운영체제 과제2

214683 장인환

1, 생산자 – 소비자로 구성된 응용프로그램 만들기

1) mywrite 함수

```
  GNU nano 7.2                                  procon.c
void mywrite(int n) {
  /* [Write here] */
  sem_wait(&semWrite);
  pthread_mutex_lock(&critical_section);

  queue[wptr] = n;
  wptr = (wptr+1) % N_COUNTER;

  pthread_mutex_unlock(&critical_section);
  sem_post(&semRead);
}
```

sem_wait(&semWrite) : semWrite-- 하여 쓸 수 있는 공간이 줄임 -> 쓰기를 통해 함수 실행 여부

pthread_mutex_lock : 뮤텍스 lock

Queue[wptr]에 n 값 추가, wptr은 원형큐에 대한 포인터이므로 원형큐 형식으로 +1

작성후 뮤텍스 언락, sem_post(&semRead) : 읽을 수 잇는 공간 +1

2) myread함수

```
// write a value from the shared memory
int myread() {
  /* [Write here] */
  sem_wait(&semRead);
  pthread_mutex_lock(&critical_section);

  int n = queue[rptr];
  rptr = (rptr +1 ) % N_COUNTER;

  pthread_mutex_unlock(&critical_section);
  sem_post(&semWrite);

  return n;
}
```

Sem_wait(&semRead) : semRead-- 하여 읽을 수 있는 공간 줄임 -> 읽기를 통해 함수 실행 여부 이후 동일

3) 세마포어 초기화

Sem_init(&semWrite, 0, N_COUNT) : 버퍼의 빈공간 4로 초기화

Sem_init(%semRead, 0, 0) : 버퍼 읽을 수 있는공간 0으로 초기화

4) 세마포어 삭제

Sem_destroy(&semWrite), sem_destroy(&semRead) : 읽기, 쓰기 두 가지 모두 삭제

```
// init semaphore
/* [Write here] */
sem_init(&semWrite, 0, N_COUNTER);
sem_init(&semRead, 0 , 0);

// create the threads for the producer and consumer
pthread_create(&t[0], NULL, producer, NULL);
pthread_create(&t[1], NULL, consumer, NULL);

for(int i=0; i<2; i++)
  pthread_join(t[i],NULL); // wait for the threads

//destroy the semaphores
/* [Write here] */
sem_destroy(&semWrite);
sem_destroy(&semRead);

pthread_mutex_destroy(&critical_section); // destroy mutex
return 0;
```

5) 실행장면

```
janginh@test2:~/hw2$ nano procon.c
janginh@test2:~/hw2$ gcc -o procon.o procon.c -lpthread
janginh@test2:~/hw2$ ./procon.o
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
producer : wrote 3
        consumer : read 2
producer : wrote 4
        consumer : read 3
producer : wrote 5
        consumer : read 4
        consumer : read 5
producer : wrote 6
        consumer : read 6
producer : wrote 7
        consumer : read 7
producer : wrote 8
        consumer : read 8
producer : wrote 9
        consumer : read 9
```

2. 소프트웨어로 문 만들기

1) 소프트웨어 기반 동기화 방식

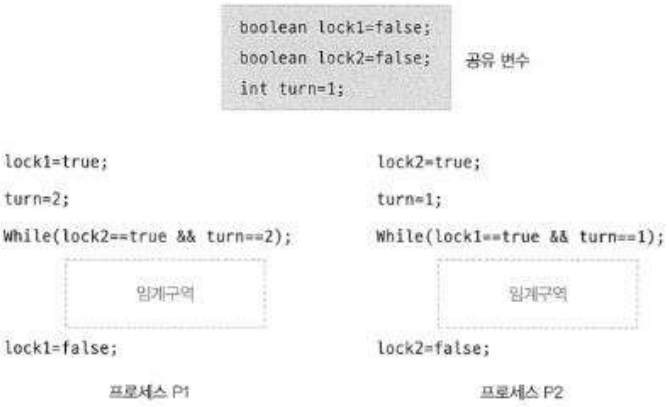
동기화 필요한 이유 : 병형 수행 중 비동기적 프로세스들이 공유자원에 동시 접근시 문제 발생 -> 해결법 : 동기화

동기화 : 협력 프로세스 간의 실행 순서를 정해주는 매커니즘

임계구역(critical section) : 공유 데이터를 접근하는 코드

임계구역 해결법 : 1. 상호배제(다른 프로세스 접근x), 2. 한정대기(종료 대기), 3. 진행동기화 알고리즘

1. 피터슨 알고리즘



프로세스 마다 lock 변수를 가지고 있으며 임계구역에 들어갈 때는 다른 프로세스의 lock 상태를 확인하여 어떤 프로세스가 임계구역에 들어간 상태인지 확인하는 방법이다. P1에서 while 조건이 lock2==true 조건이 앞에 존재하여 lock2가 false이라면 P1은 임계구역에 진입이 가능하다. 이를 통해 한정대기 조건을 만족할 수 있다.

한계점 : 2개의 프로세스만 사용이 가능하다. 여러개의 프로세스 경우 하나의 임계구역 을 사용하기 위해선 공유 변수가 추가 되어야하고 이는 복잡성이 증가하게 된다.

2. 데커 알고리즘

피터슨과 유사하다. 하지만 상대의 lock를 먼저 확인후 양보후 반복하여 확인한다.

한계점 : 피터슨과 동일하게 프로세스가 늘어나면 변수가 늘어나고 동일하게 복잡해진 다.

3. 램포트 빵집 알고리즘

빵집 번호표 뽑기 처럼 번호를 부여하여 작은 순서대로 입장하는 방식, 번호가 같은 시에는 pid를 이용해 순서를 정한다. 하지만 이는 바쁜대기를 하기때문에 cpu 낭비가 일어난다.

한계점 : 배열이 필요하며 프로세스의 수가 고정되어 있다.

2) 한가지 고르기 - 램포트 빵집 알고리즘

이유 : 번호 순서표를 뽑아서 우선순위를 주는것이 수업때 자주 들었던 은행창고 이야기가 떠올라서 고르게 되었다. 그리고 무엇보다 빵을 좋아하기때문에 빵집 알고리즘이라는 단어에 끌리게 되었다.

```
while(1) {
  choosing[i] = ture;   // 번호표를 받지 않음
  number[i] = max(number[0], number[1], ..., number[n-1]

  chossing[i] = false;  // 번호표를 받음

  for (j = 0; j < n; j++) { // 모드 프로세스와 번호표를 비
    while (choosing[j]);    // 프로세스j가 번호표를 받을 (

    while ((number[j] != 0) &&
           ((number[j] < number[i])                  // 프
           || (number[j] == number[i] && j < i))); // 우

  }

// 임계구역 내용

  number[i] = 0;  // 임계구역 사용완료를 알림.
}
```

3) #1의 문제에 적용

Procon6 경우 gcc procon6.c -o procon6 명령어로 만드니 오류가 발생하지 않음

```
janginh@test2:~/hw2$ ./procon6
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
        consumer : read 3
producer : wrote 4
        consumer : read 4
producer : wrote 5
        consumer : read 5
producer : wrote 6
        consumer : read 6
producer : wrote 7
        consumer : read 7
producer : wrote 8
        consumer : read 8
producer : wrote 9
        consumer : read 9
```

찾아보니 프로세스 2개의 낮은 최적화 수준에서는 실행순서를 재정렬을 하지 않아 오류가 발생하지 않는다는 걸 알게 되었다.

오류를 확인해보기 위해 gcc –O2 –o procn9 procon9.c -lpthread를 통해 최적화 옵션을 켜서 강제로 순서 재정렬 -> procon9의 경우 오류 발생

```
janginh@test2:~/hw2$ ./procon9
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
producer : wrote 3
producer : wrote 4
^C
```

Procon10.c 에는 #define barrier를 선언 이후

gcc –O2 –o procon10 procon10.c -lpthread를 통해 오류가 없음을 확인

```
janginh@test2:~/hw2$ gcc -O2 -o procon10 procon10.c -lpthread
janginh@test2:~/hw2$ ./procon10
producer : wrote 0
        consumer : read 0
producer : wrote 1
producer : wrote 2
        consumer : read 1
producer : wrote 3
        consumer : read 2
producer : wrote 4
        consumer : read 3
producer : wrote 5
        consumer : read 4
producer : wrote 6
        consumer : read 5
producer : wrote 7
        consumer : read 6
producer : wrote 8
        consumer : read 7
        consumer : read 8
producer : wrote 9
        consumer : read 9
```

<procon2~5,7,8의 경우 코드 수정 과정이므로 따로 첨부하지 않이함>

4) 성능 비교 (세마포어 vs 빵집 알고리즘)

< 세마포어 > <빵집 알고리즘>

```
janginh@test2:~/hw2$ time ./procon
producer : wrote 0
        consumer : read 0
producer : wrote 1
producer : wrote 2
producer : wrote 3
        consumer : read 1
producer : wrote 4
        consumer : read 2
        consumer : read 3
producer : wrote 5
        consumer : read 4
producer : wrote 6
        consumer : read 5
producer : wrote 7
producer : wrote 8
        consumer : read 6
producer : wrote 9
        consumer : read 7
        consumer : read 8
        consumer : read 9

real    0m0.571s
user    0m0.003s
sys     0m0.000s
```

```
janginh@test2:~/hw2$ time ./procon6
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
        consumer : read 2
producer : wrote 3
        consumer : read 3
producer : wrote 4
        consumer : read 4
producer : wrote 5
        consumer : read 5
producer : wrote 6
        consumer : read 6
producer : wrote 7
        consumer : read 7
producer : wrote 8
producer : wrote 9
        consumer : read 8
        consumer : read 9

real    0m0.648s
user    0m0.081s
sys     0m0.002s
```

결론 : 세마포어를 사용하는 코드쪽이 성능이 더 뛰어나다

3. 내컴퓨터 페이지 크기 확인

1) time과 함께 page.c 실행

```
janginh@test2:~/hw2$ time ./a.out

real    0m0.081s
user    0m0.080s
sys     0m0.001s
```

2) pagesize 인자값을 받고 실행시간 변화 확인

```
janginh@test2:~/hw2$ time ./a.out 64

real    0m0.091s
user    0m0.090s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 128

real    0m0.157s
user    0m0.155s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 256

real    0m0.396s
user    0m0.395s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 512

real    0m0.492s
user    0m0.490s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 1024

real    0m0.587s
user    0m0.585s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 2048

real    0m0.551s
```

```
janginh@test2:~/hw2$ time ./a.out 300

real    0m0.103s
user    0m0.101s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 400

real    0m0.088s
user    0m0.087s
sys     0m0.000s
janginh@test2:~/hw2$ time ./a.out 500

real    0m0.102s
user    0m0.100s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 600

real    0m0.088s
user    0m0.087s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 700

real    0m0.084s
user    0m0.084s
sys     0m0.000s
```

100의 단위로 증가 되었을때는 실행시간이 달라지지 않았으나 2의 제곱수 256 이상부터는 실행시간이 확 증가 되었습니다.

3) 페이지 크기 확인

```
janginh@test2:~/hw2$ time ./a.out 256

real    0m0.457s
user    0m0.456s
sys     0m0.001s
janginh@test2:~/hw2$ time ./a.out 512

real    0m0.506s
user    0m0.505s
sys     0m0.000s
janginh@test2:~/hw2$ time ./a.out 1024

real    0m0.584s
user    0m0.581s
sys     0m0.002s
janginh@test2:~/hw2$ time ./a.out 2048

real    0m0.571s
user    0m0.569s
sys     0m0.002s
janginh@test2:~/hw2$ time ./a.out 4096

real    0m0.562s
user    0m0.561s
sys     0m0.001s
janginh@test2:~/hw2$
```

256부터 4096까지 2의 제곱수로 했을때 1024까지는 조금씩 올라가는 현상을 나타남으로써 int형인 만큼 4byte * 1024로 4096byte라는 것을 알 수 있습니다.

4) 확인 명령어 및 예측값 비교

```
janginh@test2:~/hw2$ getconf PAGE_SIZE
4096
```

getconf PAGE_SIZE를 통해 페이지 용량을 알 수 있었으며 예측한 값과 동일하는 결과를 얻었습니다.

결론 : 저의 컴퓨터는 4096byte 크기의 페이지를 가지고 있습니다.