운영체제 토론일지

소속 조 : 빈센조		날짜 : 2021.11.22
구성원		
학번	성명	담당교수 확인
21660021 21960025 21960027	김승현 신가영 양혜교	

토론 주제

다양한 상호배제 알고리즘을 통해 리더/라이터 문제 해결

토론 내용

● 모니터 알고리즘을 통해 리더/라이터 문제 해결을 분석해보자

● 모니터
- 데이터와 프로시저를 모두 포함하는 객체
- 데이터와 프로시저들은 특정 공유 자원을 할당하는데 필요
- 모니터에 있는 데이터들은 오직 모니터 안에서만 접근이 가능
- 모니터는 무기한 연기를 방지하기 위해 대기 중인 스레드에 우선순위를 부여함

● 리더와 라이터 예시
- 리더
  - ▷ 데이터를 읽는 소비자 스레드
  - ▷ DB의 내용을 고치지 않음
  - ▷ 많은 리더가 DB에 접근이 가능
- 라이터
  - ▷ 데이터를 쓰는 생산자 스레드
  - ▷ 데이터 수정 가능
  - ▷ 배타적으로 접근해야 함

● Producer-Consumer Problem(생산자-소비자 문제)
- Bounded Buffer Problem 이라고도 함
- 생산자는 (공유) 데이터를 생산하고 소비자는 (공유) 데이터를 소비하는 구조
- Producer와 Consumer Thread 사이에는 배타적인 락(Exclusive Lock)이 필요
- 두 스레드 사이에 실행 흐름을 컨트롤 해야 함

- Reader-Writer Problem(판독자-기록자 문제)
  - Reader는 (공유)데이터를 읽어들이고 Writer는 (공유)데이터를 쓰는 구조
  - 생산자/소비자 문제와 판독자/기록자 문제에는 차이점이 있다.
    - ▷ Consumer는 해당 데이터를 가져가면 직접 내부처리 하고, 데이터는 공유 영역에 존재하지 않는다. 즉, A Consumer가 소비한 동일한 데이터를 B-Consumer가 동시에 소비해서는 안된다. 소비자들끼리도 배타적인 락이 필요한 상황이다.
    - ▷ Reader는 공유 영역에 데이터를 읽는 것이지 이를 가져가 버리는 것이 아니다. 즉, 다수의 Reader는 공유영역의 데이터를 서로 읽어도 문제가 되지 않는다는 차이점이 있다

- Java 모니터 사용 리더/라이터
  - Condition.java

```
package reader_writer_new;

public class Condition{ //조건변수 클래스

    private int number; //대기중인 작성자/판독자 수를 지정

    public Condition(){ //생성자
        number = 0;
    }

    public synchronized boolean is_non_empty()  { //공유 메소드 is_non_empty
        if(number == 0)  // number가 0이면 false, 아니면 true
            return false;
        else
            return true;
    }
    public synchronized void release_all(){
        number = 0;
        notifyAll(); // WAIT_SET에 있는 모든 Thread를 RUNNABLE 상태로 변경
    }


    public synchronized void release_one(){
        number -=1;
        notify(); // WAIT_SET에 있는 임의의 한 개의 Thread를 다시 Runnable로 변경
    }

    public synchronized void wait_() throws InterruptedException{
        number++;
        wait(); // 락을 가지고 들어온 스레드를 wait()이 호출된 곳에서 락을 해제하고 잠들게한다.
    }

    public synchronized void sleep_() throws InterruptedException{
        Thread.sleep(1000);
    }
}
```

- Synchronized 키워드
  ▷ 자바 객체에 상호 배제 기능을 부여
- set 메소드
  ▷ 생산자 스레드가 wait를 호출
  ▷ 다른 스레드가 객체의 잠금을 얻으려 시도
  ▷ 객체의 set과 get 메소드 호출 가능
- wait 메소드
  ▷ 호출하는 스레드가 객체에 대한 잠금을 해제
  ▷ 소비자 스레드 객체는 생산자 스레드가 통지할 때까지 대기 상태
  ▷ 진행이 가능하다는 통지를 받으면 준비상태로 돌아감

- Monitor.java

```
package reader_writer_new;

public class Monitor{ // 모니터 구현 클래스

    // main 메모리에 저장할 것을 명시
    private volatile int readers;
    private volatile boolean writing;
    private volatile Condition OK_to_Read, OK_to_Write;
    //readers 수, writing여부, 조건변수 생성

    public Monitor(){
        readers = 0;
        writing = false;
        OK_to_Read = new Condition();
        OK_to_Write = new Condition();
    }

    public synchronized void Start_Read(int n){

        System.out.println("wants to read " + n);
        if(writing || OK_to_Write.is_non_empty()){
            try{
                System.out.println("reader is waiting " + n);
                OK_to_Read.sleep_();
                //writing이 false이거나 조건변수가 비어있지 않을때 대기
            }
            catch(InterruptedException e){}
        }
        readers += 1;
        OK_to_Read.release_all();
        //writing이 true이고, 조건변수가 비어있을때 readers값 증가하고
        //thread들을 runnable로 변경
    }
```

```
public synchronized void End_Read(int n){

    System.out.println("finished reading " + n);
    readers -= 1;
    //readers 값 감소
    if(OK_to_Write.is_non_empty()){
        OK_to_Write.release_one();
        //write 조건변수 non empty시 notify
    }
    else if(OK_to_Read.is_non_empty()){
        OK_to_Read.release_one();
        //read 조건변수 non empty시 notify
    }
    else{
        OK_to_Write.release_all();
        //둘다 비어있으면 notifyAll()
    }
}

public synchronized void Start_Write(int n){
    System.out.println("wants to write " + n);
    if(readers != 0 || writing){//readers가 0이 아니거나 writing이 true이면
        try{
            System.out.println("Writer is waiting " + n);
            OK_to_Write.sleep_();//write 조건변수 대기
                }catch(InterruptedException e){}
    }
    writing = true;
}

    public synchronized void End_Write(int n){
        System.out.println("finished writing " + n);
        //writing false로 설정
        writing = false;
        if(OK_to_Read.is_non_empty()){
            OK_to_Read.release_one();
        }else if(OK_to_Write.is_non_empty()){
            OK_to_Write.release_one();
        }else{
            OK_to_Read.release_all();
        }
    }
}
```

- Writer.java

```
package reader_writer_new;

public class Writer extends Thread{
    private Monitor M;
    private int value;
    public Writer(String name, Monitor d){
        super(name);
        M = d;
    }

    public void run(){
        for(int j = 0; j < 5; j++){
                M.Start_Write(j);
                System.out.println("Writer "+getName()+" is writing data...");
                System.out.println("Writer is writing " + j);
                M.End_Write(j);
        }
    }
}
```

- Reader.java

```
package reader_writer_new;

public class Reader extends Thread{
    private Monitor M;
    private String value;
    public Reader(String name,Monitor c){
        super(name);
        M=c;
    }

    public void run(){
        for(int i = 0; i < 5; i++){
                M.Start_Read(i);
                System.out.println("Reader "+getName()+" is retreiving data...");
                System.out.println("Reader is reading " + i);
                M.End_Read(i);
        }
    }
}
```

- Demo.java

```
package reader_writer_new;


public class Demo {
    public static void main(String [] args){
        Monitor M = new Monitor();
        Reader reader = new Reader("1",M);
        Writer writer = new Writer("1",M);
        writer.start();
        reader.start();
    }
}
```

- 결과

```
wants to write 0
Writer 1 is writing data...
wants to read 0
Writer is writing 0
reader is waiting 0
finished writing 0
Reader 1 is retreiving data...
Reader is reading 0
wants to write 1
Writer is waiting 1
Writer 1 is writing data...
finished reading 0
Writer is writing 1
wants to read 1
reader is waiting 1
Reader 1 is retreiving data...
finished writing 1
Reader is reading 1
wants to write 2
Writer is waiting 2
Writer 1 is writing data...
Writer is writing 2
finished reading 1
wants to read 2
reader is waiting 2
finished writing 2
Reader 1 is retreiving data...
wants to write 3
Writer is waiting 3
Reader is reading 2
```

```
Writer 1 is writing data...
finished reading 2
Writer is writing 3
wants to read 3
reader is waiting 3
Reader 1 is retreiving data...
finished writing 3
wants to write 4
Reader is reading 3
Writer is waiting 4
Writer 1 is writing data...
finished reading 3
Writer is writing 4
wants to read 4
reader is waiting 4
finished writing 4
Reader 1 is retreiving data...
Reader is reading 4
finished reading 4
```

발표자료

모니터 사용
리더/라이터 문제 해결

SW 3B 21660021 김승현
SW 3B 21960025 신가영
SW 3B 21660027 양혜교

목차 A table of contents.

Part 1

모니터 & 리더/라이터 특징

- 모니터
  - 데이터와 프로시저를 모두 포함하는 객체
  - 데이터와 프로시저들은 특정 공유 자원을 할당하는데 필요
  - 모니터 안에서만 접근이 가능
- Reader & Writer
  - Reader
    - 데이터를 읽는 소비자 쓰레드
    - DB의 내용을 고치지 않음
    - 많은 리더가 DB에 접근이 가능
  - Writer
    - 데이터를 쓰는 생산자 쓰레드
    - 데이터 수정 가능
    - 배타적으로 접근해야 함

Part 2

동기화 관련 문제

- Producer-Consumer Problem(생산자-소비자 문제)
  - Bounded Buffer Problem 이라고도 함
  - 생산자는 (공유) 데이터를 생산하고 소비자는 (공유) 데이터를 소비하는 구조
  - Producer와 Consumer Thread 사이에는 배타적인 락(Exclusive Lock)이 필요
  - 두 스레드 사이에 실행 흐름을 컨트롤 해야 함
- Reader-Writer Problem(판독자-기록자 문제)
  - Reader는 (공유)데이터를 읽어들이고 Writer는 (공유)데이터를 쓰는 구조
  - 생산자/소비자 문제와 판독자/기록자 문제에는 차이점이 있다.
    - Consumer는 해당 데이터를 가져가면 직접 내부처리 하고, 데이터는 공유 영역에 존재하지 않는다. 즉, A Consumer가 소비한 동일한 데이터를 B-Consumer가 동시에 소비해서는 안된다. 소비자들끼리도 배타적인 락이 필요한 상황이다.
    - Reader는 공유 영역에 데이터를 읽는 것이지 이를 가져가 버리는 것이 아니다. 즉, 다수의 Reader는 공유영역의 데이터를 서로 읽어도 문제가 되지 않는다는 차이점이 있다.

Part 3

Java 모니터 사용 리더/라이터

Condition.java

```
package reader_writer_new;

public class Condition{ //조건변수 클래스

    private int number; //대기중인 작성자/판독자 수를 저장

    public Condition(){ //생성자
        number = 0;
    }

    public synchronized boolean is_non_empty()  { //공유 메소드 is_non_empty
        if(number == 0)  // number가 0이면 false, 아니면 true
            return false;
        else
            return true;
    }
```

```
    public synchronized void release_all(){
        number = 0;
        notifyAll(); // WAIT_SET에 있는 모든 Thread를 RUNNABLE 상태로 변경
    }


    public synchronized void release_one(){
        number -=1;
        notify(); // WAIT_SET에 있는 임의의 한 개의 Thread를 다시 Runnable로 변경
    }

    public synchronized void wait_() throws InterruptedException{
        number++;
        wait(); // 락을 가지고 들어온 스레드를 wait()이 호출된 곳에서 락을 해제하고 잠들게한다.
    }

    public synchronized void sleep_() throws InterruptedException{
        Thread.sleep(1000);
    }
}
```

Part 3

Java 모니터 사용 리더/라이터

Monitor.java

```
package reader_writer_new;

public class Monitor{ // 모니터 구현 클래스

    // main 메모리에 저장될 것을 명시
    private volatile int readers;
    private volatile boolean writing;
    private volatile Condition OK_to_Read, OK_to_Write;
    //readers 수, writing여부, 조건변수 생성

    public Monitor(){
        readers = 0;
        writing = false;
        OK_to_Read = new Condition();
        OK_to_Write = new Condition();
    }

    public synchronized void Start_Read(int n){

        System.out.println("wants to read " + n);
        if(writing || OK_to_Write.is_non_empty()){
            try{
                System.out.println("reader is waiting " + n);
                OK_to_Read.sleep_();
                //writing이 false이거나 조건변수가 비어있지 않을때 대기
            }
            catch(InterruptedException e){}
        }
        readers += 1;
        OK_to_Read.release_all();
        //writing이 true이고, 조건변수가 비어있을때 readers값 증가하고
        //thread들을 runnable로 변경
    }
```

```
public synchronized void End_Read(int n){

    System.out.println("finished reading " + n);
    readers -= 1;
    //readers 값 감소
    if(OK_to_Write.is_non_empty()){
        OK_to_Write.release_one();
        //write 조건변수 non empty시 notify
    }
    else if(OK_to_Read.is_non_empty()){
        OK_to_Read.release_one();
        //read 조건변수 non empty시 notify
    }
    else{
        OK_to_Write.release_all();
        //둘다 비어있으면 notifyAll()
    }
}

public synchronized void Start_Write(int n){
    System.out.println("wants to write " + n);
    if(readers != 0 || writing){//readers가 0이 아니거나 writing이 true이면
        try{
            System.out.println("Writer is waiting " + n);
            OK_to_Write.sleep_();//write 조건변수 대기
                }catch(InterruptedException e){}
    }
    writing = true;
}
```

```
    public synchronized void End_Write(int n){
        System.out.println("finished writing " + n);
        //writing false로 설정
        writing = false;
        if(OK_to_Read.is_non_empty()){
            OK_to_Read.release_one();
        }else if(OK_to_Write.is_non_empty()){
            OK_to_Write.release_one();
        }else{
            OK_to_Read.release_all();
        }
    }
}
```

Part 3

Java 모니터 사용 리더/라이터

Writer.java

```
package reader_writer_new;

public class Writer extends Thread{
    private Monitor M;
    private int value;
    public Writer(String name, Monitor d){
        super(name);
        M = d;
    }

    public void run(){
        for(int j = 0; j < 5; j++){
                M.Start_Write(j);
                System.out.println("Writer "+getName()+" is writing data...");
                System.out.println("Writer is writing " + j);
                M.End_Write(j);
        }
    }
}
```

Reader.java

```
package reader_writer_new;

public class Reader extends Thread{
    private Monitor M;
    private String value;
    public Reader(String name,Monitor c){
        super(name);
        M=c;
    }

    public void run(){
        for(int i = 0; i < 5; i++){
                M.Start_Read(i);
                System.out.println("Reader "+getName()+" is retreiving data...");
                System.out.println("Reader is reading " + i);
                M.End_Read(i);
        }
    }
}
```

Part 3

Java 모니터 사용 리더/라이터

Demo.java

```java
package reader_writer_new;


public class Demo {
    public static void main(String [] args){
        Monitor M = new Monitor();
        Reader reader = new Reader("1",M);
        Writer writer = new Writer("1",M);
        writer.start();
        reader.start();
    }
}
```

결과

```
wants to write 0
Writer 1 is writing data...
wants to read 0
Writer is writing 0
reader is waiting 0
finished writing 0
Reader 1 is retreiving data...
Reader is reading 0
wants to write 1
Writer is waiting 1
Writer 1 is writing data...
finished reading 0
Writer is writing 1
wants to read 1
reader is waiting 1
Reader 1 is retreiving data...
finished writing 1
Reader is reading 1
wants to write 2
Writer is waiting 2
Writer 1 is writing data...
Writer is writing 2
finished reading 1
wants to read 2
reader is waiting 2
finished writing 2
Reader 1 is retreiving data...
wants to write 3
Writer is waiting 3
Reader is reading 2
```

```
Writer 1 is writing data...
finished reading 2
Writer is writing 3
wants to read 3
reader is waiting 3
Reader 1 is retreiving data...
finished writing 3
wants to write 4
Reader is reading 3
Writer is waiting 4
Writer 1 is writing data...
finished reading 3
Writer is writing 4
wants to read 4
reader is waiting 4
finished writing 4
Reader 1 is retreiving data...
Reader is reading 4
finished reading 4
```

다른 조 발표내용

[박부성 & 구본성 조]

- 램포트 알고리즘을 통한 reader writer 구현
  - 우선권을 writer에게 주어 구현
- 구현 방식
  - 모니터에 대한 접근은 한번에 한 스레드만(상호 배제)
  - 모니터에 있는 데이터는 오직 모니터 안에서만 접근
  - (정보 은닉=> 모듈화를 돕고 신뢰성 향상)
  - 모니터에서 실행되는 스레드가 없을 때 스레드가 접근시 모니터에 진입후 LOCK
  - 나머지 스레드는 LOCK이 해제될 때까지 대기
  - 모니터는 무기한 연기를 방지하기 위해 이미 대기중인 스레드에 더 높은 우선순위(에이징 기법)

[이세영 & 황호현 조]

- 세마포어를 활용한 Reader/Writer 구현
- 우선권을 Reader에게 준 경우 구현사랑과 조건
  - 구현 사항
    - ▷ Reader의 개수를 설정해서 개수만큼 Reader를 생성
    - ▷ Reader가 수행되다 임의의 시점에 Writer가 수행되도록 구성
  - 조건
    - ▷ Reader는 여러 개가 공유 데이터를 동시에 읽을 수 있지만 Writer는 공유 데이터를 한 개씩만 쓸 수 있음
    - ▷ Writer가 공유 데이터를 쓸 때는 다른 프로세스는 접근할 수 없도록 구성

[허원석 & 조은새 조]

- 세마포어를 활용한 Reader/Writer 구현
- Reader와 Writer들은 하나의 공유되는 데이터 베이스를 가지고 있다.
- Writer는 데이터 베이스를 수정하는 역할을 하고, Reader는 그 데이터 베이스를 읽어 들이는 역할만 한다.
- Writer가 임계영역에 있을 때는 다른 Writer와 Reader가 접근하는 것을 철저히 막아야 한다.

- Reader의 경우에는 공유 데이터를 읽어 들이기만 하니 다른 Reader가 함께 공유 데이터를 읽는 것을 허락한다. 하지만 Wirter가 접근하지 못하게 막는다.

[김찬혁 & 송지현 조]

- 피터슨의 알고리즘을 활용한 Reader / Writer 문제 해결
- Reader 구조
  - flag1[0] = true 로 설정하여 임계 영역 진입 의사를 표시
  - turn1 = 1 로 주며 reader2에게 먼저 들어가라고 양보
  - 컨텍스트 스위칭이 되지 않았으면 while문 안에 갇힘
  - reader2가 작업을 끝내면 turn1=0, flag[1]=false가 돼서 reader1이 다시 작업을 할 수 있음
  - 1000번 루프 중에 i가 10의 배수일 때 마다 cnt값을 출력
- Writer 구조
  - 앞의 Reader와 똑같은 알고리즘
  - 공유 변수 cnt에 접근하여 값을 변경
  - writer1은 cnt에 1씩 더함
  - writer2는 cnt에 1씩 빼기를 함
  - 1000번 fnv 중에 10번째마다 출력
- Main Method 구조
  - 각 Rader와 Writer 스레드 생성
  - 스레드 실행
  - 최종적인 cnt 값 출력

[박동환 조]

- 스핀락을 통한 Read Write 문제 해결
- 기존 스핀락의 단점
  - 동기화 영역이 큰 경우, 다른 쓰레드로 제어권을 넘기지 않고 계속 Lock 얻는 것 시도하므로 오버헤드가 크다
  - write 가 발생하지 않고 여러 쓰레드에서 read 만 하는 경우에도, reader 쓰레드끼리 Lock Waiting 을 해야 하므로, 불필요한 오버헤드가 발생할 수 있다.

● RWSpinlock으로 기존 스핀락의 단점 해결 ◆ Write Lock 이 걸려있지 않은 상태라면, read Lock 은 중복하여 얻을 수 있다. (즉, 공유 자원에 대해 read 작업은 동시에 여러 개가 가능) ◆ read Lock 이 걸려있는 상태라면, write Lock 은 read Unlock 될 때까지 대기한다. (reader 가 잘못된 값을 read 하는 것을 방지하기 위해)