Implement Semaphore in NachOS

2016125003 구영민

2023. 05. 24.

Requirement Analysis

본 과제의 요구 사항은 다음과 같다.

1. Semaphore를 NachOS에서 구현한다.
2. Producer 및 Consumer를 Thread를 이용하여 구현한다.
    - 1번에서 작성한 Semaphore의 P() 연산과 V() 연산을 이용한다.
    - 이 때, Buffer Overflow와 Buffer Overrun 현상을 방지한다.

Source Code Analysis

NachOS 3.4 소스 코드를 내려받아 소스 코드의 분석을 진행하였다.

Semaphore

threads/synch.h 및 threads/synch.cc 파일에 Semaphore 클래스가 기 구현되어 있다.

```
void Semaphore::P() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    while (value == 0) {
        queue->Append((void *)currentThread);
        currentThread->Sleep();
    }
    value--;
    (void) interrupt->SetLevel(oldLevel);
}

void Semaphore::V() {
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    thread = (Thread *)queue->Remove();
    if (thread != NULL) scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

P() 연산과 V() 연산은 Atomic 하게 동작해야 한다. 즉 P() 연산과 V() 연산을 수행하던 중, 다른 쓰레드가 선점해서는 안 된다. 이를 달성하기 위해 함수의 시작과 끝에서 인터럽트를 비활성화한다.

P() 연산은 while (true) 함수를 이용하여 value 값이 1 이상이 될 때까지 현재 쓰레드를 Sleep() 상태로 전이한다. V() 연산은 value 값을 증가시키고, Queue에서 대기 중인 쓰레드를 추출하여 ReadyToRun() 상태로 전이한다.

Producer & Consumer

OS 수업 시간에서 학습한 The Coke Machine 소스 코드를 분석한다.

```
semaphore fullSlot = 0;
semaphore emptySlot = 100;
semaphore mutex = 1;

void DeliveryPerson() {
    emptySlot.P(); /* empty slot available? */
    mutex.P();     /* exclusive access */
    /* put 1 Coke in machine */
    mutex.V();
    fullSlot.V();  /* another full slot! */
}

void ThirstyPerson() {
    fullSlot.P();  /* full slot (Coke)? */
    mutex.P();     /* exclusive access */
    /* get 1 Coke from machine */
    mutex.V();
    emptySlot.V(); /* another empty slot! */
}
```

DeliveryPerson() 함수에서는 비어 있는 슬롯이 존재할 때만 실행할 수 있도록 emptySlot.P() 함수를 호출한다. 이후, 슬롯에 대한 락을 획득한 후, 슬롯에 콜라를 넣는다. 이후, 락을 해제하고, 다른 사람이 빼 갈 수 있도록 fullSlot.V() 함수를 호출한다.

ThirstyPerson() 에서는 DeliveryPerson() 함수의 emptySlot.P() 가 fullSlot.P() 으로 변경되고, fullSlot.V() 가 emptySlot.V() 으로 변경된다.

Test Code

threads/threadtest.cc 파일에 테스트 코드를 구현하고 실행할 수 있도록 설계되어 있다.

```
int testnum = 1; // testnum is set in main.cc

void ThreadTest() {
    switch (testnum) {
    case 1:
        ThreadTest1();
        break;
    default:
        printf("No test specified.\n");
        break;
    }
}
```

main.cc 파일에서 args를 파싱해 testnum 변수를 설정하도록 구성되어 있다.

```
extern int testnum;
```

```
int main(int argc, char **argv) {
    testnum = atoi(argv[1]);
}
```

위 테스트를 실행하기 위해서는 다음과 같이 실행할 수 있다.

```
$ threads/nachos -q 1

*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

Plan

1. threads/synch.h 및 threads/synch.cc 파일을 수정하여, MySemaphore 클래스를 선언하고 구현한다.
2. threads/threadtest.cc 파일에 Buffer에 Producing 및 Consuming을 하는 함수를 작성하고, Fork 함수를 이용하여 각각의 쓰레드가 동작하는 것을 확인한다.

Implement

threads/synch.h 파일에 MySemaphore 클래스를 선언한다.

```
class MySemaphore {
  public:
  MySemaphore(int initialValue);
```

```
  void P();
  void V();

  private:
  int value;   // 양수인 경우 세마포어가 사용 가능한 것.
               // 음수인 경우 세마포어의 획득을 원하는 쓰레드의 갯수.
  List *queue; // List<Thread *> 이다.
               // 함수의 구현체의 응답이 void * 여서
               // Thread * 으로 타입 캐스팅 해줘야 한다.
};
```

threads/synch.cc 파일에 P() 함수와 V() 함수의 구현체를 작성한다.

```
void MySemaphore::P() {
    // 인터럽트를 비활성화 시킴으로서
    // P() 함수를 실행하는 동안
    // 컨텍스트 스위칭이 일어나지 않도록 한다.
    IntStatus prevIntLevel = interrupt->SetLevel(IntOff);

    // 세마포어 값을 감소한다.
    this->value--;

    // 만약 현재 세마포어가 사용 불가능한 상태인 경우,
    if (value < 0) {

        // 큐에 현재 실행 중인 쓰레드를 집어넣는다.
        this->queue->Append(currentThread);

        // Sleep() 함수는 인터럽트가 이미 비활성화 되어 있다고 가정한다.
        // 이는 이 함수가 원자성을 위해
        // 인터럽트를 반드시 비활성화해야 하는 동기화 루틴에서 호출되기 때문이다.
        // 만약 인터럽트를 다시 활성화하면,
        // `ASSERT(interrupt->getLevel() == IntOff);` 코드에 의해 어설션에 실패한다.
        currentThread->Sleep();
    } else {
        // 인터럽트를 다시 활성화한다.
        interrupt->SetLevel(prevIntLevel);
    }
}

void MySemaphore::V() {
    // 인터럽트를 비활성화 시킴으로서
    // V() 함수를 실행하는 동안
    // 컨텍스트 스위칭이 일어나지 않도록 한다.
    IntStatus prevIntLevel = interrupt->SetLevel(IntOff);

    // 세마포어 값을 증가한다.
    this->value++;
```

```
    // 만약 현재 세마포어의 획득을 기다리고 있는 쓰레드가 존재하는 경우,
    if (value <= 0) {

        // 큐에서 쓰레드를 가지고 온다.
        Thread *thread = (Thread *) this->queue->Remove();

        // value 값은 세마포어의 실행을 대기하고 있는 쓰레드의 갯수이기 때문에,
        // 무조건 쓰레드를 큐에서 가져올 수 있어야 한다.
        ASSERT(thread != NULL);

        // 방금 꺼낸 쓰레드를 스케줄러에게 ReadyToRun 상태로 변경을 요청한다.
        scheduler->ReadyToRun(thread);
    }

    // 인터럽트를 다시 활성화한다.
    interrupt->SetLevel(prevIntLevel);
}
```

NachOS에서 제공하는 Semaphore 소스 코드와 비교하여 주목할 만한 차이점은 value 값의 가산/감산의 순서와 value 값 확인의 순서를 바꿨다는 점이다. 이를 통하여 로직에서 while (true)를 제거할 수 있었으며, 현재 세마포어를 기다리고 있는 쓰레드의 수를 알기 위해서 큐의 길이를 확인하는 대신, value 값을 이용해 확인할 수 있게 되었다.

threads/threadtest.cc 파일에 Producer 및 Consumer 소스 코드를 작성한다.

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int front = 0;
int rear = 0;

MySemaphore *mutex = new MySemaphore(1);
MySemaphore *empty = new MySemaphore(BUFFER_SIZE);
MySemaphore *full = new MySemaphore(0);

void Producer(int semaphoreArgument) {
    printf("Producer Started!\n");
    MySemaphore *mySemaphore = (MySemaphore *) semaphoreArgument;

    while (true) {
        // 프로듀싱할 데이터를 생성한다.
        int data = 42;

        // 버퍼가 가득 차면 대기한다.
        empty->P();

        // 버퍼에 데이터를 추가한다.
        mutex->P();
        buffer[rear] = data;
```

```
        rear = (rear + 1) % BUFFER_SIZE;
        mutex->V();

        // 생산한 데이터를 출력한다.
        printf("[Producer] Produced data = %d\n", data);

        // 컨슈머가 데이터를 소비할 수 있도록 full 세마포어를 증가시킨다.
        full->V();

        // 현재 스케줄러가 비선점형으로 구현되어 있어,
        // Yield 호출을 명시적으로 해 주어야 컨텍스트 스위치를 진행한다.
        currentThread->Yield();
    }
}

void Consumer(int semaphoreArgument) {
    printf("Consumer Started!\n");
    MySemaphore *mySemaphore = (MySemaphore *) semaphoreArgument;

    while (true) {
        // 버퍼가 비어있으면 대기한다.
        full->P();

        // 버퍼에서 데이터를 소비한다.
        mutex->P();
        int data = buffer[front];
        front = (front + 1) % BUFFER_SIZE;
        mutex->V();

        // 소비한 데이터를 출력한다.
        printf("[Consumer] Consumed data = %d\n", data);

        // 생산자가 데이터를 생산할 수 있도록 empty 세마포어를 증가시킨다.
        empty->V();

        // 현재 스케줄러가 비선점형으로 구현되어 있어,
        // Yield 호출을 명시적으로 해 주어야 컨텍스트 스위치를 진행한다.
        currentThread->Yield();
    }
}
```

위의 소스 코드는 The Coke Machine 소스 코드를 확장하여 구현하였다. 버퍼는 Circular Queue 자료 구조를 활용하여 구현하였다. 이어서, 위 테스트 프로그램을 실행할 수 있도록 구성한다.

```
void ProducerConsumerTest() {
    MySemaphore *mySemaphore = new MySemaphore(1);

    Thread *t1 = new Thread("Producer");
    Thread *t2 = new Thread("Consumer");
```

```
    printf("Before Fork()\n");

    t1->Fork(Producer, (void *) mySemaphore);
    t2->Fork(Consumer, (void *) mySemaphore);
}

void ThreadTest() {
    switch (testnum) {
    case 1:
        ThreadTest1();
        break;
    case 2:
        ProducerConsumerTest();
        break;
    default:
        printf("No test specified.\n");
        break;
    }
}
```

위 테스트 코드에서는 Fork() 함수를 호출하여 Producer 및 Consumer 쓰레드를 생성하고 실행하였다.

Test Result

Producer 쓰레드와 Consumer 쓰레드가 각자 번갈아 실행되며 데이터를 호출하는 것을 확인할 수 있다.

```
$ threads/nachos -q 2

[Producer] Produced data = 42
[Consumer] Consumed data = 42
[Producer] Produced data = 42
[Consumer] Consumed data = 42
```

Source Code

본 보고서에 사용된 소스 코드는 https://github.com/youngminz/OperatingSystem/pull/4/files 에서 확인할 수 있다.