7장. 동기화 예제 (Synchronization Examples)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

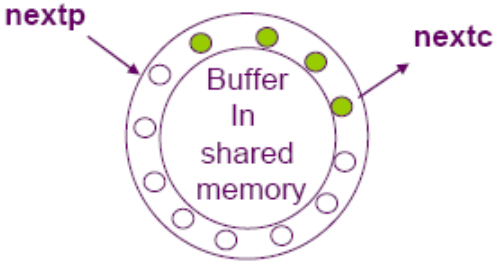
- 목표
  - 고전적인 동기화 문제들 소개
  - Linux 및 POSIX 동기화
  - 동기화 대체 방안들
- 내용
  - 고전적인 동기화 문제들
    - 유한 버퍼, ,readers-writer, 식사하는 철학자들 문제
  - Linux 및 POSIX 동기화
  - 동기화 대체 방안들

고전적인 동기화 문제들 (Classic Problems of Synchronization)

- 널리 사용되는 대표적인 동기화 문제들을 제시
  - 새로 제안된 거의 모든 동기화 방법들을 테스트하는 데 사용
  - 여기서는 동기화를 위하여 세마포가 사용
- 유한 버퍼 문제 (Bounded-Buffer Problem)
- Readers-Writers 문제 (Readers-Writers Problem)
- 식사하는 철학자들 문제 (Dining-Philosophers Problem)

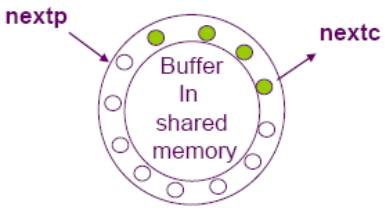
유한 버퍼 문제 (Bounded-Buffer Problem)

- n 개의 버퍼들로 구성된 풀(pool)이 있으며 각 버퍼들은 한 항목(item)을 저장
- mutex 세마포는 버퍼 풀을 접근하기 위한 상호 배제 기능을 제공하며 1로 초기화
- empty 세마포는 세마포들은 비어 있는 버퍼의 수를 기록하며 n 값으로 초기화
  - 0이면 버퍼가 차있음을 의미
- full 세마포는 채워진 버퍼의 수를 기록하며 0으로 초기화
  - 0이면 버퍼가 비어 있음을 의미



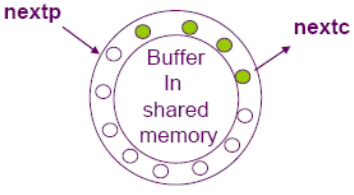
유한 버퍼 문제 - 생산자 프로세스

```
do {
      ...
       /* produce an item in next_produced */
      ...
    wait(empty);
    wait(mutex);
       ...
       /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
} while (true);
```



유한 버퍼 문제 - 소비자 프로세스

```
do {
  wait(full);
  wait(mutex);
     ...
     /* remove an item from buffer to next_consumed */
     ...
  signal(mutex);
  signal(empty);
     ...
     /* consume the item in next consumed */
     ...
} while (true);
```



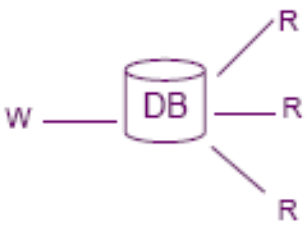
유한 버퍼 문제 - 생산자,소비자

```
do {
        ...
    // produce an item in nextp
        ...
    wait(empty);
    wait(mutex);
        ...
    // add nextp to buffer
        ...
    signal(mutex);
    signal(full);
} while (TRUE);
```

```
do {
    wait(full);
    wait(mutex);
        ...
    // remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    // consume the item in nextc
        ...
} while (TRUE);
```


Readers-Writers 문제 (Readers-Writers Problem)

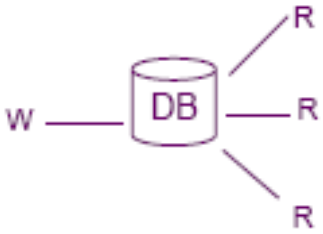
- 하나의 데이터베이스는 다수의 병행 프로세스들 간에 공유
  - 일부 프로세스들은 데이터베이스의 내용을 읽기만 수행
    => Readers
  - 어떤 프로세스들은 데이터베이스를 갱신(즉, 읽고 쓰기) 수행
    => Writers
- 문제
  - 하나 이상의 reader가 동시에 공유 자료를 접근 허용
  - 오직 하나의 writer만 공유 자료에 접근
    - writer와 다른 프로세스(write 또는 reader) 동시 접근 불능



Readers-Writers 문제, 공유 자료

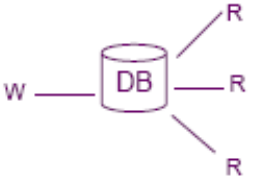
❑ 공유 자료

- 데이터 세트
- mutex 세마포
  - 1로 초기화
  - readcount를 갱신할 때 상호 배제
- rw_mutex 세마포
  - 1으로 초기화
  - writer를 위한 상호 배제 세마포
  - 또한 임계 구역으로 진입하는 첫 번째 reader와, 임계 구역을 빠져 나오는 마지막 reader에 의해서도 사용
- readcount 정수
  - 현재 몇 개의 프로세스들이 객체를 읽고 있는지 알려줌
  - 0으로 초기화



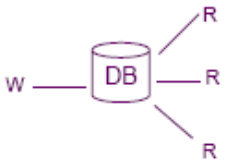
writer 프로세스

```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
        signal(rw_mutex);
} while (true);
```



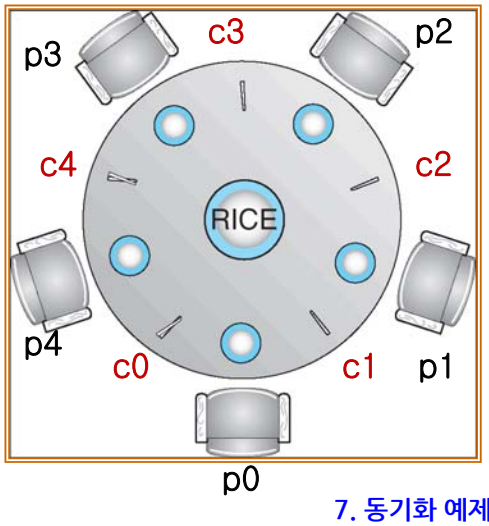
reader 프로세스

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        ...
    /* reading is performed */
        ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```



식사하는 철학자들 문제 (1) (Dining-Philosophers Problem)

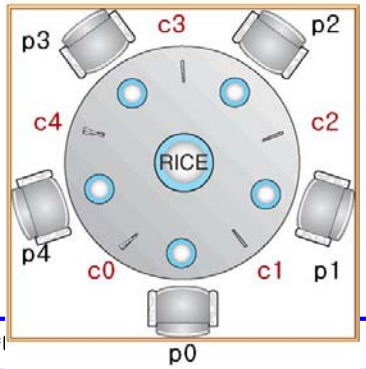
- 5명의 배고픈 철학자가 가장 가까이 있는 두 개의 젓가락(왼쪽/오른쪽)를 집어야만 식사하는 예
  - 철학자들은 사색과 식사로만 살아감
  - 옆자리의 상대방과 소통하지 않고, 그릇의 밥을 먹기 위해 2개의 젓가락(한 번에 하나씩)을 사용



- 공유 자료
  - 데이터 세트 (밥 그릇)
  - 세마포 chopstick[5] (젓가락)
    - 1로 초기화

식사하는 철학자들 문제 (2)

- 문제는 교착 상태 (deadlock)
  - 5명의 철학자 모두가 동시에 자신의 왼쪽 젓가락을 잡는 경우
  - 모든 chopstick이 0이 되어 영원히 서로 기다림



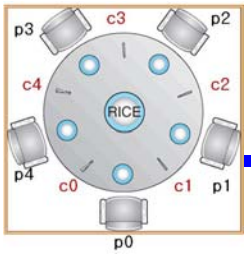

```
do {
      wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

      //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

     //  think

} while (TRUE);
```

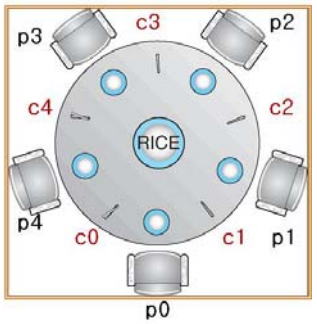



식사하는 철학자들 문제 (3)

- 교착상태 문제에 대한 해결책
  - 최대 4명의 철학자들만이 테이블에 동시에 앉을 수 있음
  - 한 철학자가 두 개의 젓가락을 모두 사용 가능할 때만 젓가락을 잡도록 허용 (임계영역 안에서만 젓가락을 잡어야 함)
  - 비대칭 해결 안을 사용
    - 홀수 번호의 철학자들은 먼저 왼쪽 젓가락을 잡고 다음에 오른쪽 젓가락을 집음
    - 반면에 짝수 번호의 철학자는 오른쪽 젓가락을 잡고 다음에 왼쪽 젓가락을 집음
- 교착상태의 해결안이 기아의 가능성도 제거하는 것은 아님
  - 따라서 만족할만한 해결안은 교착상태가 발생하지 않으면서 계속 기다리다가 굶어 죽는 철학자가 생기지 않도록 방지해야 함

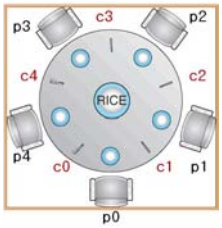
모니터를 사용한 식사하는 철학자 해결안

- 모니터를 사용하여 식사하는 철학자 문제에 대한 교착 상태가 없는 해결안을 제시
  - 철학자는 양쪽 젓가락 모두 얻을 수 있을 때만 젓가락을 집을 수 있다는 제한을 강제
  - 세가지 상태 자료구조
    enum { thinking, eating, hungry } state [5];
  - 철학자 i는 그의 양쪽 두 이웃이 식사하지 않을 때만 변수 state [i] = eating으로 설정
    - 왼쪽 이웃 조건 state [(i + 4) % 5]!= eating
    - 오른쪽 이웃 조건 state [(i + 1) % 5]!= eating
  - 조건 변수 선언
    condition self [5];




```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
       state[i] = hungry; /* 3rd state */
       test(i); /* state of neighbors? */
       if (state[i] != eating) /*two cases*/
            self[i].wait(); /*wait here*/
  }

  void   test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
         state[i] = eating;
         self[i].signal();   / *=no_op. I'm
                               already running */
    }
  }
  void   init() {
    for (int i = 0; i < 5; i++)
         state[i] = thinking;
  }
}
```



Program using Monitor

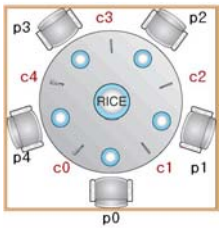
```
Each Philosopher:
{     pickup(i);
      eat();
      putdown(i);
      think();
} while(1)
```

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating) /*two cases*/
              self[i].wait(); /*wait here*/
  }

  }

  void   test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *=no_op. I'm
                                already running */
    }
  }
  void   init() {
    for (int i = 0; i < 5; i++)
          state[i] = thinking;
  }
```

Case 1 – state[i] = eating
pick up & proceed



Program using Monitor

```
Each Philosopher:
{     pickup(i);
      eat();
      putdown(i);
      think();
} while(1)
```

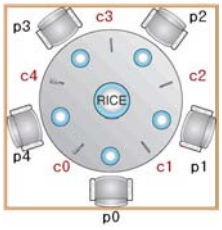

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating) /*two cases*/
              self[i].wait(); /*wait here*/
  }

  }

  void   test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *=no_op. I'm
                                already running */
    }
  }
  void   init() {
    for (int i = 0; i < 5; i++)
          state[i] = thinking;
  }
```


Condition variable – queue process here

Case 2 – state[i] != eating
block myself
CPU → other process

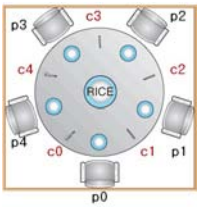


Program using Monitor

```
Each Philosopher:
{     pickup(i);
      eat();
      putdown(i);
      think();
} while(1)
```

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating)
              self[i].wait(); /*wait here*/
  }

  void   putdown(int i) {
        state[i] = thinking;
        // test left and right neighbors
        test((i+4) % 5); /*if L is waiting*/
        test((i+1) % 5);
  }
  }
```



Can (Left eat now) && (Left was blocked)?

```
void   test (int i) {
  if ( (state[(i + 4) % 5] != eating) &&
    (state[i] == hungry) &&
    (state[(i + 1) % 5] != eating)){ /OK/
        state[i] = eating;
        self[i].signal();   / *wakeup  Pi */
    }          /* used for putdown L & R */
}
void   init() {
  for (int i = 0; i < 5; i++)
        state[i] = thinking;
}
```

Program using Monitor

```
Each Philosopher:
{       pickup(i);
        eat();
        putdown(i);
        think();
} while(1)
```


Linux의 동기화 (1)

- Linux 버전 2.6 부터 선점 가능 커널
  - 커널 모드에서 실행 중일 때에도 태스크는 선점될 수 있음
  - 이전 버전은 선점 불가능 커널
    - 커널 모드에서 실행중인 프로세스는 더 높은 우선순위의 프로세스가 실행 가능한 상태가 되더라도 선점될 수 없었음

- Linux는 커널 안의 동기화를 위해 여러 기법 제공
  - 원자적 정수 (atomic integer)
  - 뮤텍스 락 (mutex lock)
  - 스핀 락 (spinlock)
  - 세마포 (semaphore)
  - .....

Linux의 동기화 (2)

❑ 원자적 정수 (atomic integer)

- Linux 커널 안에서 가장 간단한 동기화 기법
- 원자적 정수를 사용하는 모든 수학 연산은 중단 없이(인터럽트 없이) 원자적으로 수행

```
atomic_t counter;
int value;

atomic_set(&counter, 5);              // counter = 5
atomic_add(10, &counter);             //  counter = counter + 10 = 15
atomic_sub(4, &counter);              // counter = counter – 4 = 11
atomic_inc(&counter);                 // counter = counter + 1= 12
value = atomic_read(&counter);        // value = 12
```


Linux의 동기화 (3)

❑ 뮤텍스 락 (mutex lock)

- Linux에서 커널 안의 임계구역 보호
- 임계구역 진입 시 mutex_lock() 호출
- 임계구역 퇴출 시 mutex_unlock() 호출

❑ 스핀락(spinlock)

- SMP 기계에서는 기본적인 락킹 기법
- 단일 처리기에서는 스핀락을 사용하는 것은 부적합
  - 커널 선점 불능 및 가능으로 대체
- 스핀락(커널 선점 불능 및 가능 또한)은 락(또는 커널 불능 기간)이 짧은 시간 동안만 유지될 때 사용

❑ 세마포(semaphore)

POSIX 동기화

- POSIX의 Pthreads API는 운영체제에 독립적인 API
  - 뮤텍스 락 (mutex locks)
    - Pthread에서 사용할 수 있는 기본적인 동기화 기법으로 Mutex 락은 코드의 임계 구역을 보호하기 위해 사용
  - 세마포 (semaphore)
    - 기명(named)와 무명(unnamed) 세마포 제공
  - 조건 변수 (condition variables)
  - read-write 락 (read-write locks)
  - 스핀락 (spin locks)

POSIX 뮤텍스 락

- pthread_mutex_t 데이터 형
- pthread_mutex_init(&mutex, NULL) 함수로 생성
  - 첫 번째 매개변수는 mutex를 가리키는 포인터
  - 두 번째 매개변수는 속성을 표시하며, NULL은 디폴트 속성
- mutex의 획득과 방출은 pthread_mutex_lock()과 pthread_mutex_unlock() 함수에 의해 수행
- 모든 mutex 관련 함수들은 성공적인 실행 시 0을 반환

```
#include <pthreads.h>
pthreads_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX 무명 세마포 (1)

❑ 세마포 생성

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);
```

- sem_init() 함수는 세마포를 생성하고 초기화하고, 다음 세 개의 매개변수가 전달
  - 세마포를 가리키는 포인터
  - 공유 수준을 나타내는 플래그
    - 0 이면 세마포를 생성한 프로세스에 속한 스레드들만이 공유
  - 세마포의 초기값

POSIX 무명 세마포 (2)

❑ Pthreads에서 wait()와 signal() 연산은 sem_wait()와 sem_post()

```
#include <semarphore.h>
sem_t sem;

// 세마포 생성 및 초기화
sem_init(&sem, 0, 1);

// 세마포 획득
sem_wait(&sem);

// 임계 영역
……
……
// 세마포 해제
sem_post(&sem);
```

대체 방안들

- 다중 코어 시스템에서는 병렬(병행) 응용 개발로 성능 극대화
  - 코어의 개수가 증가함에 따라 경쟁 조건과 교착 상태 없는 다중 스레드 응용 개발이 점점 더 어려워짐
  - 전통적인 뮤텍스 락, 세마포, 모니터 등의 기법의 적용이 어려워짐
- 트랜잭션 메모리, OpenMP, 함수형 프로그래밍 언어 등의 대체 방안들 소개

트랜잭션 메모리 (Transactional Memory) (1)

- 원자적 트랜잭션 (Atomic Transaction)
  - 임계영역의 상호 배제는 임계영역이 원자적으로 실행되는 것을 보장
  - 중단되지 않는 하나의 단위로 실행되는 것을 보장
  - 원자적 트랜잭션 예
    - 은행의 자금 이체
      - 입금, 출금 트랜잭션
    - 데이터베이스 시스템
      - 데이터의 저장과 검색
      - 데이터 일관성 보장
- 메모리 트랜잭션 (Memory Transaction)
  - 메모리 읽기와 쓰기 연산의 원자적인 연속적 순서
  - 한 트랜잭션의 모든 연산이 완수되면 메모리 트랜잭션은 확정 (commit)
  - 모든 연산이 완수되지 못하면 그 시점까지 완수된 연산들이 취소되고, 트랜잭션 시작 이전 상태로 되돌림 (roll-back)

트랜잭션 메모리 (2)

- 공유 데이터를 수정하는 update() 함수 예

```
// 뮤텍스 락 세마포 구현 예

void  update()
{
    acquire()

    /* 공유 데이터 변경 */

    release()
}
```

```
// 트랜잭션 메모리 구현 예
// 언어에서 atomic { S } 구조물 제공 가정

void  update()
{
    atomic {
        /* 공유 데이터 변경 */
    }
}
```

- 전통적인 동기화 기법들은 스레드 개수가 증가할 수록 경쟁 수준이 매우 높아서 비효율적이고 개발이 어려움
- 트랜잭션 메모리는 개발자가 아닌 트랜잭션 메모리 시스템이 원자성을 보장
  - 락이 사용되지 않으므로 교착 상태 발생하지 않음
  - 트랜잭션 메모리는 소프트웨어 또는 하드웨어 로 구현

OpenMP

OpenMP

- 병렬 프로그래밍을 지원하는 컴파일러의 지시(디렉티브)와 API를 사용하여 임계영역을 원자적으로 실행
- 스레드의 생성과 관리가 OpenMP 라이브러리에 의해서 처리
  - 응용 개발자 부담 없음

```
void update(int value)
{
      #pragma omp critical
      {
            count += value
      }
}
```

  - 임계구역 컴파일러 디렉티브가 락처럼 동작하여 하나의 스레드만 임계구역의 실행을 보장

함수형 프로그래밍 언어 (Functional Programming language)

- 함수형 프로그래밍 언어 (Functional Programming Language)
  - 절차형 언어와는 다른 새로운 프로그래밍 패러다임의 언어
  - 다중 코어 시스템에서 병행 및 병렬 프로그래밍이 주목 받으면서 함수형 프로그래밍에 대한 관심 증대
  - 함수형 언어는 상태를 유지하지 않고, 변경 가능 상태를 허용하지 않기 때문에 경쟁 조건이나 교착 상태와 같은 근본적인 쟁점이 없음
    - 함수의 변수의 값이 지정되면 불변이며 상태 값을 바꿀 수 없음
  - 얼랑(Erlang), 스칼라(Scala)
    - 스칼라는 JVM 상에 동작하는 함수형 언어이면서 객체지향 언어

실습과제 - 유한버퍼 생산자/소비자 문제

- 유한 버퍼의 생산자, 소비자 프로그램을 자바 모니터의 동기화 기법을 적용한 멀티스레드 프로그램으로 변환하여 작성 (p.335 그림 7.9, p.337 그림 7.11 참조)
  - 생산 및 소비되는 데이터는 임의의 응용 데이터
  - 소스 프로그램 및 설명
  - 실행 예