Slide 0

# 2부 프로세스 관리
# (7장. 프로세스 동기화)

## 배경

- 4장의 공유메모리 예)
  - BUFFER_SIZE-1 개의 버퍼만 사용
  - BUFFER_SIZE 만큼 사용하도록 수정

Slide 1

```
#define BUFFER_SIZE 10
Typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

```
item nextProduced;
while (1) {
        while (counter == BUFFER_SIZE)
        ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
item nextConsumed;
while (1) {
        while (counter == 0)
        ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

## 배경 (Cont'd)

Slide 2

- counter++ 와 counter−−의 실제 기계어 수준의 동작
  - counter++

| |
|---|
| $register_1 = \text{counter}$ |
| $register_1 = register_1 + 1$ |
| $\text{counter} = register_1$ |

  - counter−−

| |
|---|
| $register_2 = \text{counter}$ |
| $register_2 = register_2 - 1$ |
| $\text{counter} = register_2$ |

  - 실행 순서에따라서 결과가 달라짐 →경합상태 (race condition)

| | | | | |
|---|---|---|---|---|
| $T_0$: | $producer$ | execute | $register_1 = \text{counter}$ | $register_1 = 5$ |
| $T_1$: | $producer$ | execute | $register_1 = register_1 + 1$ | $register_1 = 6$ |
| $T_2$: | $consumer$ | execute | $register_2 = \text{counter}$ | $register_2 = 5$ |
| $T_3$: | $consumer$ | execute | $register_2 = register_2 - 1$ | $register_2 = 4$ |
| $T_4$: | $producer$ | execute | $\text{counter} = register_1$ | $counter = 6$ |
| $T_5$: | $consumer$ | execute | $\text{counter} = register_2$ | $counter = 4$ |

## 배경 (Cont'd)

Slide 3

- 증상
  - 실제로는 counter는 5가 되어야하는데 counter가 4로 종료 (오류)
  - $T_4 \leftrightarrow T_5$ 가 바뀌었다면 counter가 6으로 종료 (오류)
- 문제원인 및 해결
  - 원인: 두개의 프로세스가 동시에 변수 counter를 조작했기 때문
  - 문제: 경합상태 (race condition) 발생
    →동시에 여러개의 프로세스가 동일한 자료를 접근 조작하여 실행 결과가 순서에 의존하는 것을 말함
  - 해결: 한 순간에 하나의 프로세스만 counter 변수를 조작하도록 보장해야함 (이것을 위해 프로세스 동기화가 필요)
- 또 다른 해결 방법
  - counter++ 와 counter−−를 원자적 (atomic operation)으로 수행
  - 원자적이란 그 동작의 수행이 인터럽트되지 않음을 뜻함
  - 공유변수가 여러개이면 동작이 원자적이어도 문제 발생

## 임계구역 (Critical Section) 문제

Slide 4

- 임계구역
  - 프로세스 코드의 일부분
  - 다른 프로세스와 공동으로 사용하는 변수/테이블/파일 등을 변경하는 코드 부분
- 제한조건
  - 하나의 프로세스가 임계구역에서 작업 중이면
  - 다른 프로세스가 임계구역을 실행하지 않도록 막아야함
    ➥시간적으로 상호배타적 (mutually exclusive)이어야함

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 5

- 임계구역 문제 해결
  - 임계구역에 진입하기전 허가 요청 (이를 진입구역 (entry section)이라함)
  - 허가받은 프로세스가 실행
  - 다른 프로세스가 임계구역 실행할 수 있도록 해제 (이를 출구구역 (exit section)이라함)
  - 임계구역 이외에 구역을 잔류구역 (remainder section) 이라함
- 임계구역

```
do {
        entry section
            critical section
        exit section
            remainder section
} while(1);
```

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 6

- 임계구역을 해결하는 3가지 메커니즘
  - 상호배제 (mutual exclusive)
    - 한 프로세스가 임계구역에서 실행하고 있으면
    - 어떤 프로세스도 임계구역에 진입할 수 없어야 함
  - 진행 (progress)
    - 임계구역을 실행하는 프로세스가 없을 때
    - 몇개의 프로세스가 임계구역에 진입하고자 한다면
    - 이들의 진입순서는 이들에 의해서만 결정되어야하며
    - 또한 이 결정이 무한정 연기되어서는 안됨
  - 한계대기 (bounded waiting)
    - 한 프로세스가 임계구역에 진입 요청을 한 후
    - 이 요청이 허용될 때까지
    - 다른 프로세스가 임계구역에 진입할 수 있는 횟수에 한계가 있어야 함

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 7

- 두 프로세스에서의 해결
  - 가정
    - 두 프로세스: $P_0$ (혹은 $P_i$), $P_1$ (혹은 $P_j$) 단 $j = 1 - i$
    - 기본적인 기계 명령어 (load, store, test)는 원자적임
  - 알고리즘 1
    - 공유변수: int *turn*;
    - 초기값: *turn* = 1; /* or 0 */
    - 동작
      - *turn*이 $i$이면 $P_i$ 가 임계구역에 진입할 수 있음
      - *turn*이 $j$이면 $P_j$ 가 임계구역에 진입할 수 있음

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 8**

```
do {
        while (turn != i) ;
            critical section
        turn = j
            remainder section
} while(1);
```

* 설명
  - 상호배제는 만족
  - 진행은 만족하지 못함
    - ➙ *turn* 이 0이라면
    - ➙ $P_0$가 잔류구역에 있더라도
      (즉 임계구역을 실행하는 프로세스가 없음)
    - ➙ 진입할 준비가 되어있는 $P_1$의 임계구역 진입이 불가능 함

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 9**

- 알고리즘 2
  * 공유변수: boolean *flag*[2];
  * 초기값: *flag*[0] = *flag*[1] = false;
  * 동작
    - *flag*[0]이 true이면 $P_1$ 이 임계구역 진입할 수 없음
    - *flag*[1]이 true이면 $P_0$ 이 임계구역 진입할 수 없음

```
do {
        flag[i] = true;
        while (flag[j]) ;
            critical section
        flag[i] = false;
            remainder section
} while(1);
```

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 10

* 설명
  · 상호배제는 만족하나 진행조건은 충족하지 못함
  · 특히 다음과 같은 상태일 때는 $P_0$ 와 $P_1$은 모두 영원히 기다림

| | | |
|---|---|---|
| $T_0$: | $P_0$ | *flag*[0] = true; |
| $T_1$: | $P_1$ | *flag*[1] = true; |

  · 다음 처럼 바꾸면 상호배제가 만족되지 못함

```
while (flag[j]) ;
flag[i] = true;
```

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 11

– 알고리즘 3
  * 공유변수
    boolean *flag*[2];
    int *turn*;
  * 초기값
    *flag*[0] = *flag*[1] = false;
    *turn* = 0; /* or 1 */
  * 동작
    · *turn*이 $i$이거나 *flag*[1]이 false이면 $P_0$ 가 임계구역에 진입
    · *turn*이 $j$이거나 *flag*[0]이 false이면 $P_1$ 가 임계구역에 진입

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 12**

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j) ;
            critical section
        flag[i] = false;
            remainder section
} while(1);
```

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 13**

* 설명
  - 상호배제: 상호진입은 *turn*으로 하나만 허용되며 허용된 프로세스가 자신의 *flag* 를 false 로 되기전에는 다른 프로세스 진입이 불가
  - 진행
    - ➥ 알고리즘1 문제 해결: 둘 중에 하나가 임계구역을 벗어나면 진입준비된 다른 하나가 바로 진입가능 (*flag* 에 의해서)
    - ➥ 알고리즘2 문제 해결: 두 *flag*가 모두 true 로 되더라도 turn 에 의하여 결정함 (즉, 무한정 기다리지 않음)
  - 한계대기
    - ➥ 다른 프로세스가 진입시도를 하지 않으면 바로 진입가능
    - ➥ 동시에 진입시도 시에는 *turn*에 의하여 하나의 프로세스가 진입가능
    - ➥ 진입한 프로세스가 *flag*를 해제 하자마자 다른 프로세스가 바로 진입가능
    - ➥ 그러므로, 최대 하나의 프로세스 대기후 진입 가능함

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 14

- 다중 프로세스에서 임계구역 문제
  - 해결방법
    * 임계구역의 진입구역에서 번호를 배정받음
      (먼저 도착한 것에 낮은 번호를 부여)
    * 임계구역 진입을 번호 순으로 허용 (낮은 번호 먼저)
    * 단, 동일한 번호가 있으면 안됨
      (이것을 막으려면 번호배정하는 영역이 임계구역이 됨)
    * 그러므로, 동일한 번호를 허용 (진입구역은 임계구역이 아님)
    * 대신 동일한 번호를 부여 받은 경우 프로세스 번호로 우선순위를 결정함
      (프로세스 번호는 시스템내에서 유일하게 주어짐)

## 임계구역 (Critical Section) 문제 (Cont'd)

Slide 15

- 알고리즘
  * 공유변수
    boolean *choosing*[n];
    int *number*[n];
  * 초기값
    *choosing*[n] = false;
    *number*[n] = 0;
  * 조건
    · $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$
    · $max(a_0, \ldots, a_{n-1})$ is a number, $k$, such that $k \geq a_i$ for $i = 0, \ldots, n-1$
  * 동작
    · *number*[i] 가 0이 아니고
    · (*number*[i],i) < (*number*[j],j)이면 $P_i$가 임계구역에 진입
    · 단, $i \neq j$이고 $0 \leq j \leq n-1$

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 16**

```
do {
    choosing[i] = true;
    number[i] = max(number[0],...,number[n-1])+1;
    choosing[i] = false;
    for(j=0; j < n; j++) {
        while(choosing[j]);
        while((number[j]!=0)&&((number[j],j)<(number[i],i)));
    }
        critical section
    number[i] = 0;
        remainder section
} while(1);
```

## 임계구역 (Critical Section) 문제 (Cont'd)

**Slide 17**

* 설명
  - 상호배제
    - ➠ $P_i$ 가 임계구역에 있을 때 *number*[k]!=0 인 모든 $P_k(k \neq i)$는
    - ➠ (*number*[i], i) < (*number*[k], k) 임으로 임계구역 진입 못함
  - 진행
    - ➠ $P_i$ 가 출구구역에 있으면 *number*[i] 가 0이 되고 어떤 $P_j$ 가 바로 실행됨
    - ➠ 단, $j \neq i$ 이고 $k \neq i, j$ 이며 (*number*[j],j) < (*number*[k],k) 일 때
  - 한계대기
    - ➠ $P_i$ 가 진입하고자 한 경우 최대 $n-1$ 프로세스가 진입 후 진입 가능

## 동기화 하드웨어

- 단일 프로세서 시스템에서의 동기화 해결방법
  - 임계구역에서 인터럽트의 발생을 막음

Slide 18

```
do {
        disable interrupt
            critical section
        enable interrupt
            remainder section
} while(1);
```

  - 다중 프로그래밍에 영향을 줌으로 좋은 해결방법이 아님
  - 다중 프로세서에서는 해결방법이 될 수 없음
    (인터럽트를 막아도 두 프로세서에서 두 프로세스가 동시에 실행될 수 있음)
  - 이와같은 이유로 인터럽트를 제어하는 방법보다 특수한 하드웨어 명령을 사용하는 방법을 사용

## 동기화 하드웨어 (Cont'd)

- 동기화를 위한 특수 명령어
  - 이 명령은 원자적(명령어 수행중 인터럽트되지 않음)으로 실행됨
  - 종류
    * 한 워드를 검사하고 수정하는 명령 →TestAndSet 명령
    * 두 워드의 값을 교환하는 명령 →Swap 명령

Slide 19

- TestAndSet 명령을 사용한 동기화
  - 명령어 동작

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

## 동기화 하드웨어 (Cont'd)

**Slide 20**

- 상호배제 방법
  - 공유변수

    boolean *lock*;
  - 초기값

    *lock* = false;

```
do {
        while(TestAndSet(lock));
            critical section
        lock = false;
            remainder section
} while(1);
```

  - 한계대기 만족 못함

## 동기화 하드웨어 (Cont'd)

**Slide 21**

- Swap 명령을 사용한 동기화
  - 명령어 동작

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

## 동기화 하드웨어 (Cont'd)

**Slide 22**

- 상호배제 방법
  - 공유변수: boolean *lock*;
  - 지역변수: boolean *key*;
  - 초기값

    *lock* = false;

    do {

    *key* = true;
    while(*key* == true) Swap(*lock*,*key*);

    critical section

    *lock* = false;

    remainder section

    } while(1);
  - 한계대기 만족 못함

## 동기화 하드웨어 (Cont'd)

**Slide 23**

- TestAndSet을 사용한 상호배제, 진행, 한계대기 만족하는 방법
  - 공유변수

    boolean *waiting*[n];
    boolean *lock*;
  - 지역변수: boolean *key*;
  - 초기값

    *waiting*[n] = false;
    *lock* = false;
  - 동작
    - *lock*이 false 이거나 *waiting*[i]가 false이면 $P_i$ 가 임계구역에 진입

## 동기화 하드웨어 (Cont'd)

Slide 24

```
do {
    waiting[i] = true ;
    key = true;
    while(waiting[i] && key) key = TestAndSet(lock);
    waiting[i] = false;

        critical section

    j = (i+1) % n;
    while((j != i) && !waiting[j]) j = (j+1) % n;
    if(j==i)
      lock = false;
    else
      waiting[j] = false;

        remainder section

} while(1);
```

## 동기화 하드웨어 (Cont'd)

Slide 25

- 설명
  - 상호배제
    - 맨 처음 진입구역에 들어간 프로세스는 *lock*이 false 로 임계구역으로 진입
    - 진입하면서 *lock* 이 true 가 됨으로 나머지 프로세는 *lock*이 true고 *waiting*이 true 로 진입 못함
    - 출구구역에서
      - ➥ *waiting*[j] 가 true 인것 (즉 기다리고 있는것) 을 while 로 찾음
      - ➥ 기다리고 있는 것이 없으면 (if(j==i)경우) *lock*을 false 로 하여 먼저 진입 구역에 진입한 프로세스가 진입하게 함
      - ➥ 기다리고 있는 것이 있으면 (else 경우) 그 프로세스의 *waiting*[j]을 false로 하여 진입구역에 진입
  - 진행문제 해결: *lock*을 false 로 하거나 *waiting*[j]을 false로 하여 임계구역에 진입하게 함
  - 한계대기 문제 해결
    - 대기중인 프로세스가 있으면 현재 진입 프로세스로부터 숫자가 큰 쪽으로 순환하며 찾아서 수행시킴
    - 그러므로 최대 $n-1$ 프로세스 대기 후 서비스 받을 수 있음

## 세마포 (Semaphore)

Slide 26

- 세마포란?
  - 동기화 문제를 해결하기위해 도입한 동기화 도구
  - 정수변수로서 원자적으로 동작하는 두개의 연산이 가능
    * wait 연산

```
wait(S) {
        while(S ≤ 0)
            ; // no-op
        S−−;
}
```

    * signal 연산

```
signal(S) {
        S++;
}
```

## 세마포 (Semaphore) (Cont'd)

Slide 27

- 사용법
  - 세마포를 이용하여 $n$개의 프로세스 임계구역문제 해결
    * 모든 프로세스는 *mutex* 라는 세마포를 공유
    * 초기값은 1
    * 한계대기는 충족하지 못함

```
do {
        wait(mutex);
            critical section
        signal(mutex);
            remainder section
} while(1);
```

## 세마포 (Semaphore) (Cont'd)

Slide 28

- 세마포를 이용한 수행 순서의 결정
  - 두개의 프로세스에서 특정 문장이 순서대로 실행해야할 경우 ($S_1 \rightarrow S_2$)
  - 0으로 초기화되는 *synch* 세마포를 공유
    * $P_1$
      $S_1$;
      signal(*synch*);
    * $P_2$
      wait(*synch*);
      $S_2$;

## 세마포 (Semaphore) (Cont'd)

Slide 29

- 구현
  - 지금까지 구현한 임계구역문제 해결방법은 모두 프로세스들이 루프를 돌면서 기다리는 방법임
    →이를 busy waiting 이라함 (또 다른 말로 spinlock 이라함)
  - spinlock의 장단점
    * 장점: 문맥교환이 필요없음
    * 단점: 스케줄되어 CPU를 사용함으로 비 효율적인 방법임 (단, lock 이 짧을 경우에는 문제되지 않음)
  - spinlock의 단점을 해결하는 방법
    * busy waiting 대신에 프로세스를 중지함 (block 됨)
    * 프로세스 상태는 대기상태로서 세마포 관련 큐에 삽입
      (준비 큐에 없음으로 실행으로 스케줄되지 않음)
    * signal 연산을 통해서 대기중인 프로세스중 하나를 다시 실행 (wakeup 됨)

## 세마포 (Semaphore) (Cont'd)

Slide 30

- busy waiting 이 없는 세마포 구조

```
typedef struct {
        int value;
        struct process *L;
} semaphore;
```

```
void wait(semaphore S) {
        S.value−−;
        if (S.value < 0) {
            add this process to S.L;
            block();
        }
}
```

## 세마포 (Semaphore) (Cont'd)

Slide 31

```
void signal(semaphore S) {
        S.value++;
        if (S.value ≤ 0) {
            remove a process P from S.L;
            wakeup(P);
        }
}
```

* 세마포 큐를 FIFO로 하면 제한대기 보장
* wait 와 signal 연산은 원자적으로 실행되어야함
  단일프로세서: 인터럽트를 막는 방법으로 가능
  다중프로세서: 특수 명령사용 혹은 소프트웨어적으로 해결

## 세마포 (Semaphore) (Cont'd)

Slide 32

- 교착상태 (Deadlock)와 기아 (Starvation)
  - 조건
    - 두 개의 프로세스: $P_0$, $P_1$
    - 1로 설정된 세마포: $S$, $Q$
  - 교착상태

| $P_0$ | $P_1$ |
|---|---|
| $wait(S);$ | $wait(Q);$ |
| $wait(Q);$ | $wait(S);$ |
| . | . |
| . | . |
| . | . |
| $signal(S);$ | $signal(Q);$ |
| $signal(Q);$ | $signal(S);$ |

  - 기아 혹은 무한 블록킹 (indefinite blocking)
    - 프로세스가 세마포에서 무한하게 대기하는 문제
    - 세마포에서 LIFO 방식으로 프로세스를 처리하면 발생할 수 있음