쓰레드 (Thread)

양희재 교수 (hjyang@ks.ac.kr) / 경성대학교 컴퓨터공학과

Thread?

- 쓰레드 (Thread)
  - 프로그램 내부의 흐름, 맥

```
class Test {
 public static void main(String[] args) {
    int n = 0;
    int m = 6;
    System.out.println(n+m);
    while (n < m)
     n++;
    System.out.println("Bye");
 }
```

Multithreads

- 다중 쓰레드 (Multithreads)
  - 한 프로그램에 2개 이상의 맥
  - 맥이 빠른 시간 간격으로 스위칭 된다 ⇒ 여러 맥이 동시에 실행되는 것처럼 보인다 (concurrent *vs* simultaneous)
- 예: Web browser
  - 화면 출력하는 쓰레드 + 데이터 읽어오는 쓰레드
- 예: Word processor
  - 화면 출력하는 쓰레드 + 키보드 입력 받는 쓰레드 + 철자/문법 오류 확인 쓰레드
- 예: 음악 연주기, 동영상 플레이어, Eclipse IDE, …

결국 요즘 os는 프로세스1의 쓰레드1,쓰레드2..로 context switching 하다가 프로세스2의 쓰레드1,쓰레드2..로 switching

Thread vs Process

- 한 프로세스에는 기본 1개의 쓰레드
  - 단일 쓰레드 (single thread) 프로그램
- 한 프로세스에 여러 개의 쓰레드
  - 다중 쓰레드 (multi-thread) 프로그램
- 쓰레드 구조
  - 프로세스의 메모리 공간 공유 (code, data)
  - 프로세스의 자원 공유 (file, i/o, …)
  - 비공유: 개별적인 PC, SP, registers, stack
- 프로세스의 스위칭 vs 쓰레드의 스위칭

예제: 자바 쓰레드

- 맥 만들기
  - java.lang.*Thread*
  - 주요 메소드
    - public void run() // 새로운 맥이 흐르는 곳 (치환)
    - void start() // 쓰레드 시작 요청
    - void join() // 쓰레드가 마치기를 기다림
    - static void sleep() // 쓰레드 잠자기

java.lang.Thread

- Thread.*run()*
  - 쓰레드가 시작되면 run() 메소드가 실행된다

    ⇒ run() 메소드를 치환한다.

```
class MyThread extends Thread {
 public void run() {                    // 치환 (override)
   // 코드
 }
}
```

- 예제: 글자 A 와 B 를 동시에 화면에 출력하기
  - 모든 프로그램은 처음부터 1개의 쓰레드는 갖고 있다 (main)
  - 2개의 쓰레드: main + MyThread

프로세스 동기화

- Process Synchronization
  - *cf.* Thread synchronization
- Processes
  - Independent *vs.* Cooperating
  - Cooperating process: one that *can affect or be affected* by other processes executed in the system
  - 프로세스간 통신: 전자우편, 파일 전송
  - 프로세스간 자원 공유: 메모리 상의 자료들, 데이터베이스 등
  - 명절 기차표 예약, 대학 온라인 수강신청, 실시간 주식거래

프로세스 동기화

- Process Synchronization
  - Concurrent access to shared data may result in data inconsistency
  - *Orderly execution of cooperating processes so that data consistency is maintained*
- Example: BankAccount Problem (은행계좌문제)
  - 부모님은 은행계좌에 입금; 자녀는 출금
  - 입금(deposit)과 출금(withdraw) 은 독립적으로 일어난다.

```
class Test {
 public static void main(String[] args)
throws InterruptedException {
            BankAccount b = new
                        BankAccount();
            Parent p = new Parent(b);
            Child c = new Child(b);
            p.start();
            c.start();
            p.join();
            c.join();
            System.out.println(
    "₩nbalance = " + b.getBalance());
 }
}
```

```
class BankAccount {
 int balance;
 void deposit(int amount) {
            balance = balance + amount;
 }
 void withdraw(int amount) {
            balance = balance - amount;
 }
 int getBalance() {
            return balance;
 }
}
```

```
class Parent extends Thread {
 BankAccount b;
 Parent(BankAccount b) {
            this.b = b;
 }
 public void run() {
            for (int i=0; i<100; i++)
                        b.deposit(1000);
 }
}
```

```
class Child extends Thread {
 BankAccount b;
 Child(BankAccount b) {
            this.b = b;
 }
 public void run() {
            for (int i=0; i<100; i++)
                        b.withdraw(1000);
 }
}
```

BankAccount Problem

- 입출금 동작 알기 위해 "+", "-" 출력하기
- 입출금 동작에 시간 지연 추가
  - 잘못된 결과값
  - 이유: 공통변수(common variable)에 대한 동시 업데이트 (concurrent update)
  - 해결: 한번에 한 쓰레드만 업데이트하도록 → 임계구역 문제

임계구역 문제

- The Critical-Section Problem
- *Critical section*
  - A system consisting of multiple threads
  - Each thread has a segment of code, called critical section, in which *the thread may be changing* common variables, updating a table, writing a file, and so on.
- Solution
  - Mutual exclusion (상호배타): 오직 한 쓰레드만 진입
  - Progress (진행): 진입 결정은 유한 시간 내
  - Bounded waiting (유한대기): 어느 쓰레드라도 유한 시간 내

프로세스/쓰레드 동기화

- 임계구역 문제 해결 (틀린 답이 나오지 않도록)
- 프로세스 실행 순서 제어 (원하는대로)

동기화 도구

- Synchronization Tools
  - Semaphores
  - Monitors
  - Misc.
- Semaphores (세마포)
  - n. (철도의) 까치발 신호기, 시그널; U (군대의) 수기(手旗) 신호
  - 동기화 문제 해결을 위한 소프트웨어 도구
  - 네덜란드의 Edsger Dijkstra 가 제안
  - 구조: 정수형 변수 + 두 개의 동작 (P, V)

세마포 (Semaphore)

- 동작
  - P: Proberen (test) → acquire()
  - V: Verhogen (increment) → release()
- 구조

```
class Semaphore {
        int value;          // number of permits
        Semaphore(int value) {
                ...
        }
        void acquire() {
                ...
        }
        void release() {
                ...
        }
 }
```

세마포 (Semaphore)

```
void acquire() {
        value--;
        if (value < 0) {
                add this process/thread to list;
                block;
        }
}
```

```
void release() {
        value++;
        if (value <= 0) {
                remove a process P from list;
                wakeup P;
        }
}
```

세마포 (Semaphore)

- 일반적 사용 (1): Mutual exclusion
  - sem.value = 1;

sem.acquire();
Critical-Section
sem.release();

- 예제: BankAccount Problem
  - java.util.concurrent.Semaphore

세마포 (Semaphore)

- 일반적 사용 (2): Ordering
  - sem.value = 0;

P_1	P_2
	sem.acquire();
S_1;	S_2;
sem.release();	

- 예제: BankAccount Problem
  - 항상 입금 먼저 (= Parent 먼저)
  - 항상 출금 먼저 (= Child 먼저)
  - 입출금 교대로 (P-C-P-C-P-C- …)

전통적 동기화 예제

Classical Synchronization Problems

전통적 동기화 예제

- Producer and Consumer Problem
  - 생산자-소비자 문제
  - 유한버퍼 문제 (Bounded Buffer Problem)
- Readers-Writers Problem
  - 공유 데이터베이스 접근
- Dining Philosopher Problem
  - 식사하는 철학자 문제

Producer-Consumer Problem

- 생산자-소비자 문제
  - 생산자가 데이터를 생산하면 소비자는 그것을 소비
  - 예: 컴파일러 > 어셈블러, 파일 서버 > 클라이언트, 웹 서버 > 웹 클라이언트
- Bounded Buffer
  - 생산된 데이터는 버퍼에 일단 저장 (속도 차이 등)
  - 현실 시스템에서 버퍼 크기는 유한
  - 생산자는 버퍼가 가득 차면 더 넣을 수 없다.
  - 소비자는 버퍼가 비면 뺄 수 없다.

```
class Buffer {
 int[]      buf;
 int        size;
 int        count;
 int        in;
 int        out;

Buffer(int size) {
           buf = new int[size];
           this.size = size;
           count = in = out = 0;
 }
```

```
 void insert(int item) {
           /* check if buf is full */
           while (count == size)
                      ;

           /* buf is not full */
           buf[in] = item;
           in = (in+1)%size;
           count++;
 }
```

```
 int remove() {
           /* check if buf is empty */
           while (count == 0)
                      ;

           /* buf is not empty */
           int item = buf[out];
           out = (out+1)%size;
           count--;
           return item;
  }
}
```

```
class Buffer {
 int[]      buf;
 int        size;
 int        count;
 int        in;
 int        out;

Buffer(int size) {
           buf = new int[size];
           this.size = size;
           count = in = out = 0;
 }
```

```
 void insert(int item) {
           /* check if buf is full */
           while (count == size)
                      ;

           /* buf is not full */
           buf[in] = item;
           in = (in+1)%size;
           count++;
 }
```

```
 int remove() {
           /* check if buf is empty */
           while (count == 0)
                      ;

           /* buf is not empty */
           int item = buf[out];
           out = (out+1)%size;
           count--;
           return item;
  }
}
```

```
/****** 생산자 ******/
class Producer extends Thread {
 Buffer b;
 int N;
 Producer(Buffer b, int N) {
            this.b = b; this.N = N;
 }
 public void run() {
            for (int i=0; i<N; i++)
                        b.insert(i);
 }
}
```

```
/****** 소비자 ******/
class Consumer extends Thread {
 Buffer b;
 int N;
 Consumer(Buffer b, int N) {
            this.b = b; this.N = N;
 }
 public void run() {
            int item;
            for (int i=0; i<N; i++)
                        item = b.remove();
 }
}
```

```
class Test {
 public static void main(String[] arg) {
            Buffer b = new Buffer(100);
            Producer p = new Producer(b, 10000);
            Consumer c = new Consumer(b, 10000);
            p.start();
            c.start();
            try {
                        p.join();
                        c.join();
            } catch (InterruptedException e) {}
            System.out.println("Number of items in the buf is " + b.count);
 }
}
```

Producer-Consumer Problem

- 잘못된 결과
  - 실행 불가, 또는
  - count ≠ 0 (생산된 항목 숫자 ≠ 소비된 항목 숫자)
  - 최종적으로 버퍼 내에는 0 개의 항목이 있어야
- 이유
  - 공통변수 count, buf[] 에 대한 동시 업데이트
  - 공통변수 업데이트 구간(= 임계구역)에 대한 동시 진입
- 해결법
  - 임계구역에 대한 동시 접근 방지 (상호배타)
  - 세마포를 사용한 상호배타 (mutual exclusion)
  - 세마포: mutex.value = 1 (# of permit) ☞ *코드 보기*

Producer-Consumer Problem

- Busy-wait
  - 생산자: 버퍼가 가득 차면 기다려야 = 빈(empty) 공간이 있어야
  - 소비자: 버퍼가 비면 기다려야 = 찬(full) 공간이 있어야
- 세마포를 사용한 busy-wait 회피 ☞ *코드 보기*
  - 생산자: empty.acquire() // # of permit = BUF_SIZE
  - 소비자: full.acquire() // # of permit = 0

[생산자]	[소비자]
empty.acquire(); **PRODUCE;** **full.release();**	**full.acquire();** **CONSUME;** **empty.release();**

Readers-Writers Problem

- 공통 데이터베이스
  - *Readers*: read data, never modify it
  - *Writers*: read data and modifiy it
  - 상호배타: 한 번에 한 개의 프로세스만 접근 ☞ 비효율적
- 효율성 제고
  - Each read or write of the shared data *must happen within a critical section*
  - Guarantee *mutual exclusion for writers*
  - Allow *multiple readers to execute in the critical section at once*
- 변종
  - The first R/W problem (readers-preference)
  - The second R/W problem (writers-preference)
  - The Third R/W problem

Dining Philosopher Problem

- 식사하는 철학자 문제
  - 5명의 철학자, 5개의 젓가락, 생각 → 식사 → 생각 → 식사 …
  - 식사하려면 2개의 젓가락 필요
- 프로그래밍
  - 젓가락: 세마포 (# of permit = 1)
  - 젓가락과 세마포에 일련번호: 0 ~ 4
  - 왼쪽 젓가락 → 오른쪽 젓가락

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

class Philosopher extends Thread {
 int id;                                          // philosopher id
 Semaphore lstick, rstick;                        // left, right chopsticks

 Philosopher(int id, Semaphore lstick, Semaphore rstick) {
            this.id = id;
            this.lstick = lstick;
            this.rstick = rstick;
 }
```

```
 public void run() {
            try {
            while (true) {
                        lstick.acquire();
                        rstick.acquire();
                        eating();

                        lstick.release();
                        rstick.release();
                        thinking();
            }
            }catch (InterruptedException e) { }
 }
```

```
 void eating() {
  System.out.println("[" + id + "] eating");
 }
 void thinking() {
  System.out.println("[" + id + "] thinking");
 }
}
```

```
class Test {
 static final int num = 5;                    // number of philosphers & chopsticks

 public static void main(String[] args) {
            int i;

            /* chopsticks */
            Semaphore[] stick = new Semaphore[num];
            for (i=0; i<num; i++)
                        stick[i] = new Semaphore(1);

            /* philosophers */
            Philosopher[] phil = new Philosopher[num];
            for (i=0; i<num; i++)
                        phil[i] = new Philosopher(i, stick[i], stick[(i+1)%num]);

            /* let philosophers eat and think */
            for (i=0; i<num; i++)
                        phil[i].start();
 }
}
```

Dining Philosopher Problem

- 잘못된 결과: starvation
  - 모든 철학자가 식사를 하지 못해 굶어 죽는 상황
- 이유 = 교착상태 (deadlock)

교착상태

Deadlocks

Deadlock

- 프로세스는 실행을 위해 여러 자원을 필요로 한다.
  - CPU, 메모리, 파일, 프린터, ……
  - *어떤 자원은 갖고 있으나 다른 자원은 갖지 못할 때* (e.g., 다른 프로세스가 사용 중) 대기해야
  - 다른 프로세스 역시 다른 자원을 가지려고 대기할 때 *교착상태 가능성!*
- 교착상태 필요 조건 (Necessary Conditions)
  - Mutual exclusion (상호배타)
  - Hold and wait (보유 및 대기)
  - No Preemption (비선점)
  - Circular wait (환형대기)

자원 (Resources)

- 동일 자원
  - 동일 형식 (type) 자원이 여러 개 있을 수 있다 (instance)
  - 예: 동일 CPU 2개, 동일 프린터 3개 등
- 자원의 사용
  - 요청 (request) → 사용 (use) → 반납 (release)
- 자원 할당도 (Resource Allocation Graph)
  - 어떤 자원이 어떤 프로세스에게 할당되었는가?
  - 어떤 프로세스가 어떤 자원을 할당 받으려고 기다리고 있는가?
  - 자원: 사각형, 프로세스: 원, 할당: 화살표

자원 할당도

- 교착상태 필요조건
  - 자원 할당도 상에 원이 만들어져야 (환형대기)
  - *충분조건은 아님!*
- 예제: 식사하는 철학자 문제
  - *원이 만들어지지 않게 하려면?*

교착상태 처리

- 교착상태 방지
  - Deadlock Prevention
- 교착상태 회피
  - Deadlock Avoidance
- 교착상태 검출 및 복구
  - Deadlock Detection & Recovery
- 교착상태 무시
  - Don't Care

(1) 교착상태 방지

- Deadlock Prevention
- 교착상태 4가지 필요조건 중 한 가지 이상 불만족
  - 상호배타 (Mutual exclusion)
  - 보유 및 대기 (Hold and wait)
  - 비선점 (No preemption)
  - 환형 대기 (Circular wait)

(1) 교착상태 방지

- 상호배타 (Mutual exclusion)
  - 자원을 공유 가능하게; 원천적 불가할 수도
- 보유 및 대기 (Hold & Wait)
  - 자원을 가지고 있으면서 다른 자원을 기다리지 않게
  - 예: 자원이 없는 상태에서 모든 자원 대기; 일부 자원만 가용하면 보유 자원을 모두 놓아주기
  - 단점: 자원 활용률 저하, 기아 (starvation)
- 비선점 (No preemption)
  - 자원을 선점 가능하게; 원천적 불가할 수도 (예: 프린터)
- 환형대기 (Circular wait)
  - 예: 자원에 번호부여; 번호 오름차순으로 자원 요청
  - 단점: 자원 활용률 저하

(2) 교착상태 회피

- Deadlock Avoidance
  - 교착상태 = 자원 요청에 대한 잘못된 승인 (≒ 은행 파산)
- 예제
  - 12개의 magnetic tape 및 3개의 process
  - 안전한 할당 (Safe allocation)

Process	Max needs	Current needs
P_0	10	5
P_1	4	2
P_2	9	2

(2) 교착상태 회피

- 예제 (계속)
  - 12개의 magnetic tape 및 3개의 process
  - 불안전한 할당 (Unsafe allocation)

Process	Max needs	Current needs
P_0	10	5
P_1	4	2
P_2	9	3

- 운영체제는 자원을 할당할 때 불안전 할당 되지 않도록
  - 불안전 할당 → 교착상태
  - 대출전문 은행과 유사: Banker's Algorithm

(3) 교착상태 검출 및 복구

- Deadlock Detection & Recovery
  - 교착상태가 일어나는 것을 허용
  - 주기적 검사
  - 교착상태 발생 시 복구
- 검출
  - 검사에 따른 추가 부담 (overhead): 계산, 메모리
- 복구
  - 프로세스 일부 강제 종료
  - 자원 선점하여 일부 프로세스에게 할당

(4) 교착상태 무시

- 교착상태는 실제로 잘 일어나지 않는다!
  - 4가지 필요조건 모두 만족해도 …
  - 교착상태 발생 시 재시동 (PC 등 가능)

모니터

Monitors

모니터

- 모니터 (Monitor)
  - 세마포 이후 프로세스 동기화 도구
  - 세마포 보다 고수준 개념
- 구조
  - 공유자원 + 공유자원 접근함수
  - 2개의 queues: 배타동기 + 조건동기
  - 공유자원 접근함수에는 최대 1개의 쓰레드만 진입
  - 진입 쓰레드가 조건동기로 블록되면 새 쓰레드 진입가능
  - 새 쓰레드는 조건동기로 블록된 쓰레드를 깨울 수 있다.
  - 깨워진 쓰레드는 현재 쓰레드가 나가면 재진입할 수 있다.

자바 모니터

- 자바의 모든 객체는 모니터가 될 수 있다.
  - 배타동기: synchronized 키워드 사용하여 지정
  - 조건동기: wait(), notify(), notifyAll() 메소드 사용

```
class C {
 private int value, ...;
 synchronized void f() {
            ...
 }
 synchronized void g() {
            ...
 }
 void h() {
            ...
 }
}
```

모니터 (Monitor)

- 일반적 사용 (1): Mutual exclusion

```
synchronized {

    Critical-Section

}
```

- 예제: BankAccount Problem ☞ 뒷면

```
class Test {
 public static void main(String[] args)
throws InterruptedException {
            BankAccount b = new
                        BankAccount();
            Parent p = new Parent(b);
            Child c = new Child(b);
            p.start();
            c.start();
            p.join();
            c.join();
            System.out.println(
    "₩nbalance = " + b.getBalance());
 }
}
```

```
class BankAccount {
 int balance;
 synchronized void deposit(int amt) {
            int temp = balance + amt;
            System.out.print("+");
            balance = temp;
 }
 synchronized void withdraw(int amt) {
            int temp = balance - amt;
            System.out.print("-");
            balance = temp;
 }
 int getBalance() {
            return balance;
 }
}
```

```
class Parent extends Thread {
 BankAccount b;
 Parent(BankAccount b) {
            this.b = b;
 }
 public void run() {
            for (int i=0; i<100; i++)
                        b.deposit(1000);
 }
}
```

```
class Child extends Thread {
 BankAccount b;
 Child(BankAccount b) {
            this.b = b;
 }
 public void run() {
            for (int i=0; i<100; i++)
                        b.withdraw(1000);
 }
}
```

모니터 (Monitor)

- 일반적 사용 (2): Ordering

P_1	P_2
	wait();
S_1;	S_2;
notify();	

- 예제: BankAccount Problem
  - 항상 입금 먼저 (= Parent 먼저)
  - 항상 출금 먼저 (= Child 먼저)
  - 입출금 교대로 (P-C-P-C-P-C- …)

전통적 동기화 예제

- Producer and Consumer Problem
  - 생산자-소비자 문제
  - 유한버퍼 문제 (Bounded Buffer Problem)
- Readers-Writers Problem
  - 공유 데이터베이스 접근
- Dining Philosopher Problem
  - 식사하는 철학자 문제

```
class Buffer {
 int[]      buf;
 int        size, count, in, out;

Buffer(int size) {
           buf = new int[size];
           this.size = size;
           count = in = out = 0;
 }
```

```
 synchronized void insert(int item) {
           while (count == size)
           try {
                      wait();
           } catch (InterruptedException e) {}

           buf[in] = item;
           in = (in+1)%size;
           notify();
           count++;
 }
```

```
 synchronized int remove() {
           while (count == 0)
           try {
                      wait();
           } catch (InterruptedException e) {}

           int item = buf[out];
           out = (out+1)%size;
           count--;
           notify();
           return item;
  }
 }
```

The Dining Philosopher Problem

```
class Chopstick {
 private boolean inUse = false;
 synchronized void acquire() throws InterruptedException {
            while (inUse)
                        wait();
            inUse = true;
 }
 synchronized void release() {
            inUse = false;
            notify();
 }
}
```