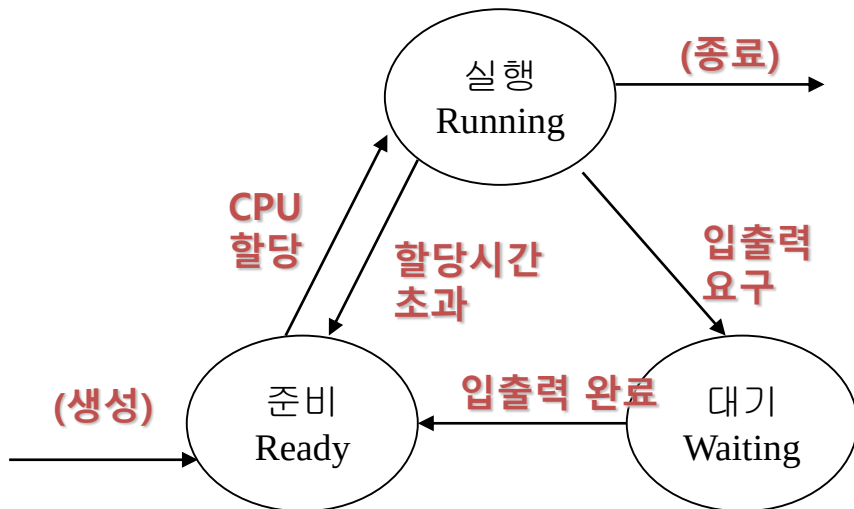




3장. 병행성(Concurrency)

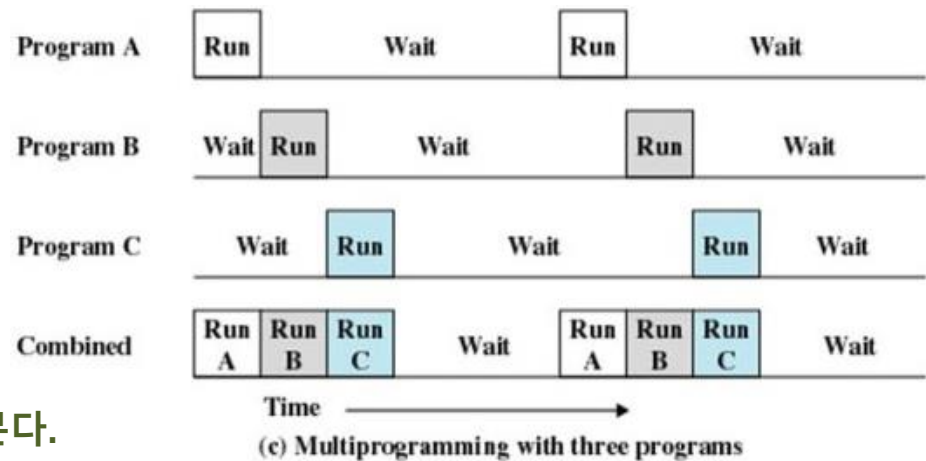
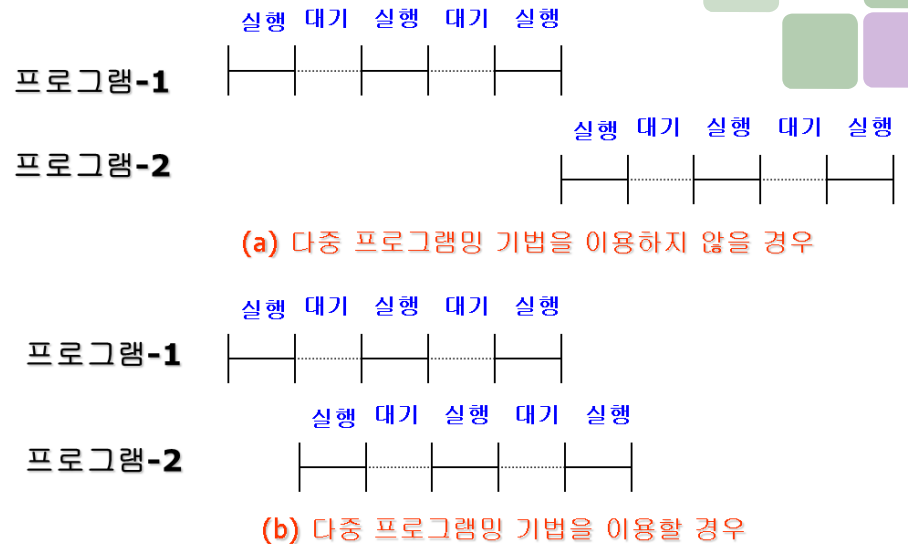
- 병행성 기본 개념
- 임계영역과 임계영역 문제 해결
- 프로세스간 통신 및 동기화 기법

학습목표



단일 CPU 시스템에서 실행(running) 상태의 프로세스는 오직 하나이지만 수행 구조상 다수의 프로세스들이 동시에 실행되는 것처럼 된다.

- ▶ 병행성 기본 개념에 대하여 알아본다.
- ▶ 임계영역과 임계영역 해결 방법에 대하여 알아본다.
- ▶ 프로세스간 통신 및 동기화 기법에 대하여 알아 본다.



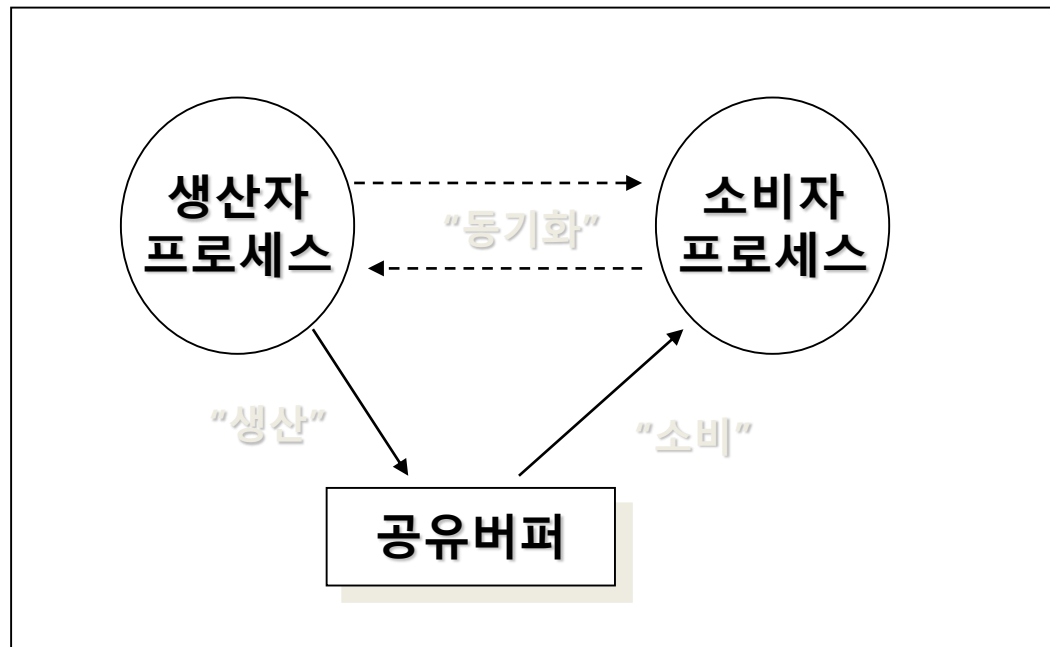
01. 개요

- ❖ 다중 프로그래밍(병행 수행) 환경에서도 모든 프로세스는 바르게 작동되어야(결정성) 함.
 - 프로세스 혹은 스레드 병행성
 - CPU를 사용하여 실행하는 프로세스 혹은 스레드가 한 순간에 어느 하나지만, 동시에 다수가 존재하며 실행되는 것을 ‘프로세스 혹은 스레드 병행성’이라고 부른다.
 - 프로세스의 결정성이란
 - 어떤 환경이든 프로세스는 동일한 입력에 대한 처리 결과가 같아야 하는 성질
- ❖ 다중 프로그래밍 환경에서 다수의 프로세스들이 자원을 공유하면서 병행적으로 실행될 경우 결정성 보장을 위해 특별한 기능이 필요하다.

01. 결정성이 보장되지 않는 예

❖ 환경 : 생산자와 소비자 문제

- 프로세스 수 : 2개
 - 생산자는 버퍼에 생산 (물품 개수 증가시킴)
 - 소비자는 버퍼에서 소비 (물품 개수 감소시킴)
- 두 프로세스간 공유 변수
 - 버퍼 : `buffer[N]` // 물품 보관
 - 변수 : `counter` // 물품 수 기록



* 동기화 : 결정성을 보장하기 위한 제어 기능

- 결정성
 - 생산자가 정확히 생산하고, 소비자는 정확히 소비하고....
 - 공유변수 카운터에는 현재 물품의 개수가 정확히 기록되어 있어야 잘못 생산 하지도 않으며, 잘못 가져 오지도 않는다.

생산자 소비자 문제 (과연 결정성이 보장될까???)

❖ 공유변수

```
char buffer[N];  
int counter = 0;
```

❖ 생산자 프로세스

```
producer() {  
    int in = 0;  
    char * nextp;  
    for(;;) { // 반복  
        * nextp = ... // 자료를 생성  
        while(counter == N) ; // 버퍼가 차면 대기.  
  
        buffer[in] = nextp; // 버퍼에 자료 저장  
        in = (in + 1) % N; // 원형 큐로 버퍼 운영  
        counter = counter + 1; // 물품 개수 증가  
    }  
}
```

❖ 소비자 프로세스

```
consumer() {  
    int out = 0;  
    char * nextc;  
    for(;;) { // 반복  
        while(counter == 0) ; // 버퍼가 비었으면 대기.  
        * nextc = buffer[out]; // 버퍼의 자료 추출  
        out = (out + 1) % N; // 원형 큐로 버퍼 운영  
        counter = counter - 1; // 물품 개수 차감  
        :  
    }  
}
```

두 프로세스에서 공유 변수에
동시에 접근 가능하므로 결정
성은 보장할 수 없다.

생산자 소비자 문제 (과연 결정성이 보장될까???)

❖ high level 언어 1문장은 다수의 machine language로 변경

counter = counter + 1; ➔ P1: mov reg, counter //생산자 프로세스

P2: add reg, 1

P3: mov counter, reg

counter = counter - 1; ➔ C1: mov reg, counter //소비자 프로세스

C2: sub reg, 1

C3: mov counter, reg

counter

4

❖ 결정성 보장이 안되는 예

P1: mov reg, counter

P2: add reg, 1

/* 문맥 교환 발생 */

C1: mov reg, counter

C2: sub reg, 1

C3: mov counter, reg

/* 문맥 교환 발생 */

P3: mov counter, reg

reg = 4; counter = 4

reg = 5; counter = 4

reg = 4; counter = 4

reg = 3; counter = 4

reg = 3; counter = 3

reg = 5; counter = 5

프로세스 경쟁조건(race condition) 발생 => 결정성 보장을 위해 OS에 방지 기능이 있어야 함.
➔ 임계 영역 문제 해결

counter의 최종 값은 4
가 되어야 그런데 5
라니.....

02. 임계 영역(Critical Section) 문제 해결

❖ 임계 영역

- 공유자원(변수)을 접근하는 코드 영역을 임계 영역이라고 함.
 - 예) `counter = counter + 1`
 - 해당 영역 코드를 수행할 때 '임계 영역 안에 있다.'고 함.
- 임계 영역 수행은 결정성 보장을 위해 상호 배제적으로 수행되어야 함.

❖ 임계 영역에 대한 해결책

- 프로세스가 '임계영역 실행을 시작'하면, '임계영역 실행을 종료'할 때까지 방해 받지 않고 수행하도록(중단되지 않도록) 하는 보장이 모든 프로세스에 적용되어야 함.
- 임계영역문제 해결 3 조건(mutual exclusion / progress / bounded waiting)
 - 상호배제 : 임계 영역에 프로세스가 있으면 다른 프로세스의 진입은 거부 되어 함.
 - 진행 : 임계영역 안에 프로세스가 없으면, 진입하려는 프로세스는 진입할 수 있어야 함.
 - 제한된 대기 : 진입을 원하는 프로세스에게 무한정 기다리게 해서는 안됨.

❖ 임계 영역 문제 해결 종류

■ 소프트웨어적인 방법

- 두($N=2$) 프로세스 간 임계영역 해결 노력
 - Dekker 알고리즘1 : 상호배제-만족; 진행-불만족; 제한된 대기-불만족
 - Dekker 알고리즘2 : 상호배제-만족; 진행-불만족; 제한된 대기-불만족
 - Peterson 알고리즘 : 상호배제-만족; 진행-만족; 제한된 대기-만족
- 다중($N>2$) 프로세스 간 임계영역 해결 노력
 - 빵집 알고리즘(Bakery Algorithm)

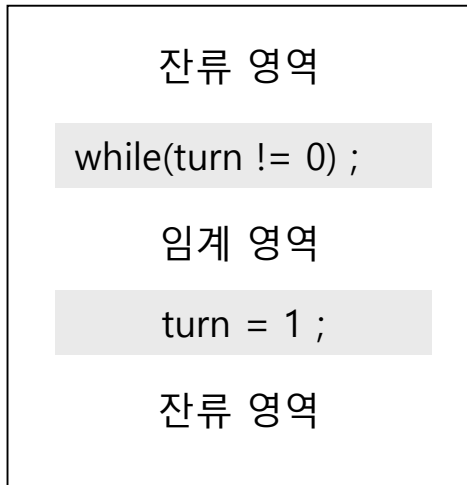
■ 하드웨어적인 방법

- Test-and-Set 명령어
- Swap 명령어

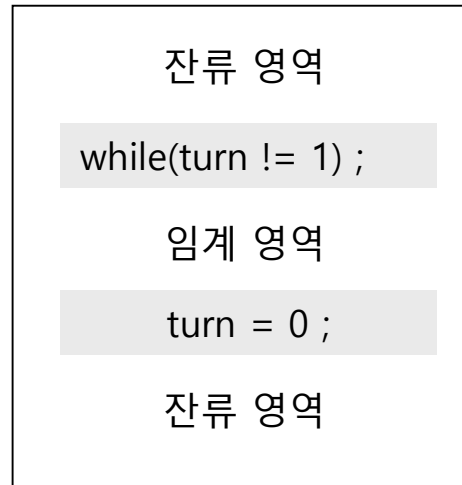
■ 세마포어(semaphore)

❖ 임계 영역 문제 해결

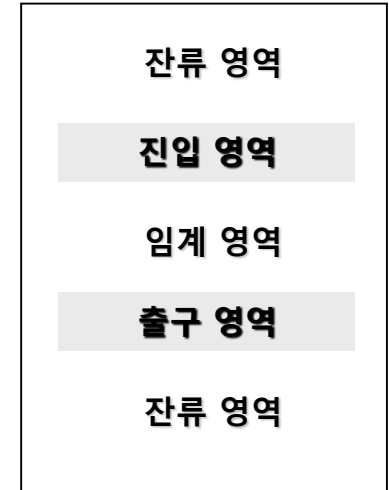
- Dekker 1 알고리즘 : s/w 방법, 두 프로세스 용, 불완전



(a) 프로세스 (P0)



(b) 프로세스 (P1)



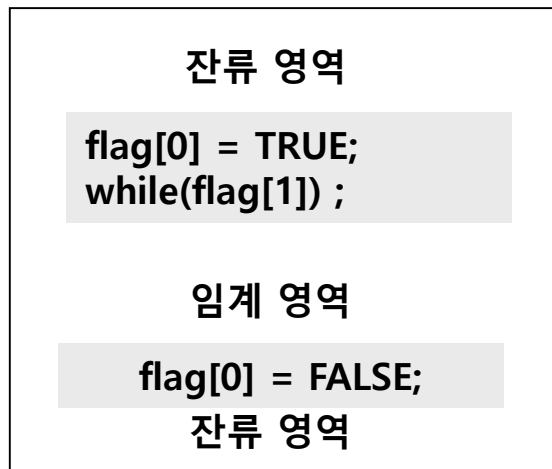
병행성 프로세스의
코드 영역

*. turn은 공유변수

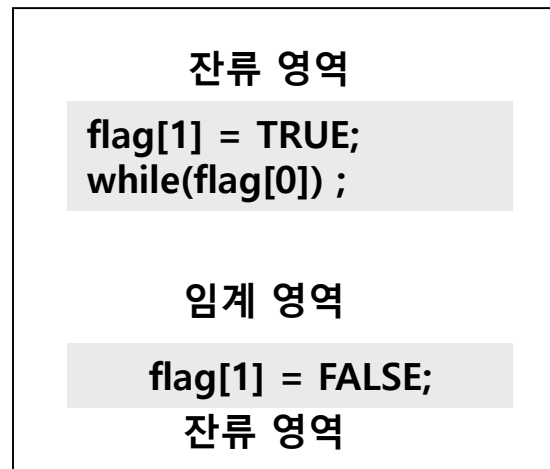
- 상호 배제 : 만족 - 임계 영역을 두 프로세스가 동시에 수행하지 않는다.
- 진행 : 불만족 - 임의의 한 프로세스가 상대가 수행하지 않아 계속 두 번 수행하려 할 때(재진입 문제) 수행할 수 없다.
- 제한된 대기 : 불만족 - P0가 임계 영역을 수행 한 후 turn 값이 0이 될 때까지 무한정 기다려야 한다.

❖ 임계 영역 문제 해결

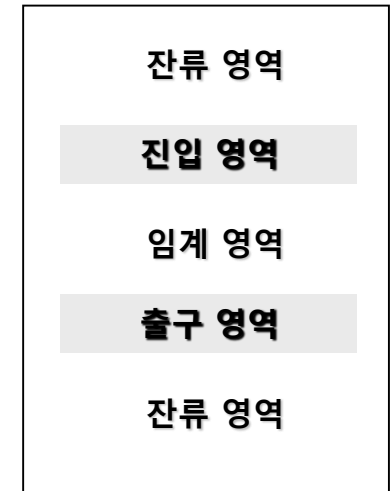
- Dekker 2 알고리즘 : s/w 방법, 두 프로세스 용, 불완전



(a) 프로세스(P0)



(b) 프로세스(P1)



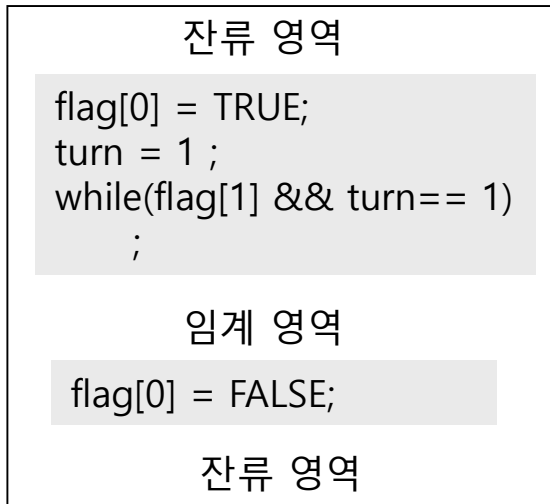
병행성 프로세스의
코드 영역

*. flag[0], flag[1]은 공유변수
초기값은 모두 FALSE로 선언

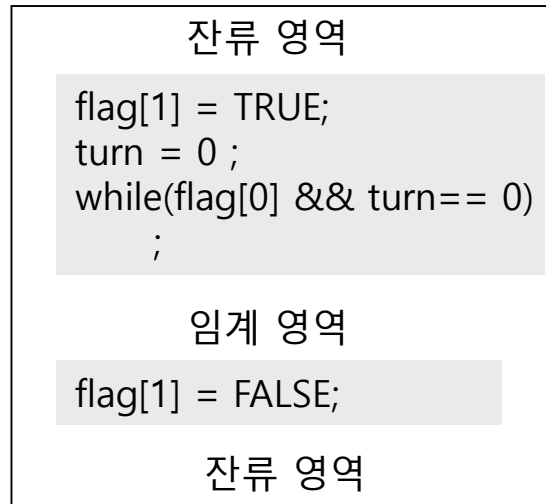
- 상호 배제 : 만족 - 임계 영역을 두 프로세스가 동시에 수행하지 않는다.
- 진행 : 불만족 - 임의의 한 프로세스가 상대가 수행하지 않아 계속 두 번 수행하려 할 때(재진입) 특별한 일이 없으면 수행가능하나, P0 프로세스가 flag[0] 변수를 TRUE로 설정한 후 문맥교환이 발생하면 P1 프로세스가 flag[1] 변수를 TRUE로 설정하더라도 임계영역에 진입하지 못하고, 또 서로 진입하지 못하는 현상이 발생한다.
- 제한된 대기 : 불만족 - 진행의 사례 참조.

❖ 임계 영역 문제 해결

- Peterson 알고리즘 : s/w 방법, 두 프로세스 용, 완전

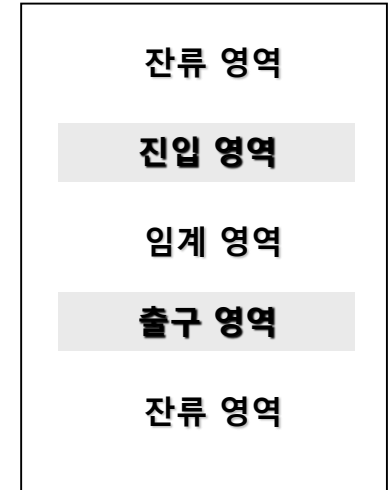


(a) 프로세스(P0)



(b) 프로세스(P1)

- 상호 배제 : 만족. 진행 : 만족 제한된 대기 : 만족



병행성 프로세스의
코드 영역

*. turn, flag[0], flag[1]은 공유변수
flag 변수는 모두 FALSE로 초기화

상황	수행	flag[0]	turn	flag[1]	결론
P1이 잔류영역에 있는 상태에서 P0 임계영역 진입 시도.	P0	TRUE	1	FALSE	P0 임계 영역 진입 가능
P1이 임계영역에 있는 상태에서 P0 임계영역 진입 시도.	P0	TRUE	1	TRUE	P0 진입 영역에 머무름
P0이 잔류영역에 있는 상태에서 P1 임계영역 진입 시도.	P1	FALSE	0	TRUE	P1 임계 영역 진입 가능
P0이 임계영역에 있는 상태에서 P1 임계영역 진입 시도.	P1	TRUE	0	TRUE	P1 진입 영역에 머무름

*. 임의 상태(문맥교환 발생, 재진입 등)에서 상호배제, 진행, 제한된 대기를 점검해 볼 것.

❖ 임계 영역 문제 해결

- 다중($N \geq 2$) 프로세스 간 임계영역 해결(빵집 알고리즘, Bakery Algorithm) : s/w 방법, $N \geq 2$ 프로세스 용, 완전

잔류 영역

```
choosing[i] = TRUE; //Pi가 임계영역 진입 시도
number[i] = max(number[0], number[1], ..., number[n-1]) + 1; //일련번호 부여. 최고 보다 1 높은 값 배정
choosing[i] = FALSE;
for(j=0; j<n; j++){ //모든 프로세스에 대해 조사하여
    while(choosing[j]); //일련번호 부여 시 정지
    while( (number[j] != 0) && (number[j] < number[i]) ); //낮은 번호 먼저
}
```

임계 영역

```
number[i] = 0 ;
```

choosing[n]과 number[n]은 공유변수
초기 값은 FALSE와 0으로 선언.

잔류 영역

- 자신의 choosing을 TRUE로 설정하고 번호를 부여 받은 후 FALSE로 변경. 번호는 기 부여된 번호보다 1큰 번호 배정하도록 되어있음.
- for 문에서는 가장 작은 프로세스를 임계영역에 진입 시킴. 만약 부여 받은 번호가 같은 경우 '(number[j], j) < number[i], i)'에 의해 프로세스 번호가 작은 프로세스가 우선적으로 진입한다.

❖ 임계 영역 문제 해결

- Test-and-Set 명령어 알고리즘 1 : h/w 방법, 불완전 해결

```
bool Test_and_Set(bool * lock) {  
    bool ret;  
    ret = *lock;  
    *lock = TRUE;  
    return ret;  
}
```

Test-and-Set 명령어 – H/W 이므로
수행 중 문맥교환/인터럽트 발생 안됨.

잔류 영역

```
while( Test_and_Set( &flag ) )  
    ;
```

임계영역

```
flag = FALSE;
```

잔류 영역

- 상호 배제, 진행 조건 : 만족 flag는 공유변수로서 초기값은 FALSE
 - 임계영역에 프로세스가 있으면, 다른 프로세스가 진입 할 수 없음.
 - 임계영역을 진입하려는 프로세스가 여러 개 있을 경우 오직 하나의 프로세스만 진입 가능.
 - 임계 영역을 실행하는 프로세스가 없으면 항상 진입 가능.
- 제한된 대기 : 불만족
 - 특정 프로세스가 임계영역을 반복해서 진입할 수 있음.

❖ 임계 영역 문제 해결

■ Test-and-Set 명령어 알고리즘 2 : h/w 방법, 완전 해결

잔류 영역

```
bool Test_and_Set(bool * lock) {  
    bool ret;  
    ret = *lock;  
    *lock = TRUE;  
    return ret;  
}
```

Test-and-Set 명령어 – H/W 이므로
수행 중 문맥교환/인터럽트 발생 안됨.
flag 변수 접근 및 변환은
Test_and_Set 에게만 허용

```
waiting[i]=TRUE;  
key=TRUE; //flag 상태를 읽어 오기 위한 변수  
while ( waiting[i] && key ) key=Test_and_Set( &flag );  
waiting[i] = FALSE;
```

임계 영역

```
j = (i+1) % N;  
while( (j != i) && ( waiting[j] == FALSE ) )  
    j = (j+1)%N; //진입 의사 표시 프로세스 찾기...  
if (j == i) flag = FALSE; //대기자가 없어 자기 진입 허용  
else waiting[j] = FALSE; //대기자가 진입할 수 있게 함
```

잔류 영역

waiting[N]과 flag는 공유변수로서
초기값은 FALSE

■ 상호 배제, 진행 조건, 제한된 대기 : 모두 만족

- 임계영역에 프로세스가 있으면, 다른 프로세스가 진입 할 수 없음.
- 임계영역을 진입하려는 프로세스가 여러 개 있을 경우 오직 하나의 프로세스만 진입 가능.
- 임계 영역을 실행하는 프로세스가 없으면 항상 진입 가능.
- 제한된 대기 조건 만족.

❖ 임계 영역 문제 해결

■ Test-and-Set 명령어 알고리즘 2

- h/w 방법, 완전 해결

```
bool Test_and_Set(bool * lock) {  
    bool ret;  
    ret = *lock;  
    *lock = TRUE;  
    return ret;  
}
```

Test-and-Set 명령어 – 기계어 이므로
수행 중 문맥교환/인터럽트 발생 않됨.

잔류 영역

```
waiting[i]=TRUE;  
key=TRUE;  
while ( waiting[i] && key ) key=Test_and_Set( &flag );  
waiting[i] = FALSE;
```

임계 영역

```
j = (i+1) % N;  
while( (j != i) && ( waiting[j] == FALSE ) )  
    j = (j+1)%N;  
if (j == i) flag = FALSE;  
else waiting[j] = FALSE;
```

Process 4:

잔류 영역

waiting[N]과 flag는 공유변수로서
초기값은 FALSE.
N은 참여 프로세스 수

잔류 영역

Process 2:

```
waiting[i]=TRUE;  
key=TRUE;  
while ( waiting[i] && key ) key=Test_and_Set( &flag );  
waiting[i] = FALSE;
```

임계 영역

```
j = (i+1) % N;  
while( (j != i) && ( waiting[j] == FALSE ) )  
    j = (j+1)%N;  
if (j == i) flag = FALSE;  
else waiting[j] = FALSE;
```

잔류 영역

❖ 임계 영역 문제 해결

- Swap 명령어 방법 : HW 방법. 보완에 의한 완전 해결

```
swap(bool *a, bool *b) {  
    bool temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

잔류 영역

```
key = TRUE;  
while(key) Swap(&flag, &key);
```

임계 영역

```
flag = FALSE;
```

잔류 영역

(Process01)

잔류 영역

```
key = TRUE;  
while(key) Swap(&flag, &key);
```

임계 영역

```
flag = FALSE;
```

잔류 영역

(Process02)

flag는 공유변수. 초기값은 FALSE

- 상호 배제, 진행 : 만족 (제한된 대기는 앞의 Test-and-Set 명령어 2와 같이 해결)
 - 임계영역에 프로세스가 있으면, 다른 프로세스가 진입 할 수 없음.
 - 임계영역을 진입하려는 프로세스가 여러 개 있을 경우 오직 하나의 프로세스만 진입 가능.
 - 임계 영역을 실행하는 프로세스가 없으면 항상 진입 가능.
 - 특정 프로세스가 임계영역을 반복해서 진입할 수 있음. (앞의 Test-and-Set 명령어 2와 같이 해결함)

❖ 임계 영역 문제 해결

- Swap 명령어 방법 : HW, 보완에 의한 완전 해결

```
swap(bool *a, bool *b)
{
    bool temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

잔류 영역

```
waiting[i] = TRUE;
key = TRUE;
while( waiting[i] && key ) key = Swap (&flag, &key ); // Swap (&flag, &key );
waiting[i] = FALSE;
```

임계 영역

```
j = (i+1) % N;
while(( j != i )&&( waiting[j] == FALSE) )
    j = (j+1) % N;
if (j == i)
    flag = FALSE; //대기자가 없으니 자신의 재 진입 허용
else
    waiting[j] = FALSE; //대기자를 진입하게 함.
```

잔류 영역

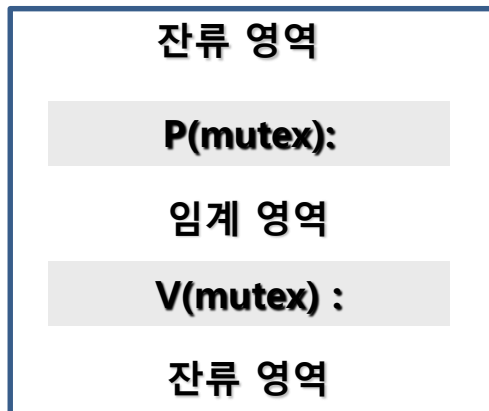
waiting[N]과 flag는 공유변수. 초기값은 FALSE

❖ 임계 영역 문제 해결

- 세마포어(semaphore)에 의한 방법 : 특별한 P, V연산자에 의존한 방법
 - 세마포어 변수
 - 초기값이 부여된 변수로 P, V 연산자에 의해서만 변경할 수 있는 정수형 변수.
 - P 연산자 : Proberen(검사하다) => wait(기다리다)
P(S): while($S \leq 0$) ; //여기에서 인터럽트 걸릴 수 있다.
 $S := S - 1$;
 - V 연산자 : Verhogen(증가하다) => signal(신호를 전하다)
V(S): $S := S + 1$;

❖ 임계 영역 문제 해결

- 세마포어(semaphore)에 의한 방법
 - 세마포어에 의한 상호 배제 알고리즘-1
 - 상호배제와 진행 만족, 제한된 대기 불만족



*. 임계 영역에 1개의 진입만 허용.
세마포어 변수로 mutex를 사용.
임계 영역 진입을 1개만 허용하므로
mutex 초기 값을 1로 설정.

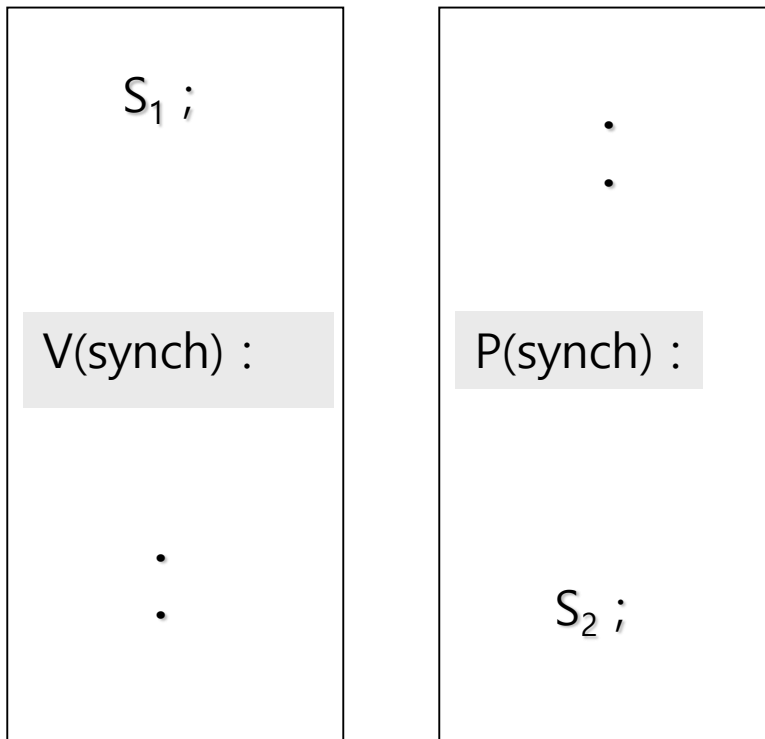
P(S): while($S \leq 0$) ;
 $S = S - 1$;

V(S): $S = S + 1$;

*교재의 ' $:=$ ' 는 치환 연산자의
파스칼식 표현

❖ 동기화 문제 해결

- 세마포어(semaphore)에 의한 방법
 - 세마포어에 의한 동기화 문제 해결 : S1이 S2 보다 먼저 수행해야 하는 경우



P(S): while(S <= 0) ;
 S = S - 1;

V(S): S = S + 1;

*. 동기(순서제어) 문제를 해결하는 것이므로
 synch 변수 이용.
 synch 변수의 초기값은 0.

❖ 임계 영역 문제 해결

- 세마포어(semaphore)에 의한 방법
 - 세마포어에 의한 임계영역 운용 busy waiting / spin lock 문제 해결

```
struct {  
    integer  value; //초기값을 1로 설정  
    struct pcb *ptr; //대기 상태의 큐 포인팅  
} S; //세마포어 변수
```

```
P(S): S.value = S.value-1;  
    if(S.value < 0) {  
        이 프로세스를 S.ptr에 등록;  
        block();  
    }
```

```
V(S): S.value = S.value+1;  
    if(S.value <= 0) {  
        S.ptr로부터 프로세스(P)를 제거;  
        wakeup(P);  
    }
```

잔류 영역

P(S):

임계 영역

V(S) :

잔류 영역

*. S.value의 값이 마이너스일 때 Sleep된 프로세스 수를 의미한다.

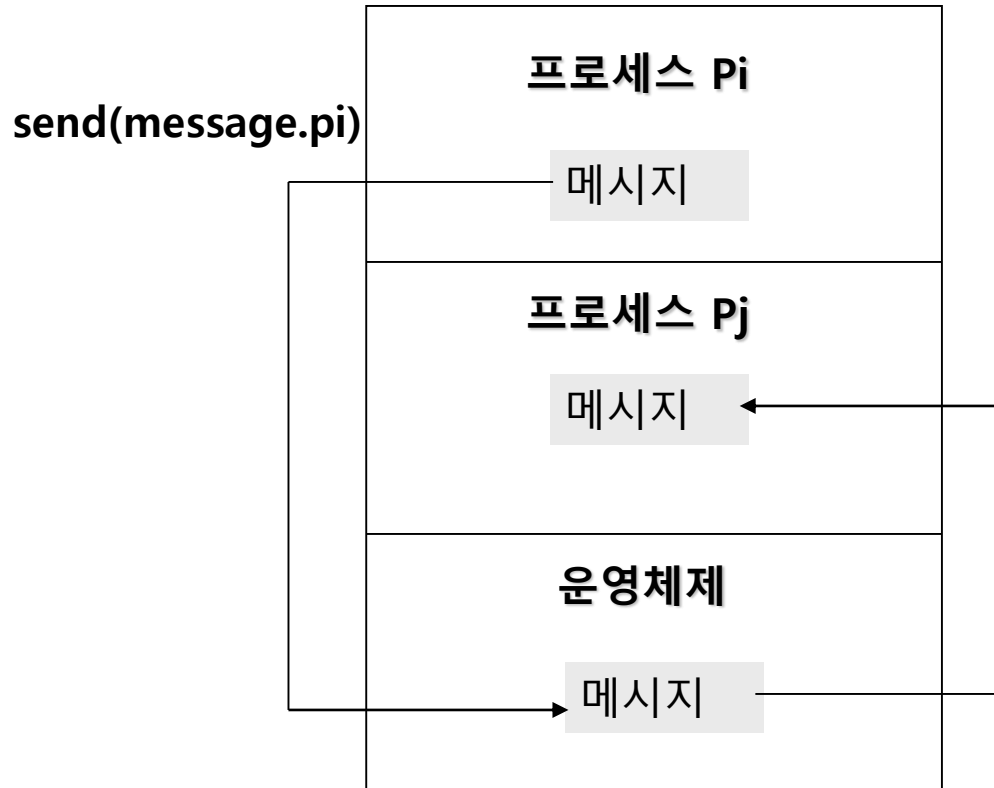


❖ 프로세스 간 통신(IPC : Inter Process Communication)

- 프로세스끼리 자료를 주고 받는 것
- ‘메시지 전송 방식’과 ‘공유 메모리 방식’이 있다.

❖ 프로세스 간 통신(IPC : Inter Process Communication)

- 메시지 전송 방식
 - OS 영역을 공유 메모리로 이용하는 방식



- *. `send()`와 `receive()`는 시스템 호출
- *. 프로세스 P_i 에서 메시지를 작성하여 `send()`를 통해 운영체제 영역에 복사하며, 프로세스 P_j 가 `receive()`를 통해 운영체제 영역에 있는 메시지를 복사해 온다. 메시지 큐는 운영체제에서 유지 관리함.
- *. 메시지가 세 군데 존재하는 단점
- *. P_i 와 P_j 의 동기 문제는 운영체제가 해결해주는 편리함.

receive(message.pi)

❖ 프로세스 간 통신(IPC : Inter Process Communication)

- 공유 메모리 방식
 - 사용자 영역을 공유 메모리로 이용하는 방식



- *. P_i는 OS에게 사용자 공간의 메모리(공유 메모리) 요청
- *. OS는 사용자 공간에 공유 메모리 설정 및 주소 통보
- *. P_i는 공유 메모리에 메시지 작성
- *. P_j는 OS에게 공유 메모리 주소 요청 후 접근
- *. 메시지가 한 군데만 존재하지만 상호 배제와 동기화 문제는 사용자 책임.