Journal of The Korea Society of Computer and Information
Vol. 19, No. 12, December 2014
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.12.359

은행계좌 문제를 사용한 프로세스 동기화 교육

양 희 재*

Teaching Process Synchronization with the Bank Account Problem

Hee-Jae Yang*

요 약

프로세스 동기화는 학생들이 운영체제 과목에서 가장 어려워하는 주제 중 하나이다. 한 번에 한 가지만을 생각하는 인간의 특성 상 여러 사건이 동시에 일어나는 병행 프로세스 환경을 이해하기 어렵기 때문이다. 유한버퍼 문제나 식사하는 철학자 문제 등 고전적 동기화 예제는 그 내용이 너무 기술적이거나 비현실적이기 때문에 운영체제를 처음 접하는 저학년 학생들의 관심과 이해를 이끌기 어려웠다. 본 논문에서는 이러한 고전적 동기화 예제의 대안으로 은행계좌 문제의 사용을 제안한다. 은행계좌 문제는 쉽고 현실적이며 일상생활에서 누구나 경험해 본 문제이기 때문에 학생들의 높은 이해와 흥미를 이끌 수 있었다. 프로세스 실행 순서의 제어, 경쟁조건으로 인한 잘못된 결과의 발생, 세마포어 사용, 교착상태, 모니터 등 다양한 동기화 주제에 대한 은행계좌 문제의 적용에 대해 연구하였다.

▸ Keywords : 프로세스 동기화, 병행 프로세스, 다중 쓰레드, 컴퓨터 교육, 운영체제

Abstract

Process synchronization is one of the most difficult subject for students learning the Operating System courses. It is due to the fact that concurrent process environment, where many events occur at the same time, is difficult to understand for ordinary human who thinks only one thing at a time. Classical synchronization examples like the Bounded buffer problem or the Dining philosopher problem fail to hook attention and interest from lower grade students who just begin to study the Operating System courses in college because these examples are either too technical or too unrealistic. In this paper we propose another synchronization example named the Bank account problem as an alternative to the classical ones. Bank account problem is proved to succeed getting high interest and understanding from the student as it is easy and realistic, and almost every student has the experience using bank account in real life. Various

•제1저자 : 양희재 •교신저자 : 양희재
•투고일 : 2014. 8. 26, 심사일 : 2014. 9. 23, 게재확정일 : 2014. 11. 10.
* 경성대학교 컴퓨터공학부 (School of Computer Engineering, Kyungsung University)

synchronization subjects including controlling the execution sequence of each process, incorrect result due to the race conditions, use of semaphores, deadlock, and monitor are considered to apply them to the Bank account problem.

▸ Keywords : process synchronization, concurrent process, multithread, computer education, operating system

I. 서 론

고전적 운영체제에서 핵심 내용 중 하나는 프로세스 관리이며 프로세스 관리의 중요 주제 중 하나는 프로세스 동기화이다. 대부분의 운영체제 교재는 프로세스 동기화 설명에 본문의 상당 부분을 할애하고 있다[1][2][3].

그러나 그 중요성에도 불구하고 학생들에게 프로세스 동기화를 효과적으로 교육하는 것은 매우 어려운 일이다. 한 번에 한 가지만을 생각하는 인간의 특성 상 여러 사건이 동시에 일어나는 병행(concurrent) 프로세스 개념도 이해하기 어려운데 그들 사이의 동기화까지 파악하는 것은 더욱 힘든 것이 어쩌면 당연하다. 임계영역(critical section), 상호배타(mutual exclusion), 바쁜 대기(busy waiting), 세마포어(semaphores), 모니터 (monitors) 등 일련의 난해한 내용은 학생들을 쉽게 낙오하게 만든다.

프로세스 동기화를 잘 이해하게 만드는 한 가지 방법은 효과적 예제를 사용하여 교육하는 것이다. 오래 전부터 읽기-쓰기 문제 (readers-writers problem), 잠자는 이발사 문제 (sleeping barber problem), 궐련 흡연자 문제 (cigarette smokers problem) 등 많은 동기화 예제들이 제안되어 발표된 바 있으며, 그 중 가장 대표적인 것으로 유한 버퍼 문제 (bounded buffer problem) 또는 식사하는 철학자 문제 (dining philosopher problem) 등을 들 수 있다. 유한 버퍼 문제는 생산자-소비자 문제 (producer-consumer problem)로 불리기도 한다.

그러나 교육 현장에서 이 예제들을 사용하여 동기화 문제를 강의해 본 결과 어려움이 많았다. 즉 유한 버퍼 문제는 그 내용이 지나치게 기술적일 뿐 아니라 현실에서 경험해 보지 못한 내용이라 저학년 학생들의 흥미를 불러일으키기 힘들다. 여러 개의 세마포어를 사용하는 것도 동기화를 처음 접하는 학생들에게는 부담으로 작용하여 이 문제에 대한 접근을 기피하게 만든다.

식사하는 철학자 문제에 대해서는 대부분의 학생들이 흥미를 가진다. 식사와 생각을 반복적으로 수행하는 철학자들의 장난스런 모습을 머리 속으로 떠 올리면서 그 상황을 재미있어 하는 것을 볼 수 있다. 그러나 이 문제는 비현실적이다. 단지 동기화 설명 목적으로만 사용될 뿐 실제로 현실 프로그램에 적용되지 못해 일시적 흥미만 일으킬 뿐이다.

따라서 유한 버퍼 문제처럼 너무 기술적이고 어렵지 않으며 식사하는 철학자 문제처럼 너무 비현실적이지 않은 새로운 동기화 예제의 제시가 요청된다. 그렇기 위해서는 문제 자체가 학생들이 현실 세계에서 많이 경험해보았고 또 쉬운 내용이어야 한다. 효과적 프로세스 동기화 교육을 위해 이런 조건을 만족하는 적절한 예제의 개발이 절실하다.

수년간의 강의 경험을 통해 저자는 은행계좌 문제(Bank Account Problem)가 프로세스 동기화 교육에 매우 효과적임을 발견하였다. 이 문제는 한 개의 은행계좌에 대해 여러 프로세스가 입금 또는 출금을 하는 상황을 나타낸 것이다. 이것을 사용하여 프로세스 동기화 교육을 시행해 본 결과 학생들의 이해도와 집중도가 월등히 향상되는 것을 볼 수 있었다. 일상생활에서 누구나 경험해 본 것이며 간단하기 때문이다. 또한 확장성도 뛰어나서 교착상태 등 다양한 동기화 문제에도 적용할 수 있다. 후속되는 전통적 동기화 문제의 이해에도 큰 도움을 준다.

본 논문에서는 은행계좌 문제를 프로세스 동기화 교육에 적용하고 또한 다양한 형태로 확장하여 학생들의 이해를 높인 실제 경험에 대해 소개하고자 한다. 이하 본 논문의 내용은 아래 순서로 기술되어있으며, 학생들에게도 같은 순서로 프로세스 동기화에 대해 교육했다.

- 은행계좌에 대한 입금 및 출금 개요
- 다중 프로세스 적용 시 발생하는 문제점

- 문제 해결을 위한 세마포어 등 동기화 도구의 사용
- 입출금 순서의 임의 제어
- 은행계좌 문제의 다양화와 교착상태
- 유한버퍼 문제와의 연관성

본 논문의 2장에서는 프로세스 동기화 교육과 관련된 그간의 연구들에 대해 알아보며, 3장에서는 은행계좌 문제의 정의와 프로세스 동기화의 필요성을 설명한다. 4장에서는 세마포어를 사용하여 은행계좌 문제를 해결하는 방법에 대해 알아보며, 5장에서는 은행계좌 문제의 다양화, 교착상태, 모니터 사용 등 확대 방안을 다룬다. 6장에서는 은행계좌 문제와 전통적 유한버퍼 문제와의 연관성에 대해 설명하며, 7장에서는 은행계좌 문제 사용 후의 학습성과 향상도를 분석했다. 마지막 8장에서 본 연구의 중요성 및 향후 연구 방향에 대해 알아본다.

II. 관련 연구

프로세스 동기화는 이미 수십 년 전부터 연구되었던 내용이며 더 이상 새로운 연구 주제는 되지 못한다. 그러나 효과적인 동기화 교육을 위한 시도는 지금까지도 많은 학자들에 의해 꾸준히 연구되어지고 있다. Fekete는 동기화를 포함한 다중쓰레드 교육의 중요성이 날로 높아지고 있음을 상기시키며 새로운 교과과정 편성에 최우선적으로 포함되어야 할 것을 강조하고 있다[4]. Shene 역시 운영체제 과목에서의 효과적인 다중 쓰레드 교육의 중요성을 개인적인 교육경험을 통해 주장했다[5].

효과적인 프로세스 동기화 교육을 위한 구체적 방법들도 많이 연구되었다. 한 번에 한 가지 일만을 하는 것에 익숙한 사람의 특성 상 동시에 여러 가지 사건이 일어나는 병행 프로그램을 이해하는 것은 무척 힘들며, 따라서 병행 프로그램에서 일어나는 일들을 시각적으로 보여주는 연구가 많았다. Convit[6], JAVAVIS[7], JaVis[8], JACOT[9] 등 병행 프로그램의 시각화에 대한 많은 일련의 논문들이 발표되고 있다.

```
class BankAccount {
 int balance;                              // 잔액
 void deposit(int amount) {                // 입금
            balance = balance + amount;// 잔액 증가
 }
 void withdraw(int amount) {               // 출금
            balance = balance - amount;// 잔액 감소
 }
 int getBalance() {                        // 잔액 조회
            return balance;
 }
}
```

그림 1. 자바 언어로 표현한 은행계좌 클래스
Fig. 1. BankAccount class in Java language

시각화 도구 외에도 Carr는 ThreadMentor 라는 다중쓰레드 프로그래밍을 위한 교육적 도구를 개발하였으며[10], Malnati는 프로세스의 실행 추적을 통한 병행 프로그램 교육법에 대한 논문을 발표하기도 했다[11].

동기화 교육을 위한 예제 문제의 개발에 대한 연구도 계속되고 있다. Robbins은 기존의 유한 버퍼 문제나 읽기-쓰기 문제가 학부생들의 프로세스 동기화 교육에 그다지 효과적이지 못함을 지적하며 동기화의 중요성을 알게 하기 위해 다른 접근법이 필요함을 강조한 바 있다[12].

1장에서 소개한 유한 버퍼 문제, 식사하는 철학자 문제, 읽기-쓰기 문제, 잠자는 이발사 문제, 궐련 흡연자 문제 등 전통적인 예제 문제 외에도 최근 Shene 은 배고픈 독수리 문제 (Hungry eagle problem), 강 건너기 문제 (River crossing problem) 등 학생들에게 호감을 줄 수 있는 기존 문제를 새로운 동기화 문제로 제시했다[10].

이와 같이 오랜 기간에 걸쳐 많은 동기화 문제가 제안되는 이유는 효과적 예제의 사용이 학생들의 동기화 교육에 무엇보다도 큰 도움이 될 수 있기 때문이다. 저자는 자바 프로그래밍 언어 교육 목적으로 은행계좌 문제[13]를 처음 접했으나 곧 이 문제가 프로세스 동기화 문제 예제로 매우 적합함을 깨닫고 4년 전부터 운영체제 강의에 적용해오고 있다. 은행계좌 문제를 프로세스 동기화 교육에 적용한 사례는 이미 존재하지만[14] 어셈블리언어 수준의 의사코드(psuedo code)를 사용했으며 동기화 문제 전반이 아닌 단지 경쟁조건(racing condition) 문제점만 언급했다는 점에서 본 논문과 차이가 있다.

III. 은행계좌 문제

1. 은행계좌 문제

은행계좌 문제는 현실세계의 은행계좌에 대한 세 가지 기본 동작, 즉 입금, 출금, 잔액 조회 등을 프로그램으로 나타낸 것이다. 학생들은 누구나 은행계좌를 사용해 본 경험이 있으므로 이 문제를 다른 어느 예제보다 빠르고 쉽게 이해한다.

은행계좌에 대한 세 가지 동작을 자바 언어로 표현하면 그림 1과 같다. 이 프로그램에서 입금과 출금 동작은 잔액(balance)이라는 공통변수(common variable)의 변경을

일으키는데, 공통변수의 변경은 프로세스 동기화 또는 임계구역 문제에서 핵심 내용이기도 하다.

2. 다중 프로세스

이제 은행계좌를 사용하는 다중 프로세스 또는 다중 쓰레드 환경에 대해 고찰해보자. 프로세스와 쓰레드를 구분하여 생각할 수도 있지만 이 논문에서는 자바 언어를 사용하기 때문에 이하 다중 쓰레드를 사용하여 문제를 설명하기로 한다[15].

우선은 두 개의 쓰레드를 사용하는 환경을 고찰하자. 5장에서 쓰레드의 개수를 확대해 나갈 것이다. 이 환경에서 우리는 부모(Parent)와 자녀(Child) 쓰레드를 각각 하나 씩 둔다. 부모 쓰레드는 은행계좌에 계속 입금을 하며 자녀 쓰레드는 같은 은행계좌에서 계속 출금을 하는 것으로 가정한다. 현실 세계에서 부모님이 자녀들에게 생활비 또는 학비를 수시로 보내는 상황을 나타낸 것이다.

부모 쓰레드의 입금액과 자녀 쓰레드의 출금액을 같게 하면 최종 잔액은 0원이 될 것이다. 그림 2에 부모와 자녀 쓰레드를 자바 언어로 각각 나타내었으며, 그림 3에 테스트 프로그램을 보였다.

```
class Parent extends Thread { // 부모 쓰레드
 BankAccount b;
 int count;
 Parent(BankAccount b, int count) {
            this.b = b;
            this.count = count;
 }
 public void run() {
            for (int i=0; i<count; i++)
                        // 1원씩 count 번 입금
                        b.deposit(1);
 }
}

class Child extends Thread { // 자녀 쓰레드
 BankAccount b;
 int count;
 Child(BankAccount b, int count) {
            this.b = b;
            this.count = count;
 }
 public void run() {
            for (int i=0; i<count; i++)
                        // 1원씩 count 번 출금
                        b.withdraw(1);
 }
}
```

그림 2. 부모와 자녀 쓰레드
Fig. 2. Parent and Child threads

3. 문제의 제기

프로세스 동기화에서 중요한 내용 두 가지는 1) 필요에 따라 프로세스의 실행 순서를 임의로 통제할 수 있게 하는 것과, 2) 프로세스간의 경쟁조건(race condition)으로 인해 잘못된 결과가 나오지 않게 하는 것을 들 수 있다[1][2][3]. 은행계좌 문제에서도 마찬가지다. 이 두 가지 내용이 은행계좌 문제에서 어떻게 적용되는지를 보임으로서 학생들에게 동기화 문제 해결의 필요성을 제기한다.

3.1 프로세스 실행 순서의 통제

입출금이 어떤 순서로 일어나는지를 보이기 위해 입금 시에는 + 기호가, 출금 시에는 - 기호가 화면에 나타나도록 그림 4와 같이 BankAccount 클래스 코드를 일부 변경한다. 이렇게 변경한 후 그림 3의 프로그램을 실행하면 화면에 + 와 - 가 섞여 출력되므로 어느 쓰레드가 언제 실행되었는지 눈으로 볼 수 있다. 매 실행마다 각기 다른 출력 화면이 나오는 것을 보임으로서 프로세스의 실행 순서가 통제되지 못했음을 확인시킬 수 있다. 이 시점에서 학생들에게 다음과 같은 질문을 던져 본다.

- 항상 부모 쓰레드가 자식 쓰레드보다 먼저 실행되게 하려면 어떻게 해야 하는가?
- 부모 쓰레드와 자식 쓰레드가 교대로 실행되게 하려면 어떻게 해야 하는가?
- 어떤 순간에도 잔액이 음수가 되지 않게 하려면 어떻게 해야 하는가?

이 질문을 통해 학생들은 프로세스 동기화의 주요 주제 중

```
class Test {
 static final int MAX = 100; // 입출금 회수
 public static void main(String[] args) {
            // 은행계좌를 만들고
            BankAccount b = new BankAccount();
            // 부모 쓰레드와
            Parent p = new Parent(b, MAX);
            // 자식 쓰레드를 만든 후
            Child c = new Child(b, MAX);
            // 각각 실행시킨다.
            p.start();
            c.start();
 }
}
```

그림 3. 테스트 프로그램 (main 프로그램)
Fig. 3. Test program (main program)

```
void deposit(int amount) {
        // 입금 시 "+" 출력
        System.out.print("+");
        balance = balance + amount;
}
void withdraw(int amount) {
        // 출금 시 "-" 출력
   System.out.print("-");
        balance = balance - amount;
}
```

그림 4. 입출금 순서가 보이도록 코드 변경
Fig. 4. Modifying code to show deposit/withdraw actions

하나가 다중 프로세스 환경에서 우리가 원하는 대로 프로세스 실행을 제어하고 통제할 수 있게 하는 것임을 이해하게 된다.

3.2 잘못된 결과의 발생

입금과 출금 동작은 balance 라는 공통변수를 업데이트한다 (그림 1). 두 개 이상의 프로세스가 동시에 공통변수를 변경하려 하면 잘못된 결과가 발생될 수 있음을 보인다.

그림 4의 코드만으로는 잘못된 결과의 발생 빈도가 극히 낮으므로 데모 효과를 위해 그림 5와 같이 입출금 과정에 지연 시간을 추가하도록 BankAccount 코드를 변경한다. 이 코드에서는 입출금 후 balance 변수를 즉시 업데이트하지 않고 temp 라는 임시 변수에 잠시 저장하고 화면에 + 또는 - 기호를 출력한 후 (이때 시간 지연이 발생된다) 비로소 업데이트한다. 단일 프로세스 환경이라면 아무 문제가 없음을 먼저 이해하게해야 한다.

또한 그림 6과 같이 입출금이 끝난 후 최종 잔액을 화면에 출력하도록 테스트 프로그램도 변경한다. 이 프로그램을 실행하면 최종 잔액에서 잘못된 결과가 출력되는 것을 발견하게 된다. 그림 5의 지연 시간 추가로 인해 입금 또는 출금 동작이 채 끝나지 않은 상태에서 콘텍스트 스위칭이 일어날 수 있기 때문이다. 학생들에게 다음과 같은 질문을 던져 동기화의 중요성을 깨닫게 한다.

- 여러 번 반복 실행하여 최종 잔액이 얼마가 나오는지 알아보라.
- 최종 잔액이 0원이 아니라 100원, -85원 등 틀린 값이 나온 이유는 무엇인가?
- 실제 은행 전산 시스템에서 이런 일이 발생되면 그 심각성은 얼마나 클까?

```
void deposit(int amount) {
            // 변경된 값을 임시변수에 저장하고
            int temp = balance + amount;
            System.out.print("+");// 시간 지연 후
            balance = temp;          // 잔액 업데이트
}
void withdraw(int amount) {
            // 변경된 값을 임시변수에 저장하고
            int temp = balance - amount;
            System.out.print("-"); // 시간 지연 후
            balance = temp;          // 잔액 업데이트
}
```

그림 5. 입출금 과정에 지연 시간 추가
Fig. 5. Adding delay in deposit/withdraw actions

이 질문을 통해 프로세스 동기화는 몰라도 괜찮은 내용이 아니라 다중 프로세스 환경에서 반드시 해결해야 할 중요 문제임을 인식하게 하고 공통변수의 동시 업데이트가 얼마나 큰 위험을 야기할 수 있는지 이해하게 한다.

이상의 문제 제기를 통해 대부분의 학생들은 프로세스 동기화가 무엇이며 또 얼마나 심각한 문제인지를 깨닫게 된다. 은행계좌는 누구에게나 쉽게 피부에 와 닿는 문제일뿐더러 잔액은 정확해야 한다는 사실을 알기 때문이다. 반면 유한 버퍼 문제나 식사하는 철학자 등 고전적 문제로는 학생들의 주의를 끌기도 어려웠고 동기화의 중요성을 이해시키기도 어려웠다.

IV. 세마포어를 사용한 동기화

기본적으로 자바는 모니터[16]를 사용해 프로세스 동기화를 하지만 운영체제 강의에서 보다 범용적인 동기화 도구는 세마포어이다[17]. 따라서 모니터 사용은 나중으로 미루고 처음에는 세마포어를 사용한 프로세스 동기화를 교육한다. 자

```
class Test {
static final int MAX = 100;
public static void main(String[] args) throws
                InterruptedException {
                BankAccount b = new BankAccount();
                Parent p = new Parent(b, MAX);
                Child c = new Child(b, MAX);
                p.start();// 부모 쓰레드와
                c.start();// 자식 쓰레드 각각 실행 후
                p.join();// 부모와 자식 쓰레드가
                c.join();// 각각 종료하기를 기다린다.
                System.out.println("Final balance = "
                  + b.getBalance());// 최종 잔액 출력
 }
}
```

그림 6. 최종 잔액을 화면에 출력하도록 수정
Fig. 6. Modifying code to show the final balance

```
import java.util.concurrent.Semaphore;

class BankAccount {
 int balance;
 Semaphore sem;
 BankAccount() {
            sem = new Semaphore(1);// 초기값 = 1
 }
 void deposit(int amount) {          // 입금
            try {
            sem.acquire();          // 진입 전: acquire()
            } catch (InterruptedException e) {}
            int temp = balance + amount;
            System.out.print("+");
            balance = temp;
            sem.release();          // 나온 후: release()
 }
 void withdraw(int amount) {         // 출금
            try {
            sem.acquire();          // 진입 전: acquire()
            } catch (InterruptedException e) {}
            int temp = balance - amount;
            System.out.print("-");
            balance = temp;
            sem.release();          // 나온 후: release()
 }
 int getBalance() {
            return balance;
 }
}
```

그림 7. 계산 오류를 방지한 BankAccount 클래스
Fig. 7. Preventing incorrect result in BankAccount class

바 모니터로 세마포어를 구현하는 방법은 여러 문헌에 소개되어 있으며[18], Java 5부터는 아예 java.util.concurrent.Semaphore 라는 클래스가 자바 API 라이브러리에 포함되어있다.

3.3절에서 제기했던 두 가지 문제, 즉 프로세스 실행 순서의 통제와 잘못된 결과의 발생을 세마포어를 사용하여 해결할 수 있음을 학생들에게 보인다. 먼저 더 심각한 문제인 잘못된 결과의 발생, 즉 계산 오류를 방지하는 방법에 대해 교육한다.

1. 계산 오류의 방지

은행계좌 문제에서 잘못된 계산 결과가 나온 것은 공통변수인 balance를 여러 쓰레드가 동시에 업데이트했기 때문임을 주지시킨 후 세마포어를 사용하여 해결할 수 있음을 보인다.

즉 공통변수를 업데이트하는 임계영역을 정의한 후 임계영역 진입 전에 세마포어의 P 동작, 즉 acquire() 메소드를 호출하고, 임계영역을 나온 후 V 동작, 즉 release() 메소드를 호출함으로서 오직 하나의 쓰레드만 임계영역에 들어가게 하여 (상호배타) 계산 오류를 방지할 수 있음을 이해시킨다 (그림 7). 최대 한 개의 쓰레드만 임계영역에 진입할 수 있게 하기 위해 세마포어의 초기 값은 1로 둔다.

그림 7의 BankAccount 클래스를 사용하여 실행 결과 항상 올바른 계산, 즉 최종 잔액이 언제나 0원이 나오는 것을 확인시킨다. 이 내용을 교육함으로서 학생들은 임계구역, 상호배타, 세마포어 등의 개념을 이해하게 된다.

2. 프로세스 실행 순서

임계구역에 대한 배타적 접근을 통해 잘못된 결과를 해결할 수 있음을 보인 후 이번에는 프로세스 실행 순서를 우리가 원하는 대로 통제할 수 있음을 교육한다. 3장에서 제기했던 문제들을 풀어본다.

가. 항상 부모 쓰레드가 먼저 실행되게 하려면 어떻게 해야 하는가?

그림 7의 코드에 상호배타를 위한 **sem** 세마포어 외에 실행순서 조정을 위한 **sem2** 세마포어를 추가한다. 프로그램이 시작되면 부모 쓰레드는 그대로 실행되게 하고, 자식 쓰레드는 초기 값이 0인 **sem2** 세마포어에 대해 **acquire()** 를 호출하게 하도록 **deposit()**, **withdraw()** 메소드를 각각 수정한다. 즉 자식 쓰레드가 먼저 실행되면 세마포어 **sem2**에 의해 블록되고, 블록된 자식 쓰레드는 나중에 부모 쓰레드가 깨워주게 한다.

나. 부모 쓰레드와 자식 쓰레드가 교대로 실행되게 하려면 어떻게 해야 하는가?

블록된 부모 쓰레드는 자식 쓰레드가 깨워주고, 블록된 자식 쓰레드는 부모 쓰레드가 각각 깨워주도록 한다. 상호배타를 위한 **sem** 세마포어 외에 부모 쓰레드의 블록을 위해 **dsem** 세마포어를, 자식 쓰레드의 블록을 위해 **wsem** 세마포어를 각각 사용한다.

다. 어떤 순간에도 잔액이 음수가 되지 않도록 하려면 어떻게 해야 하는가?

출금하려는 액수보다 잔액이 작으면 자식 쓰레드가 블록되도록 하며 이후 부모 쓰레드가 깨워주게 한다. 상호배타를 위한 **sem** 세마포어 외에 **sem2** 세마포어를 사용하여 잔액 부족 시 자식 쓰레드가 블록 되도록 한다.

V. 은행계좌 문제의 다양화

앞서 살펴본 3장과 4장의 내용을 실험을 통해 확인하게 함으로써 학생들은 아래 내용을 이해하게 된다.

- 동기화 문제의 정의와 필요성
- 동기화가 이루어지지 않았을 때의 심각성 이해 (은행과 돈 등 실생활 예제)
- 프로세스 스케쥴링 방식의 다양성에도 불구하고 우리가 원하는 순서대로 프로세스 실행 순서를 통제할 수 있음
- 세마포어 등 동기화 도구의 사용법

이상의 내용 이해가 이루어지면 은행계좌 문제를 확대하여 동기화 문제를 더욱 다양하게 다루어 본다. 먼저 프로세스 실행 순서를 보다 공교하게 통제하는 문제를 제시해본다. 주요 내용은 다음과 같다.

- 부모의 입금 동작이 모두 끝난 후 (즉 총 금액 모두를 입금 후) 자녀의 출금 동작이 일어나게 하라.
- 부모의 입금 동작이 절반 끝난 후 (즉 총 금액의 절반까지 입금 후) 자녀의 출금 동작이 일어나게 하라.
- 프로그램 실행 내내 잔액이 항상 음수 또는 0 이 되게 하라.

```
class BankAccount {
 int balance;
 synchronized void deposit(int amount) {
              int temp = balance + amount;
              System.out.print("+");
              balance = temp;
              notify();// 자식 쓰레드를 깨워준다.
 }
 synchronized void withdraw(int amount) {
              // 잔액이 부족하면 블록된다.
              while (balance < amount)
               try {
                            wait();
               }
               catch (InterruptedException e) {}
              int temp = balance - amount;
              System.out.print("-");
              balance = temp;
 }
 int getBalance() {
              return balance;
 }
}
```

그림 8. 모니터를 사용한 BankAccount 클래스
Fig. 8. Using Monitor in BankAccount class

- 프로그램 실행 내내 잔액이 항상 K원을 넘지 않도록 하라.
- 부모 쓰레드가 N개 있고 자녀 쓰레드가 1 개 있는 환경에서 위 문제를 반복하라.
- 부모 쓰레드가 한 개 있고 자녀 쓰레드가 N개 있는 환경에서 위 문제를 반복하라.
- 부모 쓰레드와 자녀 쓰레드가 각각 N개씩 있는 환경에서 위 문제를 반복하라.
- 부모 쓰레드가 N개 있고 자녀 쓰레드가 M개 있는 환경에서 위 문제를 반복하라.

교착상태 문제에 대해서도 적용할 수 있다. 예를 들어 4장에서 소개한 두 번째 예제인 부모, 자식 쓰레드의 교대 실행 프로그램의 경우 **deposit()** 메소드에서 조건동기를 위한 세마포어를 먼저 사용 후 배타동기를 위한 세마포어를 사용해야 하는데 **(dsem.acquire(); sem.acquire();)**, 순서를 반대로 하여 **sem.acquire(); dsem.acquire();** 와 같이 하면 교착상태가 발생된다. **withdraw()** 메소드에서도 **sem** 과 **wsem** 의 순서를 바꾸면 역시 교착상태가 발생한다. 교착상태 발생의 네 가지 필요조건[1]을 언급하면서 이 예제를 소개하면 학생들이 쉽게 이해함을 볼 수 있었다.

동기화 도구로 세마포어 외에 모니터를 소개하는 것도 중요하다. 4장에 언급한 네 가지 코드를 각각 모니터를 사용하는 코드로 변환해보고 다양화된 동기화 문제, 교착상태 등에도 각각 적용시켜 볼 수 있음을 교육한다. 그림 8은 4장의 세 번째 예제인 잔액이 음수가 되지 않는 **BankAccount** 클래스를 모니터를 사용하여 작성한 것이다.

VI. 유한버퍼 문제와의 연관성

서론에서 언급했던 것처럼 대표적 운영체제 교과서들은 동기화 예제로서 1) 유한버퍼 문제 (bounded buffer problem), 2) 읽기/쓰기 문제 (readers and writers problem), 3) 식사하는 철학자 문제 (dining philosophers problem)를 이 순서대로 소개하고 있다.

그러나 학생들은 처음 문제인 유한버퍼 학습부터 쉽게 좌절하는 경향이 있었다. 문제가 어렵기 때문이다. 세마포어의 개념을 아직도 잘 이해하지 못하는 학생들에게 이 문제는 mutex, full, empty 등 세 개의 세마포어 사용을 요구하고 있으며, 유한버퍼를 필요로 하며, 환형 대기열 (circular queue) 도 사용하고 있다 [1]. 학생들 입장에서 동기화 개념은 고사하고 문제 자체가 복잡한 것이다.

본 논문에서 설명한 은행계좌 문제를 가장 먼저 제시하고

이후 유한버퍼 문제 등을 다룬 뒤부터 동기화에 대한 학생들의 이해도 및 강의 만족도가 크게 향상되었다. 은행계좌 문제는 그림 7처럼 상호배타를 위한 세마포어 하나만 사용하는 문제부터 시작하므로 매우 단순하다.

실제로 은행계좌 문제는 유한버퍼 문제와 매우 큰 연관성을 가진다. 부모 쓰레드는 은행계좌에 입금을 하며 자녀 쓰레드는 출금을 하는데, 이것은 유한버퍼 문제의 생산자, 소비자 쓰레드에 각각 정확히 대응된다. 그림 7의 코드에서는 입출금액수에 아무 제한을 두지 않으므로 무한버퍼 문제(unbounded buffer problem) 에 해당된다.

4장의 세 번째 문제, 즉 어떤 순간에도 잔액이 음수가 되지 않도록 하는 것은 잔액 부족 시 자식 쓰레드가 블록되도록 하는 것이다. 이것은 유한버퍼 문제에서 버퍼가 비면 소비자 프로세스가 블록되는 것과 동일하다.

또한 5장 은행계좌 문제의 다양화에서 네 번째 문제, 즉 프로그램 실행 내내 잔액이 항상 K원을 넘지 않도록 하는 것은 잔액이 K원을 초과할 시 부모 쓰레드가 블록되게 하는 것이다. 이것은 유한버퍼 문제에서 버퍼가 가득차면 생산자 프로세스가 블록되는 것과 동일하다.

유한버퍼 문제는 버퍼가 비면 소비자 프로세스가 블록되고, 버퍼가 가득차면 생산자 프로세스가 블록되도록 해야 하는데, 이것은 은행계좌 문제에서 프로그램 실행 내내 잔액이 0원부터 K원 사이에 있도록 하는 것과 동일하다. 다만 버퍼를 사용하지 않기 때문에 환형 버퍼 구현에 필요한 배열, 입출력 위치 인덱스 등을 사용해야 하는 유한버퍼 문제에 비해 학생들이 훨씬 쉽게 이해할 수 있다.

전통적 교재 내용처럼 처음부터 유한버퍼 문제를 교육하기보다는 은행계좌 문제를 먼저 다룬 후 전통적 문제로 넘어가면 학생들의 학습능률은 훨씬 더 높아지는 것을 수년간의 강의 경험을 통해 확인할 수 있었다.

Ⅶ. 학습성과 평가

은행계좌 문제를 사용한 프로세스 동기화 교육의 효과는 학생들의 교과목 학습성과 평가에서 잘 나타나고 있다. 저자의 소속 학과에서는 한국공학교육인증원[19]의 공학교육인증제를 시행 중인데 이 제도에 따라 매학기말마다 과목별 학습성과를 설문형식으로 조사하고 있다.

표 1에 운영체제 과목의 학습성과 평가 항목 내용을 나타내었다. 6개의 점수형 항목과 2개의 주관식 항목으로 구성되어있는데, 프로세스 동기화는 점수형 항목 세 번째에 포함되어있다.

표 1. 운영체제 과목의 학습성과 평가 항목
Table 1. Items of evaluating course outcome in Operating System

No	학습성과 평가 항목
1	운영체제의 역할과 주요 역사를 설명할 수 있다.
2	인터럽트, 시스템 호출 등 주요 개념을 설명할 수 있다.
3	CPU 스케줄링, 프로세스 동기화 등을 설명할 수 있다.
4	페이징, 세그멘테이션 등 메모리관리를 설명할 수 있다.
5	가상 메모리와 요구 페이징 등의 원리를 설명할 수 있다.
6	파일의 개념, 디스크 스케줄링 등을 설명할 수 있다.
7	이 과목에서 가장 도움이 되었던 주제는 무엇인가?
8	위 주제 외에 이 과목에서 배우기 원하는 주제는 무엇인가?

프로세스 동기화를 포함한 프로세스 관리 부분 학습성과를 표 2에 나타내었다. 이 결과는 학생들 스스로 판단한 성과를 매우 그렇다(5), 그렇다(4), 보통이다(3), 아니다(2), 매우 아니다(1) 로 대답한 평균값인데, 은행계좌 문제를 적용한 2012학년도부터 학습성과가 매우 향상되었음을 알 수 있다. 적용하기 전 2년 평균치가 3.9 인데, 적용 이후 2년 평균치는 4.3이 되어 괄목할만한 효과가 있었음을 발견할 수 있다.

표 2. 프로세스 관리 부분의 학습성과 평가 결과
Table 2. Result of course outcome in process management

연도	2010	2011	2012	2013
점수	3.8	3.9	4.3	4.2

은행계좌 문제 적용 전에는 표1의 7번 문제의 답으로 프로세스 동기화를 든 학생이 매우 드물었는데, 적용 후에는 많은 학생들이 프로세스 동기화를 가장 도움이 된 주제라고 평가하였다. 평가한 학생들의 답변을 일부 발췌 정리하여 표 3에 나타내었다. 좋은 동기화 문제 개발이 학생들의 관심과 이해를 대폭 끌어내었음을 알 수 있다.

본 학습성과 평가는 2010년 2학기부터 2013년 2학기까지 4개년 내용을 정리한 것이며, 연도별 수강인원 및 평가 참

표 3. 과목에서 가장 도움이 된 주제
Table 3. Most helpful subjects in the class

- 프로세스 동기화에 대해 알게 되어 좋았습니다.
- 가장 도움 되었던 것은 세마포어이다.
- 인상적이면서 가장 잘 기억이 남는 부분은 CPU스케줄링과 프로세스 동기화 부분이었던 것 같습니다.
- Process Management 주제가 가장 도움이 많이 되었다.
- Process 동기화에 대한 이야기 입니다.
- 프로세스 동기화 문제
- CPU스케줄링, 프로세스 동기화 등을 이해하고 운영체제에 대해서 알게 되었던 것 같습니다.

표 4. 연도별 수강인원 및 평가인원
Table 4. Yearly class sizes and evaluation participants

연도	2010	2011	2012	2013
수강인원	51	44	62	23
평가인원	42	42	53	18
비율(%)	82	95	85	78

여인원은 표 4와 같다.

Ⅷ. 결 론

학생들에게 운영체제 과목을 강의해 오면서 교육하기 가장 까다롭고 학생들도 어려워하는 주제는 단연 프로세스 동기화에 대한 내용이었다.

전형적인 운영체제 교재에서는 유한 버퍼 문제, 읽기-쓰기 문제, 식사하는 철학자 문제 등을 사용하여 동기화를 설명하고 있는데, 이 내용만으로는 학생들의 흥미를 이끌어내기가 무척 어려웠다. 식사하는 철학자 문제가 그 중 가장 쉽고 재미난 주제였지만 현실 세계로의 적용 분야가 명쾌하지 않아 학생들은 동기화를 그렇게 중요한 것으로 보지도 않고 그저 어려운 주제인 것으로만 받아 들였다. 학기말 강의평가 설문조사에서도 대다수 학생이 가장 어려운 분야가 프로세스 동기화라고 답변했다.

그러나 은행계좌 문제를 전통적 동기화 문제에 앞서 강의한 후부터는 학생들의 반응이 크게 달라졌다. 오히려 운영체제 과목에서 가장 흥미로운 주제가 프로세스 동기화라고 응답하는 학생도 많이 생겼다. 이 주제에 대한 학습성과 점수도 기존 3.9에서 4.3으로 크게 향상되었다.

효과적 예제의 사용이 학생 교육에 실로 중요함을 실감할 수 있었다. 은행계좌 문제는 학생들이 현실세계에서 경험을 통해 익히 알고 있는 내용일 뿐 아니라 간단하며 또한 돈이 오고 가는 상황으로 인해 동기화 문제의 중요성도 피부로 느끼게 하고 있다.

은행계좌 문제가 프로세스 동기화 교육을 위한 가장 효과적 문제로 한정할 수는 없을 것이다. 향후 이 문제를 더욱 다양화하고 확대하는 방안에 대해 연구해보며 또한 더욱 효과적인 새로운 예제의 개발에 대해서도 연구하고자 한다. 아울러 시각화 도구나 다중 쓰레드 분석 등 기존 연구에서 개발된 내용을 은행계좌 문제에 적용해보며 동기화에 따른 부담 분석 등으로 연구를 확대해 나갈 것이다.

참고문헌

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts with Java*, 8th edition, John Wiley & Sons, Inc., 2010

[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice Hall, 2011

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th edition, Prentice Hall, 2014

[4] A. D. Fekete, "Teaching about Threading: Where and What?", *ACM SIGACT News*, vol.40, no.1, pp.51-57, Mar 2009

[5] C-K Shene, "Multithreaded Programming Can Strengthen an Operating Systems Course", *Computer Sci. Edu*, vol.12, pp.275-299, 2002

[6] H. Jarvinen, M. Tiusanen, and A. Virtanen, "Convit, a Tool for Learning Concurrent Programming", *Proc of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*, pp.2220-2223, 2003

[7] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)", *Lecture Notes In Computer Sci*, vol. 2269, Springer-Verlag, pp.672-675, 2002

[8] K. Mehner, "JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs", *Lecture Notes In Computer Science*, vol. 2269, Springer-Verlag, pp.163-175, 2002

[9] H. Leroux, A. Requile-Romanczuk, and C. Mingins, "JACOT: A Tool to Dynamically Visualise the Execution of Concurrent Java Programs," *Proc of the 2nd International Conf on Principles and Practice of Programming in Java (PPPJ 2003)*, pp.201-206, 2003

[10] S. Carr, J. Mayo, and C-K Shene, "ThreadMentor: A Pedagogical Tool for

Multithreaded Programming", *ACM Journal on Educational Resources in Computing*, Vol.3, No.1, 2003

[11] G. Malnati, C. M. Cuva, and C. Barberis, "JThreadSpy: Teaching Multithreading Programming by Analyzing Execution Traces", *Proc of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pp.3-13, 2007

[12] S. Robbins, "Experimentation with Bounded Buffer Synchronization", *Proc of the 31st SIGCSE Technical Symp on Computer Science Education*, pp. 330-334, 2000

[13] W. Campbell and E. Bolker, "Teaching Programming by Immersion, Reading and Writing", *Proc of 32nd ASEE/IEEE Frontiers in Education Conference*, Boston, MA, Nov 2002

[14] G. Nutt, *Operating Sustems - A Modern Perspective*, 2nd ed., Addison-Wesley, 2002

[15] S. Oaks and H. Wong, *Java Threads*, O'Reilly & Assoc., Inc, 1997

[16] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17, No. 10. pp.549-557, Oct 1974

[17] E. W. Dijkstra, "Cooperating Sequential Processes", *Technological University, Eindhoven, The Netherlands*, Sept 1965

[18] S. J. Hartley, "Alfonse, Wait Here For My Signal!", *ACM SIGCSE*, pp.58-62, Mar 1999

[19] Accreditation Board for Engineering Education of Korea, http://www.abeek.or.kr

[20] K. Kim, J.Song, and T. Lee, "Effect of Digital Storytelling based Programming Education on Motivation and Achievement of Students in Elementary school", J. of the Korea Society of Computer and Information, vol.14, no. 1, pp.47-55, Jan 2009

저 자 소 개

양 희 재

1985: 부산대학교
전자공학과 공학사.

1987: 한국과학기술원(KAIST)
전기및전자공학과 공학석사

1991: 한국과학기술원(KAIST)
전기및전자공학과 공학박사

1991-현재: 경성대학교
컴퓨터공학부 교수

관심분야: 컴퓨터구조, 임베디드시스템

Email : hjyang@ks.ac.kr