세스중 먼저 양보한쪽이 먼저 임계구역에 들어가게 됨.)
  while (flag[j] && turn = j);      // 프로세스i의 차례가 될 때까지 대기를 함.
// critical section
  flag[i] = false                   // 임계구역 사용완료를 알림.
}
```

01-2 다익스트라 알고리즘

- flag의 상태를 3가지로 정의함 : idle, want-in, in-CS

- **idle** : 진입 시도를 하지 않을 때 있을 곳 / 진입이 끝났을 때 도달하는 곳
- **want-in** : 진입하기위한 첫 번째 단계 - 들어갈 의사를 밝히는 곳
- **in-CS** : 진입하기위한 두 번째 단계 - 진입 직전에 머무르는 곳

이 진행 과정을 단계별로 간단히 살펴보면

1. flag[프로세스 i] 상태를 want-in으로
2. turn을 확인 : 만약 i의 turn이 아니면, 현재 turn인 프로세스가 idle 상태가 될 때까지 돌기(while문)
3. i의 차례가 되면 flag[프로세스 i] 상태를 want-in에서 in-CS로 변경
4. flag를 돌면서 만약 in-CS 상태인 프로세스가 없다면 while문을 빠져나와 i 가 CS로 들어가고, 만약 in-CS 상태인 프로세스가 존재한다면 또 계속 돌이(while문)

흐름을 잘 살펴보면, while 문에서 빠져나오는 순서는 따로 정해져 있는 게 아니다. 따라서 대기 중인 상태에서 계속 while문을 돌아야 하는 Busy waiting 문제가 발생하거나, preemption 문제도 생길 수 있음을 알 수 있다.

```
while(1) {                                  // 프로세스i의 진입 영역
  do {
  // 임계구역 진입시도 1단계
    flag[i] = want-in                       // 1단계 진입시도를 한다고 알림
    while (turn != i ) {                    // 자신의 차례가 될 때까지 대기를 함.
      if (flag[turn] == idle) {             // 임계구역에 프로세스가 없다면,
        turn = i;                           // 자신의 차례로 변경함.
      }
    }
  // 임계구역 진입시도 2단계
    flag[i] = in-CS                        // 임계구역에 진입하겠다고 알림.
    j = 0;
    while ((j < n) && (j == i|| flag[j] != in-CS ){ // 자신을 제외한 in-CS 상태의 프로세스가 있는지 검사 함.
      j = j + 1;
    }
```

```
16    } while(j < n)   // 자신 외에 2단계 진입을 시도하는 프로세스가 있다면 다시 1단계로 돌아감.
17    // critical section   // in-CS 상태의 프로세스가 자신밖에 없다면 임계영역에 진입함.
18    flag[i] = idle;  // 임계구역 사용완료를 알림.
19  }
```

01-3 램퍼드의 베이커리 알고리즘 (Lamport's bakery algorithm)

- 프로세스 n개의 상호 배제 문제를 해결한 알고리즘입니다.
- bakery 알고리즘은 프로세스에게 고유한 번호를 부여하고, 번호를 기준으로 우선순위를 정하여 우선순위가 높은 프로세스가 먼저 임계 구역에 진입하도록 구현되었습니다.
- 번호가 낮을수록 우선순위가 높음.

```
1  while(1) {              // 프로세스i의 진입 영역
2    choosing[i] = ture;   // 번호표 받을 준비
3    number[i] = max(number[0], number[1], ..., number[n-1]) + 1;  // 번호표 부여
4                                                                  // (번호표 부여중 선점이 되어 같은 번호를 부여 받는 프로세스가 발생할 수 있음)
5    chossing[i] = false;  // 번호표를 받음
6    for (j = 0; j < n; j++) { // 모드 프로세스와 번호표를 비교함.
7      while (choosing[j]);    // 프로세스j가 번호표를 받을 때까지 대기
8      while ((number[j] != 0) &&
9              ((number[j] < number[i])                // 프로세스 j가 프로세스 i보다 번호표가 작거나(우선순위가 높고)
10             || (number[j] == number[i] && j < i)); // 또는 번호표가 같을 경우 j 가 i 보다 작다면
11                                                     // 프로세스 j가 임계구역에서 나올 때까지 대기.
12   }
13 // Critical Section
14   number[i] = 0;  // 임계구역 사용완료를 알림.
15 }
```

2. 1.을 Dekker로 구현

가장 원시적인 형태이기 때문에 Dekker로 구현했다.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
#include <dispatch/dispatch.h>

#define N_COUNTER 4 // the size of a shared buffer
#define MILLI 1000  // time scale

void mywrite(int n);
int myread();

bool critical_section = false;    // POSIX mutex
dispatch_semaphore_t semWrite, semRead; // POSIX semaphore
int queue[N_COUNTER];         // shared buffer
int wptr;               // write pointer for queue[]
int rptr;               // read pointer for queue[]

// producer thread function
void *producer(void *arg)
{
  for (int i = 0; i < 10; i++)
  {
    mywrite(i); /**** write i into the shared memory ****/
    printf("producer : wrote %d\n", i);

    // sleep m miliseconds
    int m = rand() % 10;
    usleep(MILLI * m * 10); // m*10
  }
  return NULL;
```

```
33 }
34
35 // consumer thread function
36 void *consumer(void *arg)
37 {
38   for (int i = 0; i < 10; i++)
39   {
40     int n = myread(); /**** read a value from the shared memory ****/
41     printf("\tconsumer : read %d\n", i);
42
43     // sleep m miliseconds
44     int m = rand() % 10;
45     usleep(MILLI * m * 10); // m*10
46   }
47   return NULL;
48 }
49
50 // write n into the shared memory
51 void mywrite(int n)
52 {
53   dispatch_semaphore_wait(semWrite, DISPATCH_TIME_FOREVER);
54   while (critical_section == true);
55   critical_section = true;
56   queue[wptr] = n;
57   wptr = (wptr + 1) / N_COUNTER;
58   critical_section = false;
59   dispatch_semaphore_signal(semRead);
60 }
61
62 // write a value from the shared memory
63 int myread()
64 {
65   int n;
66   dispatch_semaphore_wait(semRead, DISPATCH_TIME_FOREVER);
67   while (critical_section == true);
68   critical_section = true;
```

```
69    n = queue[rptr];
70    rptr = (rptr + 1) / N_COUNTER;
71    critical_section = false;
72    dispatch_semaphore_signal(semWrite);
73    return n;
74  }
75
76  int main()
77  {
78    pthread_t t[2]; // thread structure
79    srand(time(NULL));
80
81    //wptr, rptr 초기화
82    wptr  = 0;
83    rptr = 0;
84
85
86    // init semaphore
87    semWrite = dispatch_semaphore_create(4);
88    semRead = dispatch_semaphore_create(0);
89
90    // create the threads for the producer and consumer
91    pthread_create(&t[0], NULL, producer, NULL);
92    pthread_create(&t[1], NULL, consumer, NULL);
93
94    for (int i = 0; i < 2; i++)
95      pthread_join(t[i], NULL); // wait for the threads
96
97    // destroy the semaphores
98    dispatch_release(semWrite);
99    dispatch_release(semRead);
100
101   return 0;
102 }
```

3. 내 컴퓨터의 페이지 크기는 얼마일까?

```
⚙ mtak ~/hw/os  master ±
 time  ./page.out 3000
./page.out 3000  0.22s user 0.00s system 99% cpu 0.221 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 3100
./page.out 3100  0.22s user 0.00s system 99% cpu 0.221 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 3200
./page.out 3200  0.56s user 0.00s system 99% cpu 0.563 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 3300
./page.out 3300  0.22s user 0.00s system 99% cpu 0.221 total
```

```
 time  ./page.out 3900
./page.out 3900  0.22s user 0.00s system 99% cpu 0.220 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 4000
./page.out 4000  0.22s user 0.00s system 99% cpu 0.220 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 4100
./page.out 4100  0.24s user 0.00s system 99% cpu 0.244 total
⚙ mtak ~/hw/os  master ±
 time  ./page.out 4200
./page.out 4200  0.22s user 0.00s system 98% cpu 0.221 total
⚙ mtak ~/hw/os  master ±
```

page size가 3200, 4100의 배수일 때마다 실행 시간이 오래 걸렸다.

하지만 실제로 macOS로 페이지 크기를 측정했을 때 다음과 같이 4096이 나왔다.

```
mtak ~/hw/os  master ±
 sysctl -n hw.pagesize
4096
```