Assignment #2. Synchronization

214823 박종현

#1. 생산자-소비자로 구성된 응용프로그램 만들기

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.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
sem_t semWrite, semRead; // POSIX semaphore
int queue[N_COUNTER]; // shared buffer
int wptr = 0; // write pointer for queue[]
int rptr = 0; // 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 milliseconds
    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++) {
    int n = myread(); // read a value from the shared memory
    printf("\tconsumer : read %d\n", n);

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

// write n into the shared memory
void mywrite(int n) {
  sem_wait(&semWrite);  // 세마포어가 가용할 때 까지 대기
  pthread_mutex_lock(&critical_section);  // 임계 영역 진입; "pthread_mutex_t critical_section 뮤텍스 이용"
```

```
49
50    queue[wptr] = n;  // 버퍼에 값 작성
51    wptr = (wptr + 1) % N_COUNTER;  // 포인터 이동, 원형 큐로 동작하므로 % 연산 처리
52
53    pthread_mutex_unlock(&critical_section);  // 임계 영역 이탈
54    sem_post(&semRead);  // 세마포어 반환
55  }
56
57  // read a value from the shared memory
58  int myread() {
59    sem_wait(&semRead);  // 세마포어가 가용할 때까지 대기
60    pthread_mutex_lock(&critical_section);  // 임계 영역 진입
61
62    int n = queue[rptr];  // 버퍼에서 값 읽어오기
63    rptr = (rptr + 1) % N_COUNTER;  // 포인터 이동
64
65    pthread_mutex_unlock(&critical_section);  // 임계 영역 이탈
66    sem_post(&semWrite);  // 세마포어 반환
67
68    return n;
69  }
70
71  int main() {
72    pthread_t t[2]; // thread structure
73    srand(time(NULL));
74
75    pthread_mutex_init(&critical_section, NULL); // init mutex
76
77    // init semaphore
78    sem_init(&semWrite, 0, N_COUNTER);
79    sem_init(&semRead, 0, 0);
80
81    // create the threads for the producer and consumer
82    pthread_create(&t[0], NULL, producer, NULL);
83    pthread_create(&t[1], NULL, consumer, NULL);
84
85    for(int i = 0; i < 2; i++) {
86      pthread_join(t[i], NULL); // wait for the threads
87    }
88
89    // destroy the semaphores
90    sem_destroy(&semWrite);
91    sem_destroy(&semRead);
92
93    pthread_mutex_destroy(&critical_section); // destroy mutex
94    return 0;
95  }
```

#2. 소프트웨어로 문 만드는 방법

Step 1, 2. 동기화 알고리즘 조사 및 구현

1) Dekker 알고리즘

```
1  bool flag[2] = { false, false };  // 진입 대기 여부
2  int turn = 1;  // 우선권이 있는 프로세스
```
C

```c

void proc_0() {
  while (true) {
    flag[0] = true;  // 0번 프로세스 진입 대기
    while (flag[1]) {  // 1번 프로세스 진입 의사가 참인 경우
      /*
        1번 프로세스가 우선인 경우 우선권이 넘어오기 전까지 진입 대기를 풀고 대기한다.
        우선권이 넘어온다면 다시 본 프로세스의 진입 의사를 참으로 설정한다.
      */
      if (turn == 1) {
        flag[0] = false;
        while (turn == 1);
        flag[0] = true
      }
    }

    // 임계영역

    // 처리가 완료된 후 우선권을 상대 프로세스에게 넘기고, 진입 의사를 거짓으로 설정한다.
    turn = 1;
    flag[0] = false;
  }
}

// proc_0과 동일
void proc_1() {}
```

2) Peterson 알고리즘

```c
bool flag[2] = { false, false };
int turn = 1;

void proc_0() {
  while (true) {
    flag[0] = true;  // 0번 프로세스 진입 의사를 참으로 설정
    turn = 1;  // 우선 순위를 상대에게 넘긴다.

    /*
      만약 1번 프로세스의 진입 의사가 참이라면, 거짓이 될 때까지 대기한다.

      1번 프로세스 처리 시: 1번 프로세스에서의 구현도 동일할 것이므로
      1. `flag[1]`은 처리가 종료되면 `false`가 될 것이다.
      2. `turn`은 1번 프로세스에서 처리에 진입하기 전에 0번에게 넘겼을 것임이 보장된다.

      두 판단 요소 모두 1번 프로세스가 작업 처리중임을 나타낸다면, 상황이 변화할 때까지 대기한다.
    */
    while (flag[1] && turn == 1);

    // 임계영역

    // 우선 순위는 이미 상대에게 넘김
    // 진입 의사를 거짓으로 설정
    flag[0] = false;
  }
}

```

```c
// proc_0과 동일
void proc_1() {}
```

3) Dijkstra 알고리즘

```c
#define idle 0
#define request 1
#define processing 2

int flag[2] = { idle, idle };
int turn = 1;

void proc_0() {
  while (true) {
    flag[0] = request;  // 0번 프로세스 상태를 진입 의사 있음(`request`)으로 설정
    /*
      우선순위가 자신이 아니라면 현재 턴의 프로세스 상태를 확인한다.
      만약 현재 턴의 프로세스가 작업을 하지 않고 있다면 턴을 자신으로 설정한다.
    */
    while (turn != 0) {
      if (flag[turn] == idle) {
        turn = 0;
      }
    }

    // 이 부분에 도달함은 턴이 자신으로 지정됨을 보장한다.
    // 현재 프로세스 상태를 처리중으로 변경한다.
    flag[0] = processing;

    // 하지만 만약 다른 프로세스 상태도 처리중라면, 임계영역 진입 전에 프로세스를 중단하고 처음 과정으로 복귀한다.
    if (flag[1] == processing) {
      continue;
    }

    // 임계영역

    // 처리가 끝나면 `idle` 상태로 복원한다.
    flag[0] = idle;
  }
}

// proc_0과 동일
void proc_1() {}
```

4) Lamport의 베이커리 알고리즘

```c
// 우선순위(= 번호표) 요청
request[2] = { false, false };
// 우선순위 (작을수록 우선); 0은 임계영역 진입 소요 없음
priority[2] = { 0, 0 };

void proc_0() {
  request[0] = true;
  // priority[0] = max(priority) + 1;
  priority[0] = priority[1] + 1;
  request[0] = false;  // 우선순위 발급 요청 처리 완료

```

```
  int i = 0;
  for (i = 0; i < 2/* n */; i++) {
    while (request[i]);  // 우선순위 발급 요청이 처리될 때 까지 대기
    while (priority[i] <= priority[0])  // 다른 프로세스의 우선순위가 0번 프로세스보다 높거나 같으면 대기
  }

  // 임계영역

  priority[0] = 0;  // 처리가 끝나면 임계영역 진입 소요 없음으로 복원한다.
}

// proc_0과 동일
void proc_1() {}
```

Step 3. Dijkstra 구현체의 도입

(추가)

```c
#define barrier() asm ("mfence")
#define LOCK_IDLE 0
#define LOCK_REQ  1
#define LOCK_PROC 2

int flag[2] = { LOCK_IDLE, LOCK_IDLE };
int lock_turn = 1;
int lockWrite = 0, lockRead = 1;

void lock_wait(int now) {
  barrier();
  while (true) {
    flag[now] = LOCK_REQ;
    while (lock_turn != now) {
      if (flag[lock_turn] == LOCK_IDLE) {
        lock_turn = now;
      }
    }

    flag[now] = LOCK_PROC;

    // 이 시나리오에 specific한 코드; 다른 프로세스가 LOCK_PROC 중인지 확인
    if (flag[(now + 1) % 2] == LOCK_PROC) continue;

    break;
  }
  barrier();
}

void lock_post(int now) {
  flag[now] = LOCK_IDLE;
}
```

(수정)

```c
// write n into the shared memory
void mywrite(int n) {
  lock_wait(lockWrite);

```

```
    queue[wptr] = n;  // 버퍼에 값 작성
    wptr = (wptr + 1) % N_COUNTER;  // 포인터 이동, 원형 큐로 동작하므로 % 연산 처리

    lock_post(lockWrite);
  }

// read a value from the shared memory
int myread() {
    lock_wait(lockRead);

    int n = queue[rptr];  // 버퍼에서 값 읽어오기
    rptr = (rptr + 1) % N_COUNTER;  // 포인터 이동

    lock_post(lockRead);
    return n;
}
```

Step 4. 성능 비교

#1에서 작성

```
$ time ./1.out
```
Shell

```
0.00user 0.00system 0:00.43elapsed 0%CPU (0avgtext+0avgdata 1648maxresident)k
40inputs+0outputs (1major+75minor)pagefaults 0swaps
```

#2에서 작성

```
time ./2.out
```
Shell

```
0.00user 0.00system 0:00.45elapsed 0%CPU (0avgtext+0avgdata 1832maxresident)k
40inputs+0outputs (1major+78minor)pagefaults 0swaps
```

∴ 오리지널 `ptheread_mutex`의 성능이 더 좋음

#3. 내 컴퓨터의 페이지 크기는 얼마일까?

```
for i in $(seq 0 $STEPS)
do
  time ./a.out `expr $i \* $SIZE_PER_STEP`
done
```
Shell

Step 1.

```
time ./a.out
```
Shell

```
0.20user 0.00system 0:00.20elapsed 99%CPU (0avgtext+0avgdata 1312maxresident)k
0inputs+0outputs (0major+74minor)pagefaults 0swaps
```

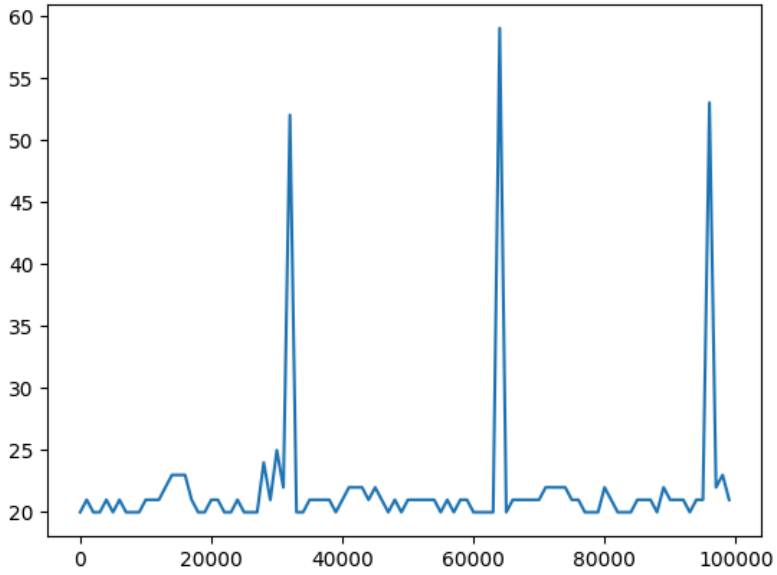
Step 2.

```
time ./a.out 777
```
Shell

```
0.24user 0.00system 0:00.24elapsed 99%CPU (0avgtext+0avgdata 1572maxresident)k
0inputs+0outputs (0major+141minor)pagefaults 0swaps
```

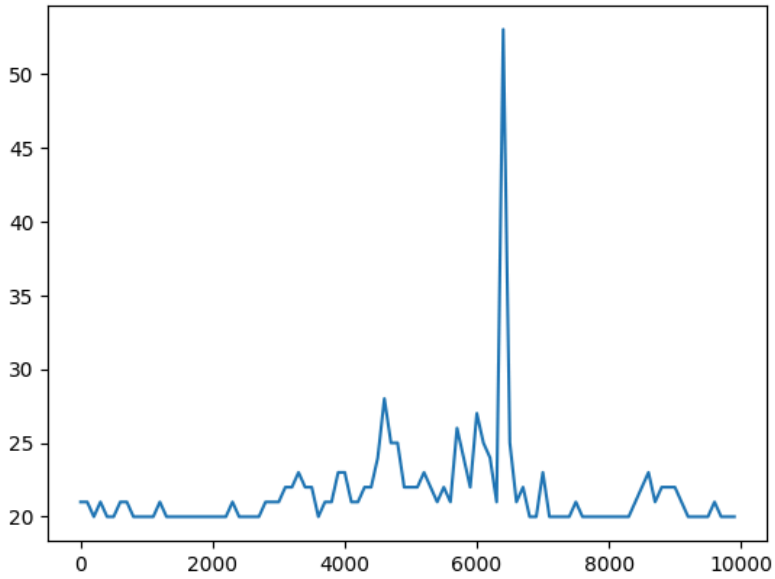
Step 3. 여러 케이스 확인하기

1) 1000 간격으로 100스텝



#	페이지 크기 (argv[1])	러닝 타임
1	32,000	0.52
2	64,000	0.59
3	96,000	0.53

2) 100 간격으로 1000스텝



#	페이지 크기 (argv[1])	러닝 타임
1	64,000	0.58

3) 32,000과 그 주변값 검증

- 32,001

```shell
time ./a.out 32001
```

```
0.20user 0.01system 0:00.21elapsed 98%CPU (0avgtext+0avgdata 13644maxresident)k
32inputs+0outputs (1major+85minor)pagefaults 0swaps
```

- 32,000

```shell
time ./a.out 32000
0.48user 0.00system 0:00.49elapsed 99%CPU (0avgtext+0avgdata 13596maxresident)k
32inputs+0outputs (1major+82minor)pagefaults 0swaps
```

- 31,999

```shell
time ./a.out 31999
```

```
0.22user 0.00system 0:00.22elapsed 98%CPU (0avgtext+0avgdata 15604maxresident)k
32inputs+0outputs (1major+71minor)pagefaults 0swaps
```

4) 추측

```
$ time ./a.out 32000
0.48user 0.00system 0:00.49elapsed 99%CPU (0avgtext+0avgdata 13596maxresident)k
32inputs+0outputs (1major+82minor)pagefaults 0swaps
```

1. 32,000 바이트의 배수인 경우, 실행 시간이 오래 걸림.
2. 32,000 바이트를 시스템의 페이지 크기라고 결론내리기에는 지나치게 값이 큼.
3. 지금까지 시도한 값과 서로소인 어떤 값의 특정 실수배가 32,000에 인접하여 32,000 바이트 경우에 이와 같은 현상이 나타나는 것일 수 있음.

∴ 페이지 크기 설정값은 32,000 바이트와 연관 있는 수이다.

Step 4. 추측 검증하기

```
shapelayer@DESKTOP-T20AQHE:~$ grep -ir pagesize /proc/self/smaps
KernelPageSize:        4 kB
MMUPageSize:           4 kB
```

```
1  grep -ir pagesize /proc/self/smaps
```
Shell

```
1  KernelPageSize:        4 kB
2  MMUPageSize:           4 kB
```

페이지 크기를 4096 바이트로 지정
- 러닝 타임: 1.20초

```
1  time ./a.out 4096
```
Shell

```
1  1.20user 0.00system 0:01.20elapsed 99%CPU (0avgtext+0avgdata 1664maxresident)k
2  0inputs+0outputs (0major+164minor)pagefaults 0swaps
```

페이지 크기를 4095 바이트로 지정
- 러닝 타임: 0.66초

```
1  time ./a.out 4095
```
Shell

```
1  0.66user 0.00system 0:00.66elapsed 99%CPU (0avgtext+0avgdata 3392maxresident)k
2  0inputs+0outputs (0major+87minor)pagefaults 0swaps
```

페이지 크기를 4097 바이트로 지정
- 러닝 타임: 0.67초

```
1  time ./a.out 4097
```
Shell

```
1  0.67user 0.00system 0:00.67elapsed 99%CPU (0avgtext+0avgdata 3768maxresident)k
2  0inputs+0outputs (1major+161minor)pagefaults 0swaps
```