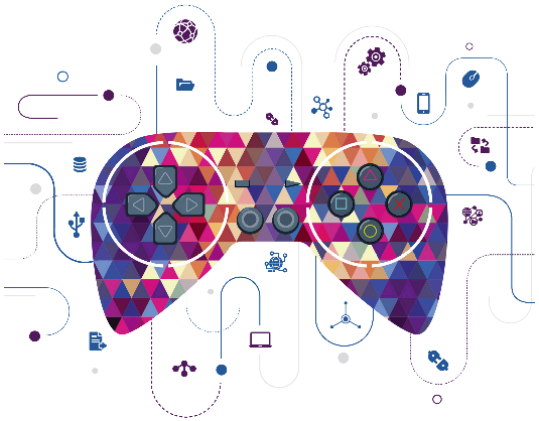


그림으로 배우는 구조와 원리

운영체제 개정 3판

Chatpter 04
병행 프로세스와 상호배제

01 병행 프로세스

02 상호배제와 동기화

03 상호배제 방법들

- 병행성의 원리를 이해
- 병행 프로세스 수행과 관련하여 상호배제를 해결하는 방법

Section 01 병행프로세스(1.병행 프로세스의 개념)

■ 병행 프로세스의 개념

- 운영체제가 프로세서를 빠르게 전환, 프로세서 시간 나눠 마치 프로세스 여러 개를 동시에 실행하는 것처럼 보이게 하는 것

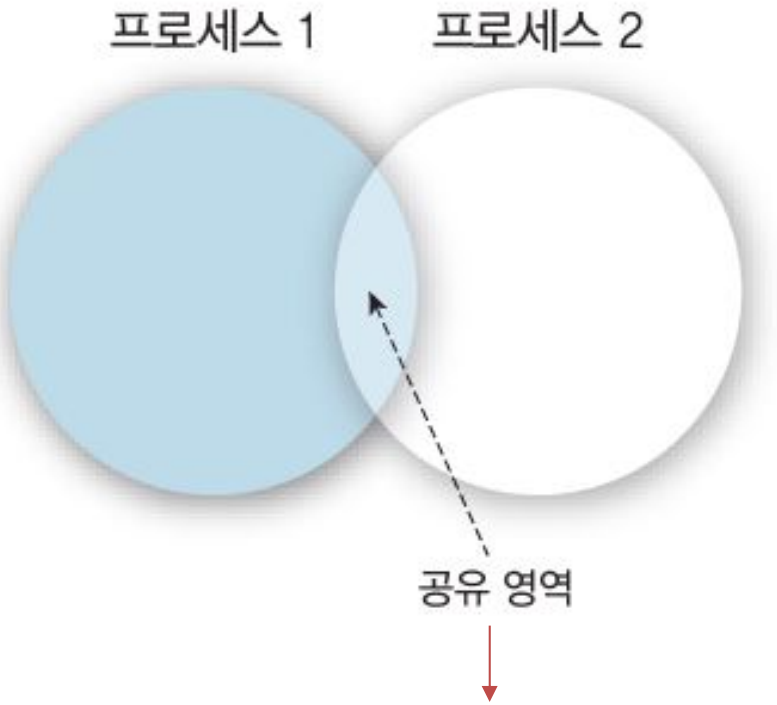


그림 4-1 메모리의 공유 영역

모든 프로세스가 동시에 공유
즉, 이 메모리 자원은 공유 영역에서 병렬parallel로 사용

1.병행 프로세스의 개념

병행 프로세스 종류

독립 프로세스

- 단일 처리 시스템에서 수행하는 병행 프로세스, 다른 프로세스에 영향 주고받지 않으면서 독립 실행
- 다른 프로세스, 데이터와 상태 공유 않고 동작도 재현 가능
- 주어진 초기값에 따라 항상 동일한 결과
- 중지 후 변동 없이 다시 시작 가능
- 독립 실행할 수 있는 프로세스
  - 단일 프로그래밍 : 프로세서를 사용 중이던 프로세스 완료 후 다른 프로세스 실행
  - 다중 프로그래밍 : 프로세스 여러 개가 프로세서 하나 공유. 공유하지 않는 상태일 때 디스패치 순서 상관 없음
  - 다중 처리 : 프로세서 2개 이상 사용하여 동시에 프로그램 여러 개를 병렬 실행

    프로세스는 한 번에 프로세서 하나에서 실행하지만, 동일한 시스템에서는 서로 다른 시간에 서로 다른 프로세서에서 실행 가능

1.병행 프로세스의 개념

- 협력 프로세스
  - 다른 프로세스와 상호작용하며 특정 기능 수행하는 비동기적 프로세스
  - 제한된 컴퓨터 자원의 효율성 증대, 계산 속도 향상, 모듈적 구성 강화, 개별 사용자의 여러 작업 동시에 수행 편의성 제공에 사용
  - 간단한 예 : 두 프로세스의 동일한 파일 사용 시
  - 프로세스 하나가 파일에서 읽기 수행 동안 다른 프로세스가 해당 파일에 쓰기하면 서로 영향
  - 병행 프로세스들이 입출력장치, 메모리, 프로세서, 클록 등 자원을 서로 사용 시 충돌 발생
  - 충돌을 피하기 위한 프로세스의 상호작용 형태

❶ 프로세스는 서로 인식하지 못하는 경쟁 관계 유지. 다중 프로그래밍 환경이 대표적인 예로, 운영체제가 자원 경쟁 고려하여 동일한 디스크나 프린터로 접근 조절

❷ 프로세스는 입출력 버스를 비롯한 개체를 공유하는 단계에서 간접적으로 서로 관계 인식. 이때 다른 프로세스에서 얻은 정보에 의존, 프로세스의 타이밍에 영향. 프로세스들은 개체 공유에 따른 협력 필요

❸ 프로세스에는 서로 인식하고 프로세스끼리 통신할 수 있는 기본 함수 있음. 프로세스가 서로 협력 관계에 있으면 직접 통신 가능, 병행해서 함께 동작 가능

2.병행 프로세스의 해결 과제

병행성

- 여러 프로세스를 이용하여 작업을 수행하는 것
- 시스템 신뢰도 높이고 처리 속도 개선, 처리 능력 높이는 데 중요

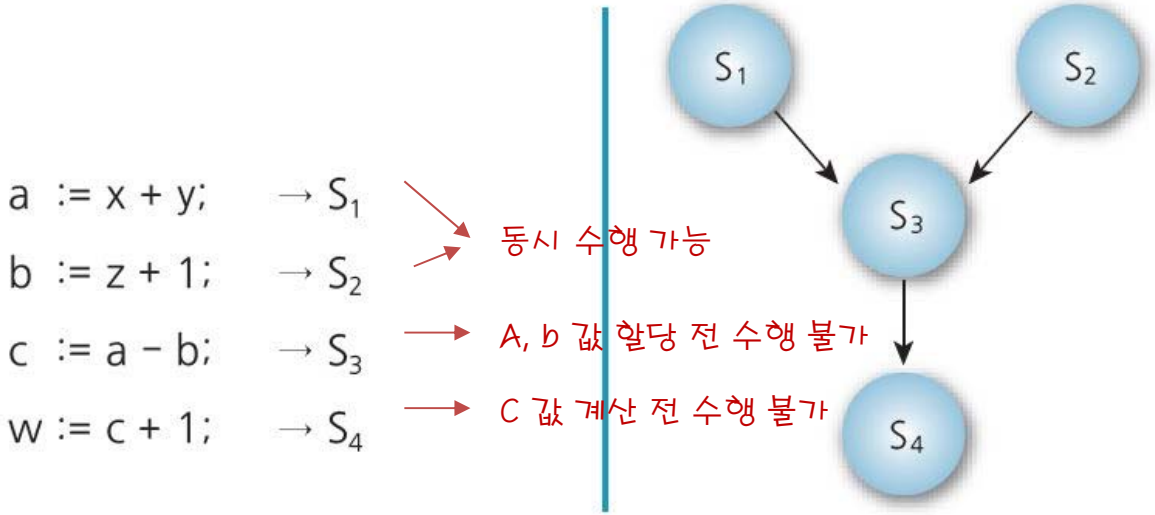
병행 프로세스의 문제

- 공유 자원 상호 배타적 사용(프린터, 통신망 등은 한 순간에 프로세스 하나만 사용)
- 병행 프로세스 간의 협력이나 동기화(상호배제도 동기화의 한 형태)
- 두 프로세스 간 데이터 교환을 위한 통신
- 동시에 수행하는 다른 프로세스의 실행 속도와 관계 없이 항상 일정한 실행 결과 보장 (결정성determinacy)확보
- 교착 상태 해결, 병행 프로세스들의 병렬 처리 능력 극대화
- 실행 검증 문제 해결
- 병행 프로세스 수행 과정에서 발생하는 상호배제 보장

3.선형 그래프와 병행 프로그램

■ 선행 그래프precedence graph

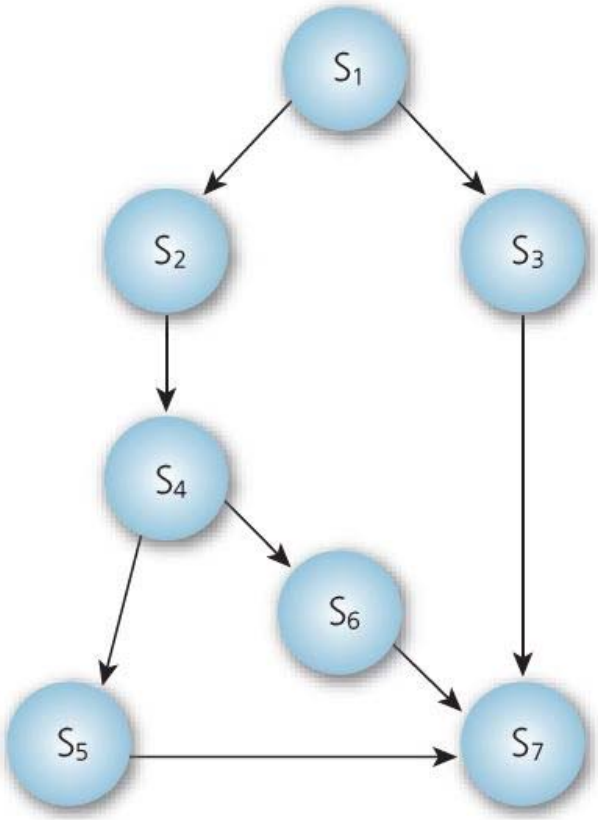
- 선행 제약의 논리적 표현
- 프로세스 : 프로세스 집합과 이것의 선행 제약precedence constraint 두 가지 요소로 정의
- 선행 제약 : 프로세스를 순서대로 다른 상태로 옮기는 것
  - 두 프로세스에 선행 제약이 없으면 이 둘은 독립적이므로 병행 실행 가능
- 순차적 활동을 표현하는 방향성 비순환 그래프
- 선행 그래프에서 노드는 소프트웨어 작업이거나 동시에 실행할 수 있는 프로그램 명령



(a) 알고리즘 (b) 선행 그래프

그림 4-2 간단한 산술 연산의 알고리즘과 선행 그래프

3.선형 그래프와 병행 프로그램



(a) 선행 그래프

- S_2와 S_3은 S_1이 끝난 후에 수행한다.
- S_4는 S_2가 끝난 후에 수행한다.
- S_5와 S_6은 S_4가 끝난 후에 수행한다.
- S_7은 S_5, S_6, S_3이 끝난 후에 수행하고, S_3은 S_2, S_4, S_5, S_6과 병행하여 수행할 수 있다.

(b) 선행 관계

그림 4-3 비순환 선행 그래프와 선행 관계 예

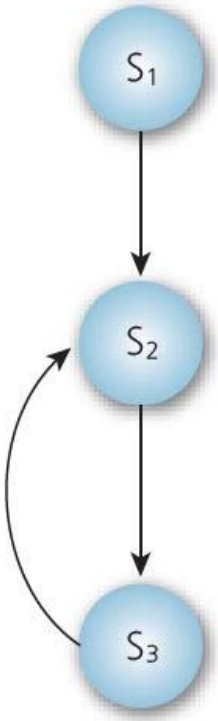


그림 4-4 순환 선행 그래프

3.선형 그래프와 병행 프로그램

fork와 join 구조

- 선행 그래프는 연산의 선행 제약 정의에 유용하지만, 2차원이라 프로그램에는 사용 곤란
- 선행 관계 명시 위해 fork와 join 구조, 병행 문장(parbegin/parend) 등 다른 방법 필요
- 콘웨이Conway(1963년)와 데니스Dennis (1966년), 혼Van Horn(1966년)이 소개
  fork와 join 두 명령어 사용 최초로 병행을 언어적으로 표현

fork 명령어

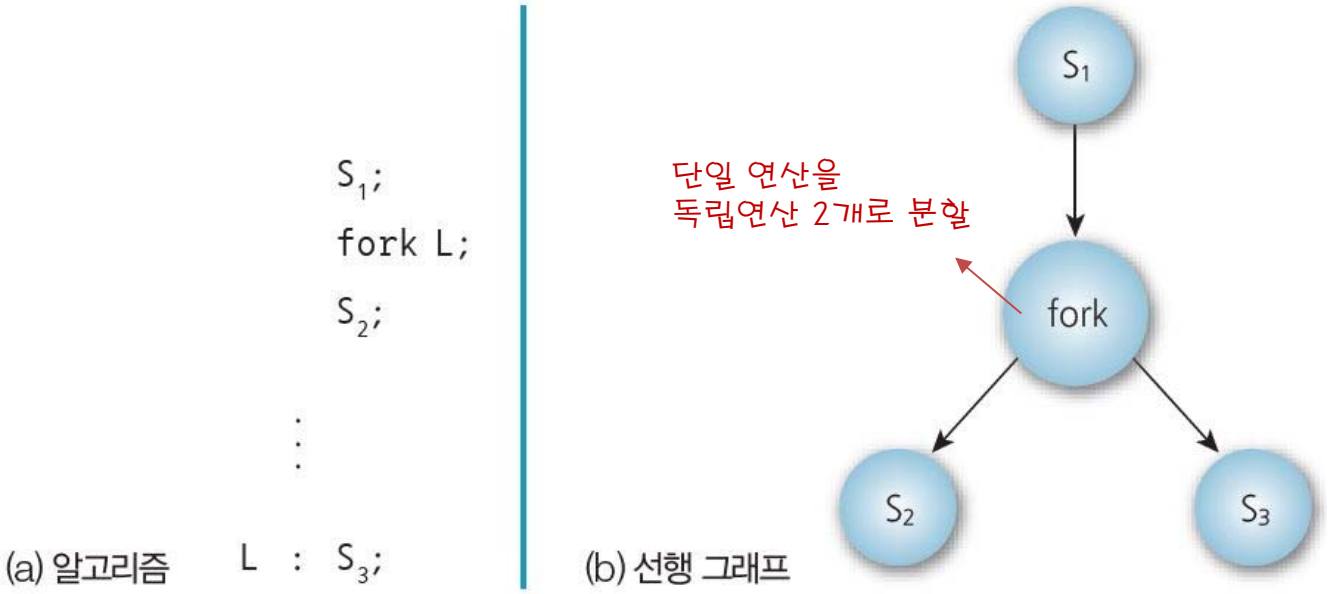


그림 4-5 fork 구조의 알고리즘과 선행 그래프

3.선형 그래프와 병행 프로그램

join 명령어

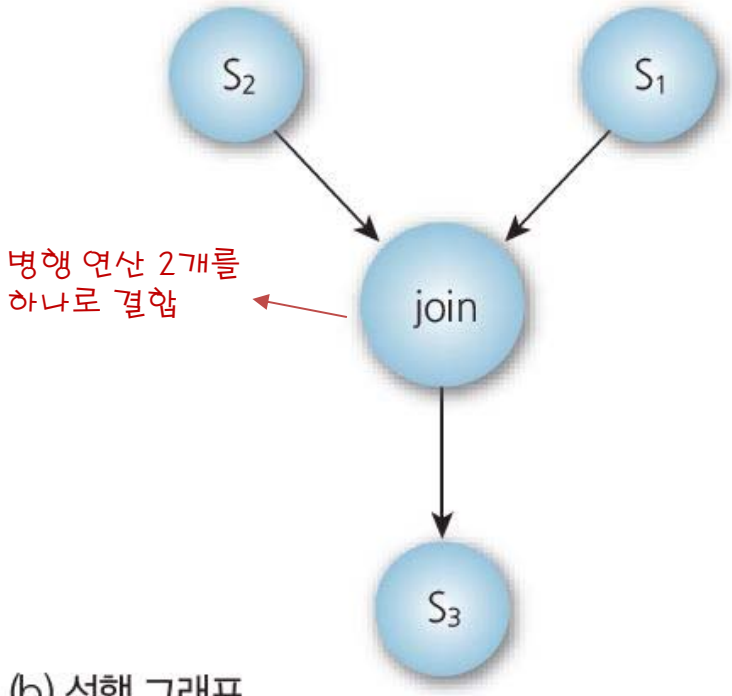
```
...
count := count - 1;
if count = 0 then quit;
```

0 이 아닌 정수값

종료 명령어

```
        count := 2;
        fork L1;
        .
        .
        .
        S1;
        goto L2;
L1 :    S2;
L2 :    join count;
        S3;
```

(a) 알고리즘



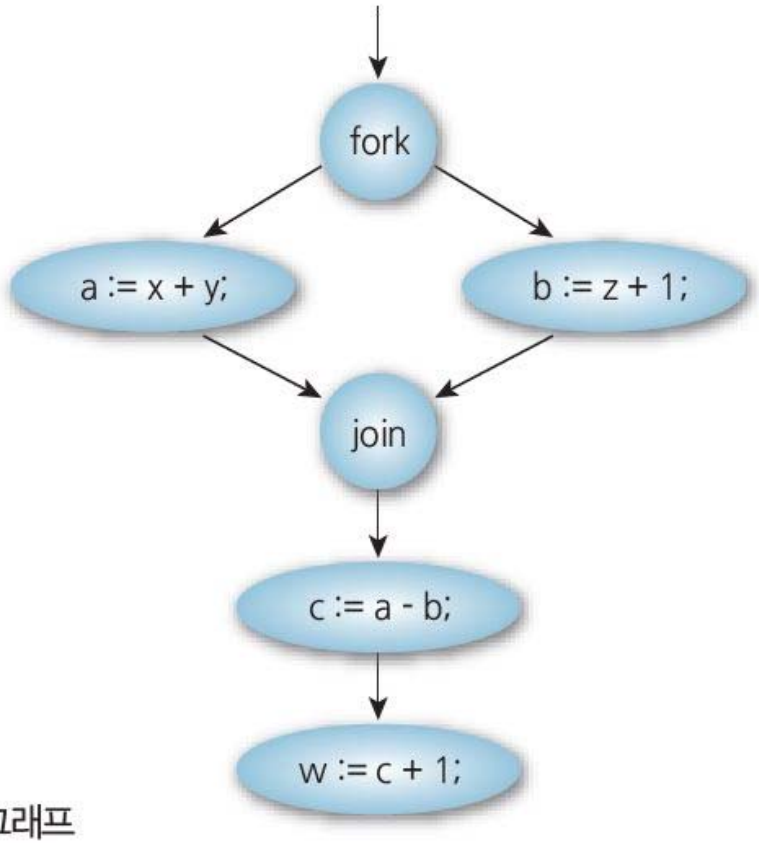
(b) 선행 그래프

그림 4-6 join 구조의 알고리즘과 선행 그래프

3.선형 그래프와 병행 프로그램

```
        count := 2;
        fork L1;
        a := x + y;
        goto L2;
        b := z + 1;
L1 :    join count;
L2 :    c := a - b;
        w := c + 1;
```

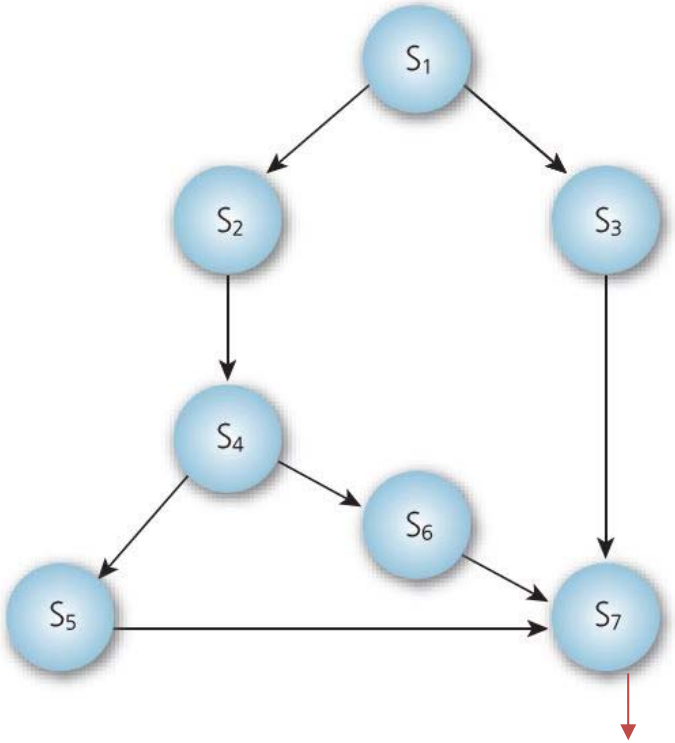
(a) 알고리즘



(b) 선행 그래프

그림 4-7 산술 연산에서 fork와 join 구조의 알고리즘과 선행 그래프

3.선형 그래프와 병행 프로그램



(a) 선행 그래프

유일한 join 노드

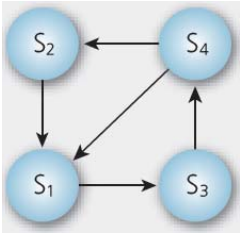
```
      S1;
      count := 3;
      fork L1;
      S2;
      S4;
      fork L2;
      S5;
      goto L3;
L2 :  S6;
L1 :  goto L3;
      S3;
L3 :  join count;
      S7;
```

(b) 알고리즘

그림 4-8 그림 4-3의 선행 그래프, fork와 join 구조 알고리즘

유입 정도와 유출 정도

유입 정도in-degree는 방향 그래프에서 들어오는 간선 수이고, 유출 정도out-degree는 방향 그래프에서 나가는 간선 , 예를 들어, 다음 그래프에서 S1의 유입 정도 는 2, 유출 정도는 1이다.



3.선형 그래프와 병행 프로그램

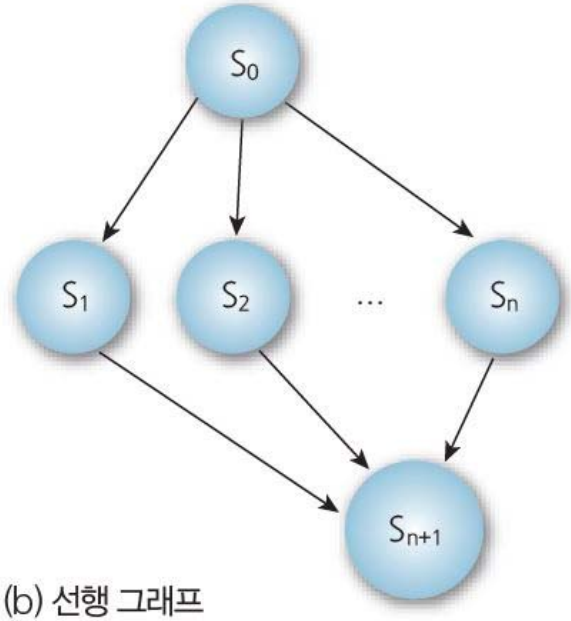
■ 병행 문장

- 하나의 프로세스가 여러 병렬 프로세스로 퍼졌다가 다시 하나로 뭉쳐지는 것을 나타냄
- 대표적인 예 : 다익스트라(1965년)가 제안한 parbegin/ parend
- 일반적인 형태

```
parbegin S1; S2; . . . . . ; Sn; parend;
```

S_0; parbegin S_1; S_2; . . . . . ; S_n; parend; S_{n+1};

(a) 일반 구조의 병행 문장



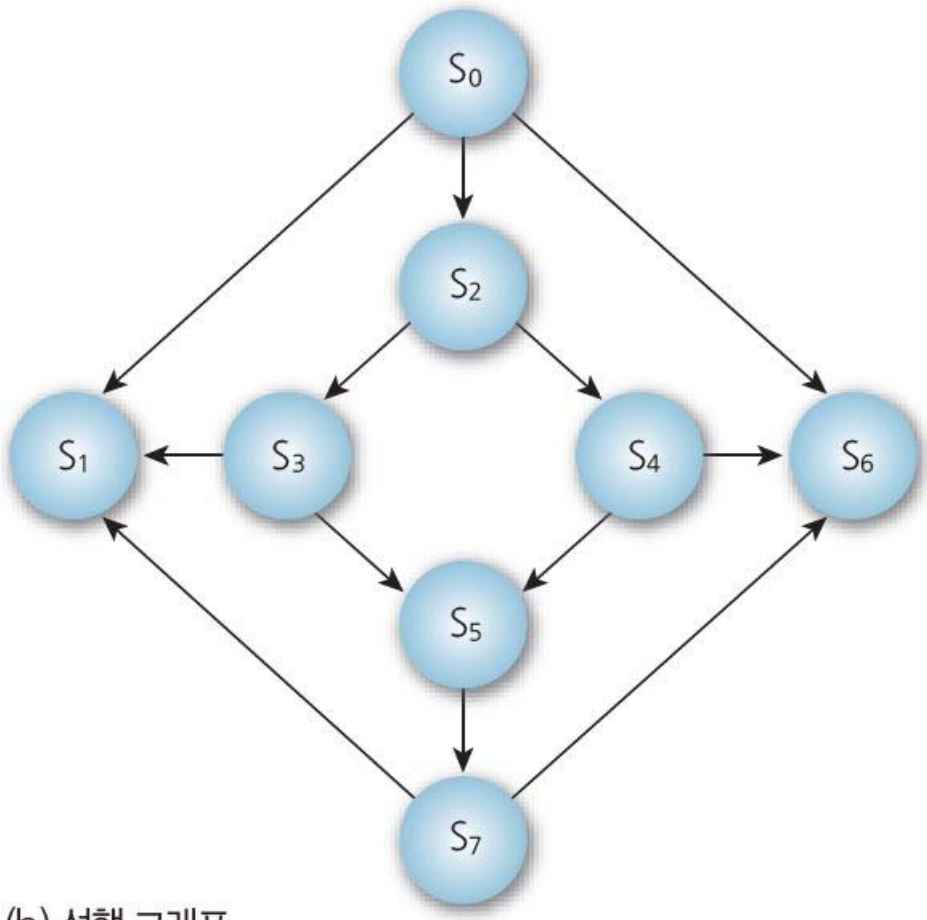
(b) 선행 그래프

그림 4-9 일반 구조의 병행 문장과 선행 그래프

3.선형 그래프와 병행 프로그램

```
S0;
PARBEGIN
    S1;
    BEGIN
        S2;
        PARBEGIN
            S3;
            S4;
        PAREND;
        S5;
    END;
    S6;
PAREND;
S7;
```

(a) 알고리즘



(b) 선행 그래프

그림 4-10 복잡한 구조의 병행 문장과 선행 그래프 모든 문장을 S1; S2; ……; Sn;과 같이 실행한 후 Sn + 1 결과

3.선형 그래프와 병행 프로그램

```
a := x + y;
b := z + 1;
c := a - b;
w := c + 1;
```

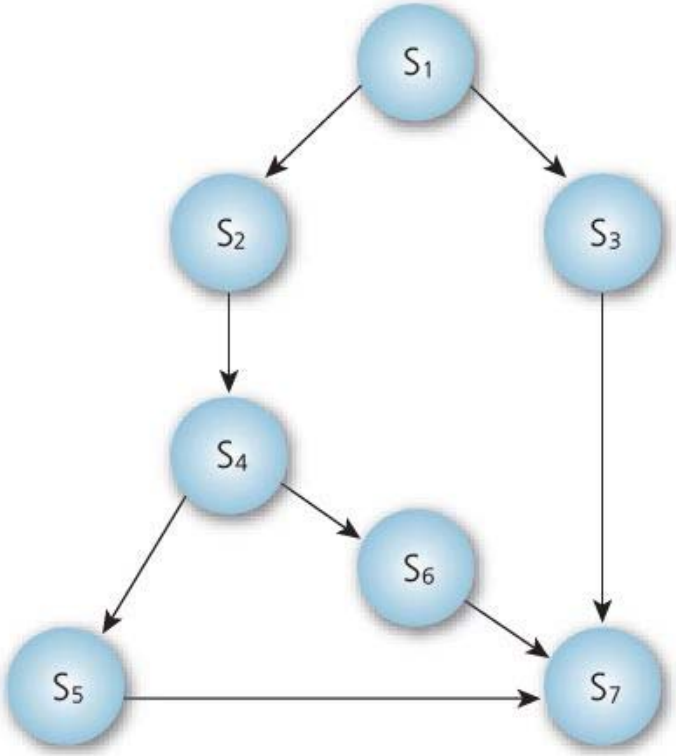
(a) 간단한 산술 알고리즘

```
parbegin
    a := x + y;
    b := z + 1;
parend;
    c := a - b;
    w := c + 1;
```

(b) parbegin/parend 구조 알고리즘

그림 4–11 그림 4–2 간단한 산술 연산의 parbegin/parend 구조

3.선형 그래프와 병행 프로그램



(a) 선행 그래프

```
S1;
parbegin
    S3;
    begin
        S2;
        S4;
        parbegin
        S5;
        S6;
        parend;
    end;
parend;
S7;
```

(b) 알고리즘

그림 4-12 그림 4-3 선행 그래프의 parbegin/parend 구조 알고리즘

Section 02 상호배제와 동기화(1.상호배제의 개념)

상호배제mutual exclusion의 개념

- 병행 프로세스에서 프로세스 하나가 공유 자원 사용 시 다른 프로세스들이 동일한 일을 할 수 없도록 하는 방법

- 읽기 연산은 공유 데이터에 동시에 접근해도 문제 발생 않음

- 동기화 : 변수나 파일은 프로세스별로 하나씩 차례로 읽거나 쓰도록 해야 하는데, 공유 자원을 동시에 사용하지 못하게 실행을 제어하는 방법 뜻 함
  - 동기화는 순차적으로 재사용 가능한 자원을 공유하려고 상호작용하는 프로세스 사이에서 나타남
  - 동기화로 상호배제 보장할 수 있지만, 이 과정에서 교착 상태와 기아 상태가 발생할 수 있음

1. 상호배제의 개념

■ 상호배제의 구체적인 예

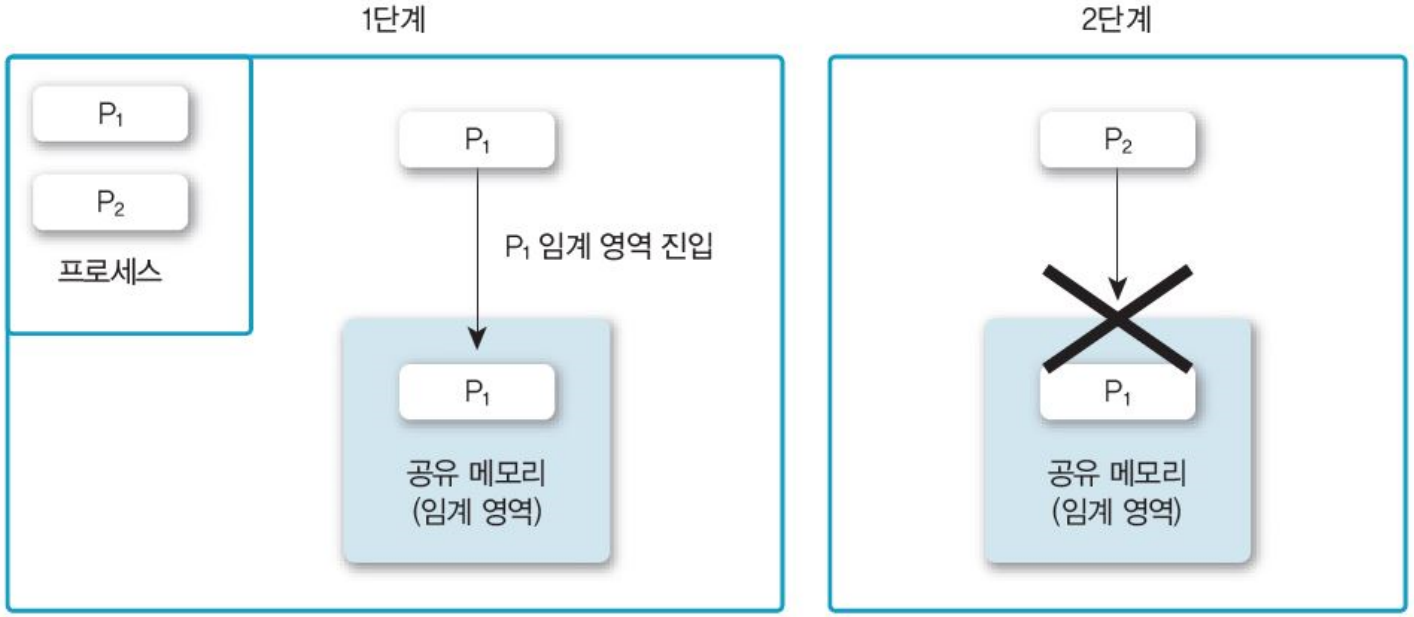


그림 4-13 상호배제의 개념

임계자원critical resource : 두 프로세스가 동시에 사용할 수 없는 공유 자원
임계영역critical section : 임계 자원에 접근하고 실행하는 프로그램 코드 부분

■ 상호배제의 조건

❶ 두 프로세스는 동시에 공유 자원에 진입 불가

❷ 프로세스의 속도나 프로세서 수에 영향 받지 않음

❸ 공유 자원을 사용하는 프로세스만 다른 프로세스 차단 가능

❹ 프로세스가 공유 자원을 사용하려고 너무 오래 기다려서는 안 됨

2. 임계영역

■ 임계 영역의 개념

- 다수의 프로세스 접근 가능하지만, 어느 한 순간에는 프로세스 하나만 사용 가능
- 예

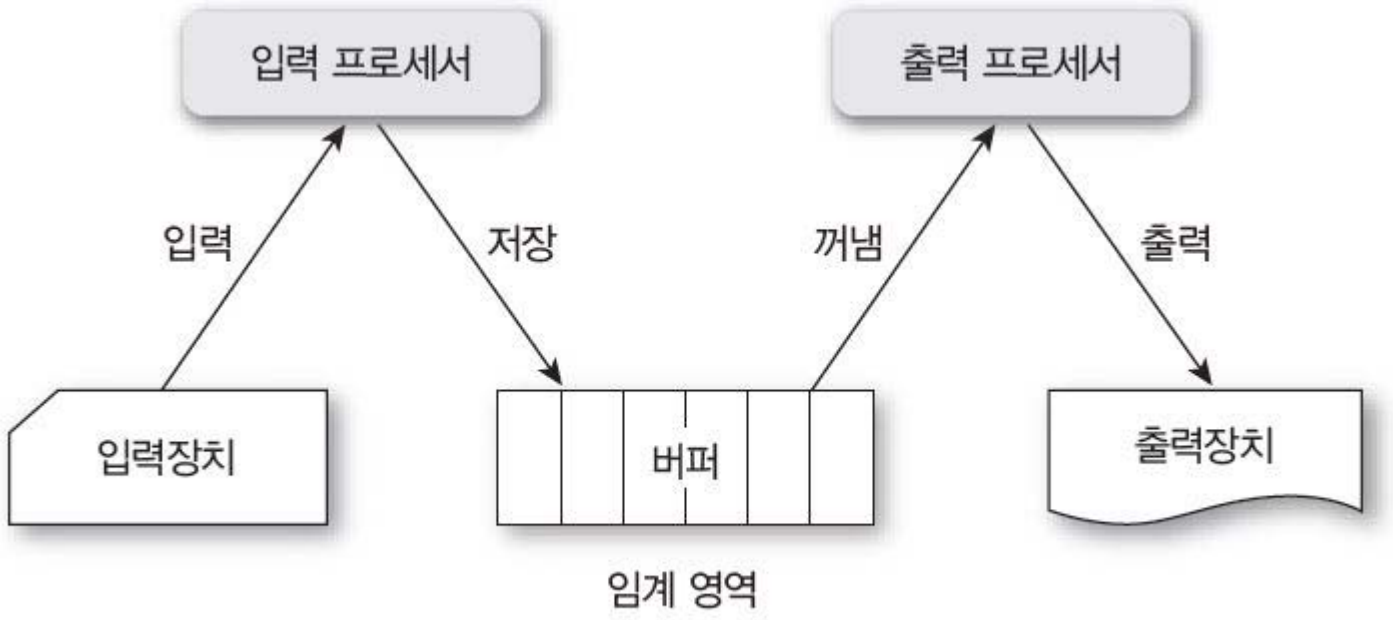


그림 4-14 **임계 영역 예**

2. 임계영역

■ 임계 영역 이용한 상호배제

- 간편하게 상호배제 구현 가능(자물쇠와 열쇠 관계)
  - 프로세스가 진입하지 못하는 임계 영역(자물쇠로 잠근 상태)
- 어떤 프로세스가 열쇠 사용할 수 있는지 확인하려고 검사 하는 동작과 다른 프로세스 사용 금지하는 동작으로 분류
- 예

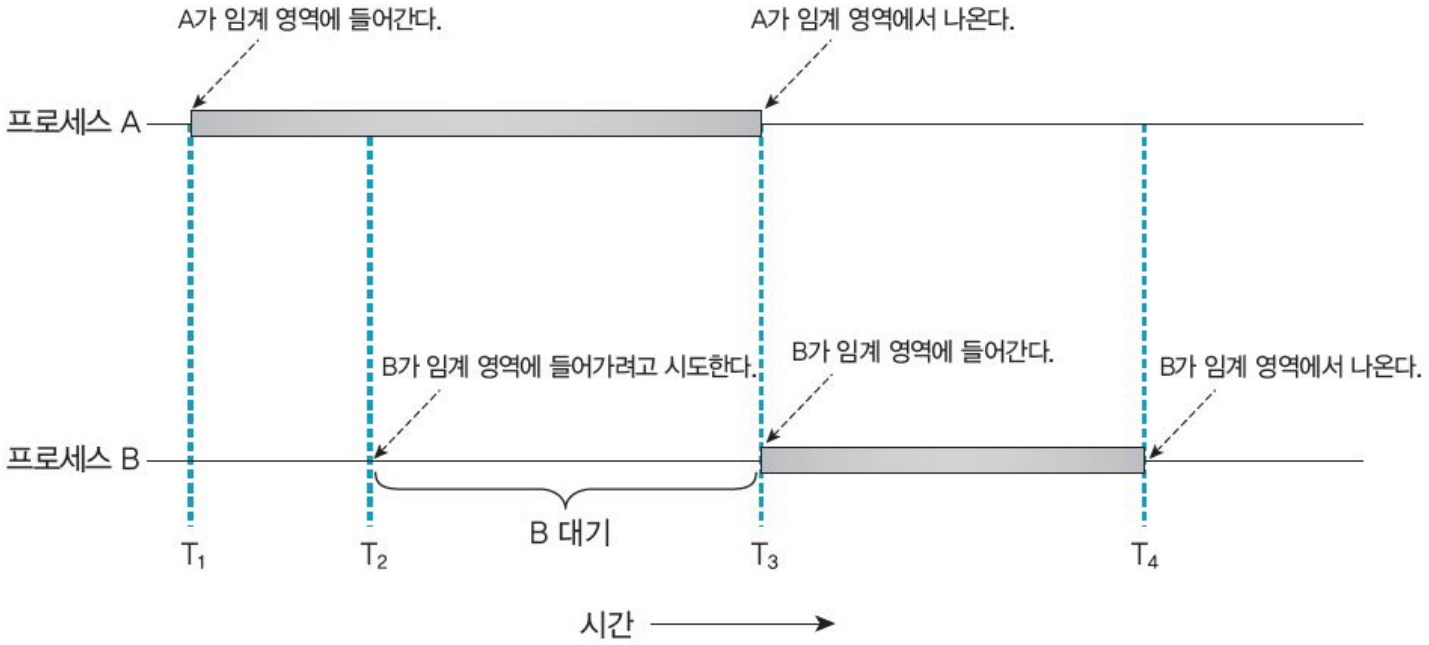


그림 4-15 **임계 영역을 이용한 상호배제**

2. 임계영역

■ 병행 프로세스에서 영역 구분

```
do {
        while (turn != i);      ← 진입 영역
        /* 임계 영역 */
        turn = j;               ← 탈출 영역
        /* 나머지 영역 */
} while (TRUE)
```

그림 4-16 병행 프로세스에서 영역 구분

■ 임계 영역의 조건

❶ 상호배제 : 어떤 프로세스가 임계 영역에서 작업 중, 다른 프로세스 임계 영역 진입 불가

❷ 진행 : 임계 영역에 프로세스가 없는 상태에서 어떤 프로세스가 들어갈지 결정

❸ 한정 대기 : 다른 프로세스가 임계 영역을 무한정 기다리는 상황 방지 위해 임계 영역에 한 번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

■ 생산자·소비자 문제

- 운영체제에서 비동기적으로 수행하는 모델
- 생산자 프로세스가 생산한 정보를 소비자 프로세스가 소비하는 형태
- 생산자/소비자, 판독자/기록자(입력기/출력기) 문제
  - 생산자(데이터 생산, 공유 객체에 저장)
  - 소비자(공유 객체로부터 데이터 읽음)
- 여러 프로세스가 공통 작업 수행을 위해 서로 협동하고, 병행 처리되는 대표적인 예
- 상호배제와 동기화가 필요하며 세마포어를 이용해 구현
- 운영체제에서 비동기적으로 수행하는 모델로 생산자 프로세스는 소비자 프로세스가 소비하는 정보를 생산

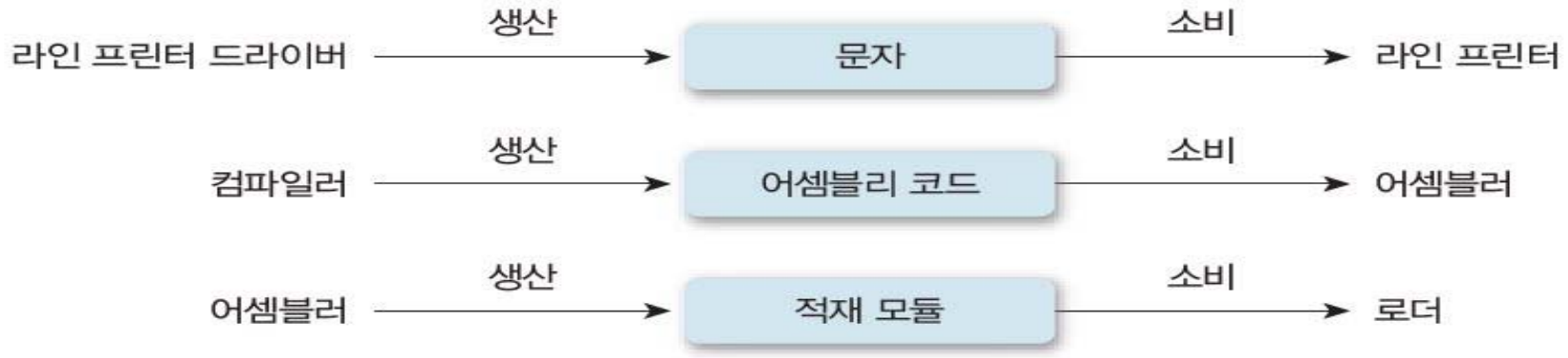


그림 4-17 생산자 · 소비자 프로세스의 관계

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

■ 생산자의 소비자에게 데이터 전송

- 소비자가 데이터를 받을 준비를 마칠 때까지 생산자는 버퍼로 데이터 전송

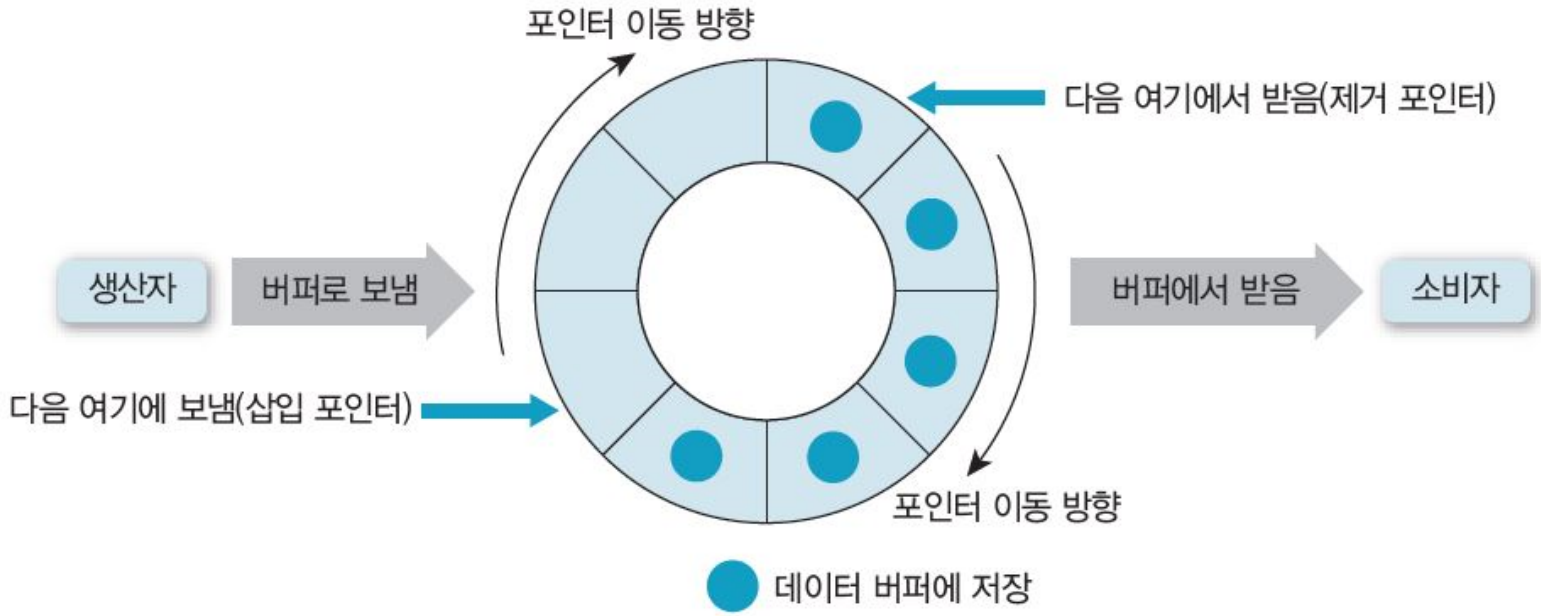


그림 4-18 공유 버퍼를 이용한 데이터 전송과 수신

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

■ 생산자와 소비자의 공유 버퍼

- 생산자와 소비자 프로세서들을 병행 실행하기 위해 공유 버퍼가 필요함
- 생산자의 데이터 생산 속도와 소비자의 데이터 소비 속도는 서로 독립적이므로 버퍼가 필요함
  - 생산자와 소비자는 같은 버퍼에 접근하므로 동시에 사용할 수 없음
- 생산자가 이미 채워진 버퍼에 더 채우거나, 소비자가 빈 버퍼에서 데이터를 꺼낼 때 문제 발생
- 속도가 다른 생산자와 소비자가 데이터를 일시 저장할 수 있는 버퍼 사용 시 버퍼는 다음의 세 가지 상태 중 하나

▪ 생산자는 버퍼가 꽉 차면 더 이상 생산 불가, 소비자는 버퍼가 비면 데이터 소비 불가

▪ 공유 버퍼의 상태

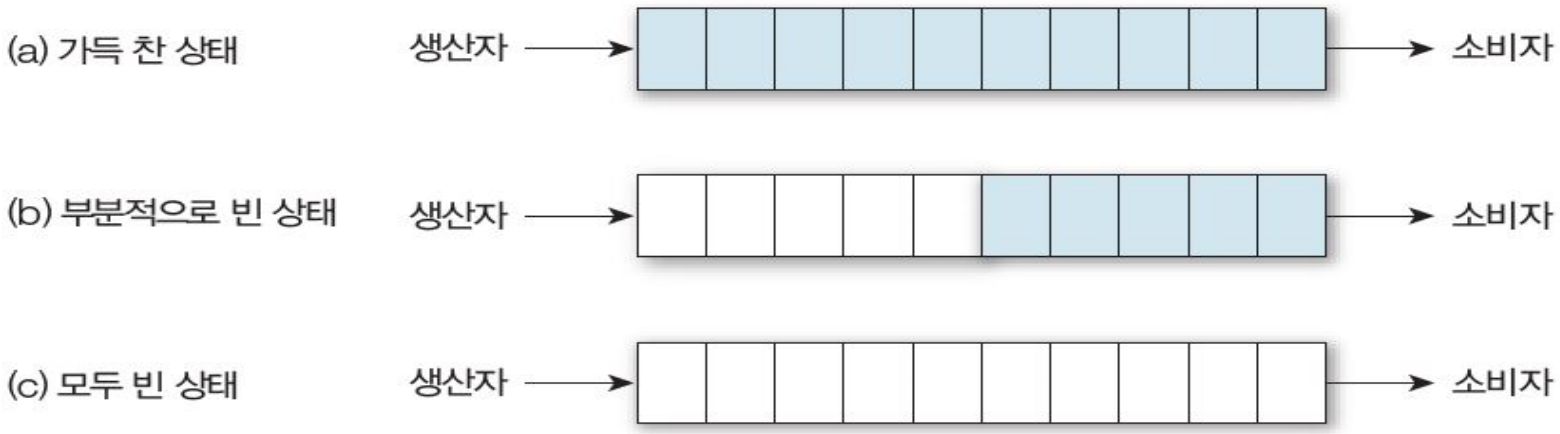


그림 4-19 공유 버퍼의 세 가지 상태

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- 무한 버퍼
  - 생산자와 소비자가 독립적으로 알고리즘 수행

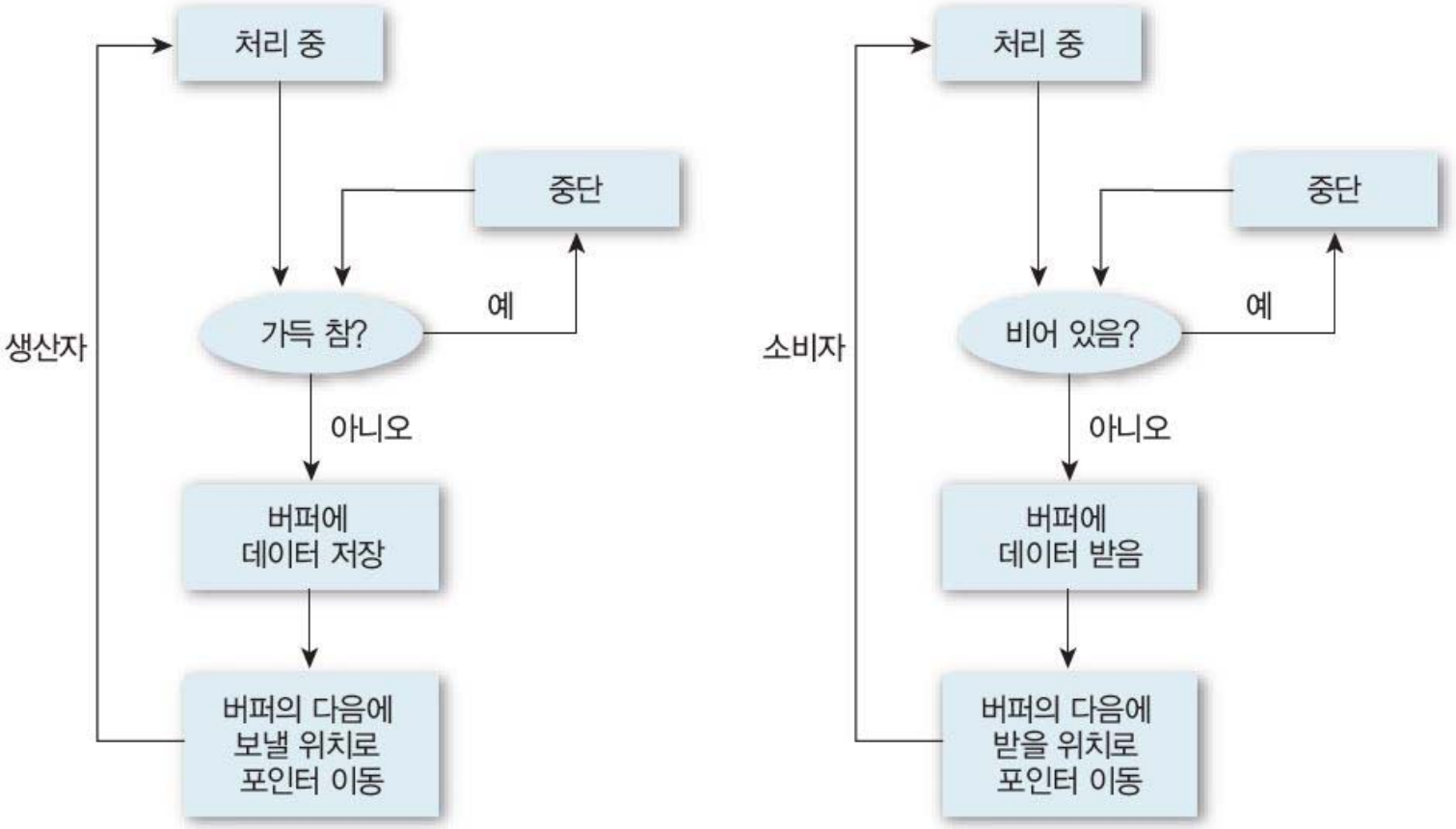


그림 4–20 무한 버퍼를 이용한 생산자–소비자 알고리즘

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- 유한 버퍼
  - 논리적 포인터 in과 out 2개로 버퍼 순환 배열

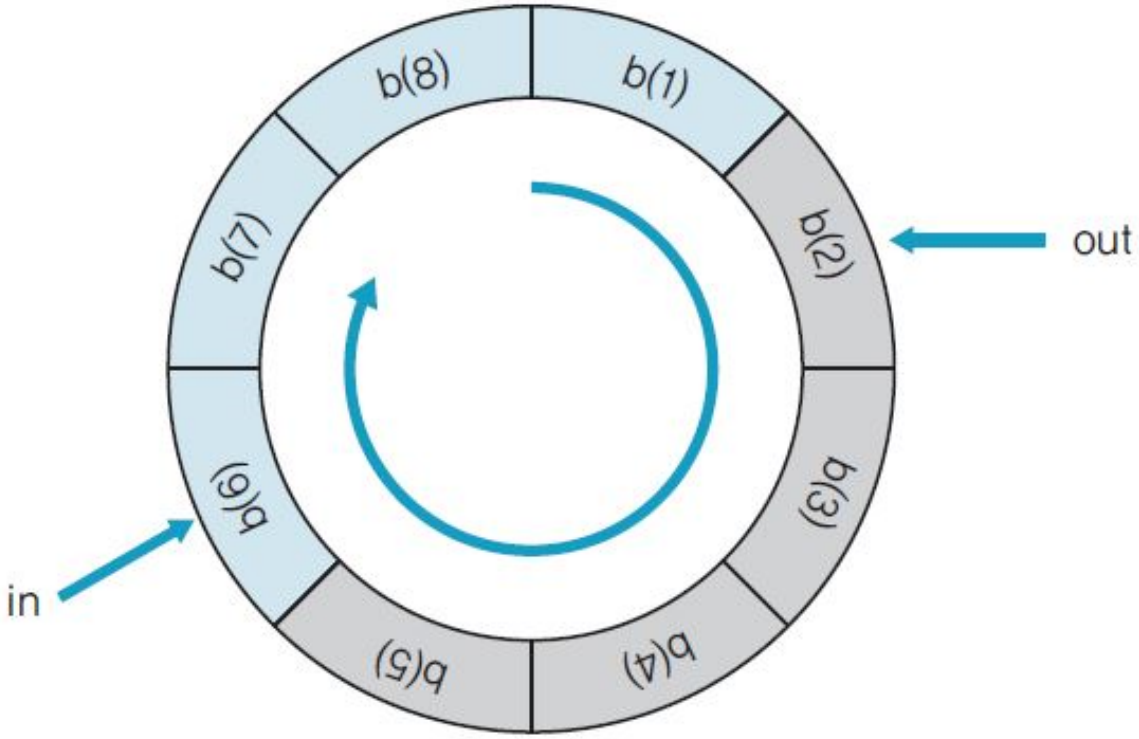


그림 4-21 버퍼 b의 구조(회색은 데이터가 채워진 공간)

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- [그림 4-21] 버퍼 구조 프로그램 구현

예제 4-1 공유 데이터의 선언

```
#define BUFFER_SIZE 10    // 버퍼 크기
typedef struct {
    DATA data;
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- 생산자 프로세스는 생산하는 새로운 원소를 지역변수 nextProduced에 저장, 소비자 프로세스는 소비하는 원소를 지역변수 nextConsumed에 저장 각 프로세스 구현

예제 4-2 생산자와 소비자 프로세스

생산자 프로세스

```
item nextProduced;

while (true) {
    // 버퍼가 가득 차 아무 일도 하지 않음
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

소비자 프로세스

```
item nextConsumed;

while (true) {
    // 버퍼가 비어 아무 일도 하지 않음
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

경쟁 상태race condition

경쟁상태의 개념

- 여러 프로세스가 동시에 공유 데이터에 접근 시, 접근 순서에 따라 실행 결과 달라지는 상황 말함
- 공유데이터에 마지막으로 남는 데이터의 결과 보장할 수 없는 상황
- 장치나 시스템이 둘 이상의 연산 동시 실행 시, 어느 프로세스를 마지막으로 수행한 후 결과를 저장했느냐에 따라 오류가 발생하므로 적절한 순서에 따라 수행 해야 함
- 읽기와 쓰기 명령을 거의 동시에 실행해야 한다면, 기본적으로 읽기 명령을 먼저 수행 후 쓰기 명령 수행하는 접근

경쟁 상태의 예방

- 병행 프로세스들을 동기화해야 함(임계 영역 이용한 상호배제로 구현)
- 즉, 공유 변수 counter를 한 순간에 프로세스 하나만 조작할 수 있도록 해야 하는 임계 영역과 counter 연산하는 부분을 임계 영역으로 설정하여 상호배제하는 방법으로 해결

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- **임계 영역**
  - 수정 가능한 공유 데이터에 접근
  - 한번에 한 스레드만 특정 자원에 접근 실행 가능
    - 무한 루프 등 잘못된 코드 없도록 신중
  - 종료 정리 작업
    - 스레드 종료 시, 상호 배제를 해제

- **상호 배제 프리미티브**
  - 상호 배제와 관련 가장 기본적인 연산 호출
  - 스레드의 임계영역에서 일어나는 일을 캡슐화
    - enterMutualExclusion()
      - 임계 영역에 들어가려고 할 때
    - exitMutualExclusion()
      - 임계 영역을 나올 때

3. 생산자·소비자 문제와 상호배제를 해결하는 초기의 시도

- 생산자·소비자 문제에서 임계 영역

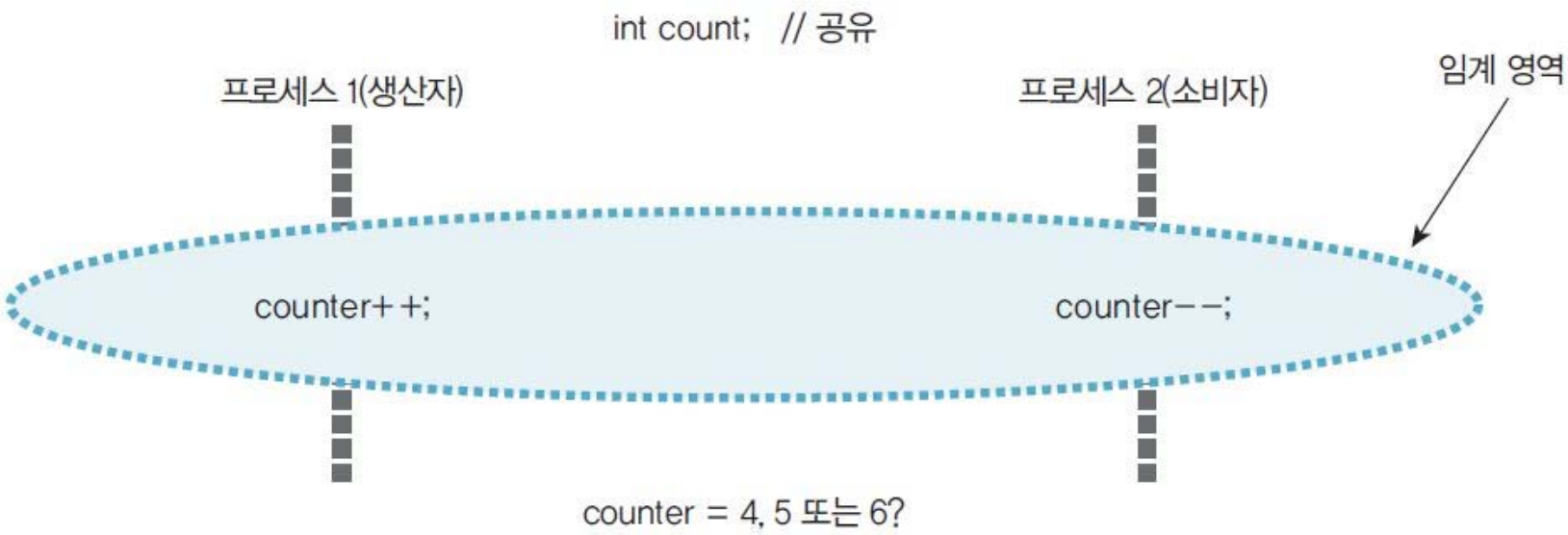


그림 4-22 생산자 · 소비자 문제에서 임계 영역

Section 03 상호배제 방법들

■ 상호배제 방법

표 4-1 상호배제 방법들

수준	방법	종류
고급	소프트웨어로 해결	• 데커의 알고리즘 • 램포트의 베이커리(빵집) 알고리즘 • 크누스의 알고리즘 • 핸슨의 알고리즘 • 다익스트라의 알고리즘
	소프트웨어가 제공 : 프로그래밍 언어와 운영체제 수준에서 제공	• 세마포 • 모니터
저급	하드웨어로 해결 : 저급 수준의 원자 연산	TestAndSet[TAS](테스)

1. 데커의 알고리즘

데커의 알고리즘 개념

- 두 프로세스가 서로 통신하려고 공유 메모리를 사용하여 충돌 없이 단일 자원을 공유할 수 있도록 허용하는 것
- 다익스트라가 임계 영역 문제에 적용
- 병행 프로그래밍 상호배제 문제의 첫 번째 해결책
- 각 프로세스 플래그 설정 가능, 다른 프로세스 확인 후 플래그 재설정 가능
- 프로세스가 임계 영역에 진입하고 싶으면 플래그 설정하고 대기

첫 번째 버전(고정된 동기화와 바쁜 대기 문제 야기)

- 하드웨어의 도움없이 프로세스 두 개의 상호 배제 문제를 해결
- 제어를 위해 변수 사용
- 임계 영역이 사용가능한지 계속적으로 확인
  - 바쁜 대기(busy waiting)
  - 중요한 프로세서 시간 낭비

- 고정된 동기화 문제
  - 한 스레드가 다른 스레드보다 빈번히 임계영역 사용한다면
    - 더 빠른 스레드가 더 느린 스레드의 속도에 맞춰 제한
  - 각 스레드는 엄격한 교대에 의해 실행

1. 데커의 알고리즘

- **두 번째 버전(상호 배제 위반)**
  - 고정된 동기화 문제 해결
    - 두 개의 변수 사용
  - 상호 배제 보장하지 않음
    - 두 스레드가 동시에 임계 영역 사용 가능
  - 적절하지 않은 방법
- **세 번째 버전(교착 상태 야기)**
  - 임계 영역 테스트 전 자신의 플래그 설정
    - 상호 배제 보장
  - 교착 상태
    - 두 스레드가 동시에 자신의 플래그 설정 가능
    - While문만 반복

1. 데커의 알고리즘

- **네 번째 버전(무기한 연기 야기)**
  - Loop를 도는 각 스레드가 짧은 기간 자신의 플래그를 false로 설정
  - 상호 배제 보장, 교착 상태 방지
  - 무기한 연기
    - 두 스레드가 같은 시간에 같은 값으로 플래그 설정 가능

- **데커의 알고리즘(적합한 해결책)**
  - 선호 스레드 개념 사용
    - 임계 영역에 여러 스레드 접근 시 먼저 진입 가능
      - 충돌 문제 해결
  - 상호 배제 보장
  - 교착 상태, 무기한 연기 방지

1. 데커의 알고리즘

■ 데커의 알고리즘을 사용한 상호배제(1)

예제 4-5 데커의 알고리즘을 이용한 상호배제

```
// 프로세스가 공유하는 데이터 flag[] : 부울(boolean) 배열, turn : 정수
flag[0] = false;
flag[1] = false;
turn = 0;                       // 공유 변수, 0 또는 1

// 프로세스 P0:                 // 프로세스 P0의 임계 영역 진입 절차
❶ flag[0] = true;               // P0의 임계 영역 진입 표시
❷ while (flag[1] == true) {     // P1의 임계 영역 진입 여부 확인
      if (turn == 1) {          // P1이 진입할 차례가 되면
       ❹ flag[0] = false;       // 플래그를 재설정하여 P1에 진입 순서 양보
         while (turn == 1) {    // turn을 바꿀 때까지 대기
      ❺      // 바쁜 대기
         }
          flag[0] = true;       // P1이 임계 영역에 재진입 시도
      }
}
❸ /* 임계 영역 */;
turn = 1;                       // P1에 진입 순서 제공
flag[0] = false;                // P0의 임계 영역 사용 완료 지정
/* 나머지 영역 */;               // P0이 나머지 영역 수행
```

❶에서 P0은 flag[0]을 true로 설정, 자신이 임계 영역으로 들어간다는 사실 알림
❷에서 while 문 검사하여 P1의 임계 영역 진입 여부 확인, P1의 flag[1]이 false이면
❸ P0이 임계 영역으로 진입하고, true이면
❹ P1이 임계 영역에 진입할 차례라서 플래그 false로 재설정 후
❺ while 문에서 순환하며 대기
여기서 공유 변수 turn은 두 프로세스 P0과 P1이 동시에 임계 영역으로 들어가려고 충돌하는 것 방지

1. 데커의 알고리즘

■ 데커의 알고리즘을 사용한 상호배제(2)

```
// 프로세스 P1
flag[1] = true;
while (flag[0] == true) {
    if (turn == 0) {
        flag[1] = false;
        while (turn == 0) {
            // 바쁜 대기
        }
        flag[1] = true;
    }
}
/* 임계 영역 */;
turn = 0;
flag[1] = false;
/* 나머지 영역 */;
```

1. 데커의 알고리즘

데커의 알고리즘 특징

- 특별한 하드웨어 명령문 필요 없음
- 임계 영역 바깥에서 수행 중인 프로세스가 다른 프로세스들이 임계 영역 진입 막지 않음
- 임계 영역에 들어가기를 원하는 프로세서 무한정 기다리게 하지 않음

1. 데커의 알고리즘

- 기타 유용한 상호배제 알고리즘

- 다익스트라dijkstra
  - 최초로 프로세스 n개의 상호배제 문제를 소프트웨어적으로 해결
  - 실행 시간이 가 장 짧은 프로세스에 프로세서 할당하는 세마포 방법
  - 가장 짧은 평균 대기시간 제공

- 크누스knuth
  - 이전 알고리즘 관계 분석 후 일치하는 패턴을 찾아 패턴의 반복을 줄여서 프로세스에 프로세서 할당
  - 무한정 연기할 가능성을 배제하는 해결책을 제시했으나, 프로세스들이 아주 오래 기 다려야 함

1. 데커의 알고리즘

● 기타 유용한 상호배제 알고리즘

• 램포트lamport

- ➢사람들로 붐비는 빵집에서 번호표 뽑아 빵 사려고 기다리는 사람들에 비유해서 만든 알고리즘
- ➢준비 상태 큐에서 기다리는 프로세스마다 우선순위를 부여하여 그 중 우선순위가 가장 높은 프로세스에 먼저 프로세서를 할당, '램포트의 베이 커리(빵집) 알고리즘' 이라고 함
- ➢n-thread 상호 배제
  - ✓ 티켓 뽑기 시스템 사용
  - ✓ 티켓 번호는 오름차순
  - ✓ 티켓의 번호가 가장 낮을 때, 각 스레드 실행

- ➢ 장점
  - ✓ 가장 간단한 n-thread 상호 배제 알고리즘
  - ✓ 멀티프로세서 시스템에서 동작
  - ✓ FCFS순서 가능
    - ▪ 임계 영역에 들어가려고 대기하는 스레드들이 같은 티켓 번호 가지지 않을 시
  - ✓ 교착 상태, 무기한 연기 발생 하지 않음
    - ▪ 하드웨어 장치의 고장으로 인한 전체 시스템 실패 없음

1. 데커의 알고리즘

- 기타 유용한 상호배제 알고리즘

- 핸슨brinch hansen
  - ➢실행 시간이 긴 프로세스에 불리한 부분을 보완하는 것
  - ➢대기시간과 실행 시간을 이용하는 모니터 방법
  - ➢분산 처리 프로세서 간의 병행성 제어 많이 발표

- 페터슨의 알고리즘
  - ➢데커의 알고리즘보다 간단
  - ➢바쁜 대기, 선호 스레드 사용
  - ➢상호 배제 프리미티브 수행 시, 적은 단계 요구
  - ➢스레드가 갑자기 종료 되지 않는 한, 교착 상태, 무기한 연기는 일어 나지 않음

2. TestAndSet^{TAS}(테스) 명령어

TestAndSet 명령어의 개념(하드웨어를 통한 상호배제 구현)

- 공유 변수 수정하는 동안 인터럽트 발생 억제하여 임계 영역 문제 간단 해결
- 항상 적용할 수 없고 실행 효율 현저히 떨어짐
- 소프트웨어적인 해결책은 더 복잡하고 프로세스가 2개 이상일 때는 더 많은 대기 가능성
- 메모리 영역의 값에 대해 검사와 수정을 원자적으로 수행할 수 있는 하드웨어 명령어
- 알고리즘이 간단, 하나의 메모리 사이클에서 수행하여 경쟁 상황 해결
- 기계명령어 2개(원자적 연산 명령어 TestAndSet, TestAndSet에 지역변수 lock 설정명령어)
- 일부 시스템에서 원자 명령어의 하나로, 읽기와 쓰기 모두 제공
- 해당 주소의 값을 읽고 새 값으로 교체하면서 해당 메모리 위치의 이전 값 반환
- testAndSet(a,b)
  - b의 값을 읽고, a에게 복사, b를 true로 설정
  - 단일 프로세서 또는 메모리를 공유하는 다중 처리 환경과 관계없이 적용되며, 간단하여 쉽게 적용된다는 장점을 가짐
  - 임계영역에 진입하려는 프로세스에 바쁜 대기가 발생
  - 무한 연기 가능성이 발생할 수는 있지만 프로세스 수가 많으면 거의 발생하지 않음

원자적 연산atomic operation
중단 없이 실행하고 중간에 다른 사람이 수정할 수 없는 최소 단위 연산, 메모리의 1비트에서 작동하고, 대다수 기계에서 워드의 메모리 참조, 할당은 원자적이지만 나머지 많은 명령은 원자적이지 않음

2. TestAndSet^{TAS}(테스) 명령어

■ TestAndSet 명령어의 장점과 단점

표 4-2 TestAndSet 명령어의 장점과 단점

장점	사용자 수준에서 가능하다. - 메인 메모리를 공유하는 다중 프로세서나 단일 프로세서에서 프로세스 수에 관계없이 적용할 수 있다. - lock 변수 수에 상관없이 구현할 수 있다. - 구현이 단순하고 확인이 용이하다. - 다중 임계 영역을 지원한다.
단점	• 바쁜 대기 발생 - 프로세서 시간 소모가 크다. - 대기 프로세스는 비생산적, 자원이 소모되는 대기 루프에 남는다. • 기아 상태 발생 : 프로세스가 임계 영역을 떠날 때 프로세스 하나 이상을 대기하는 경우 가능하다. • 교착 상태 발생 : 플래그는 우선순위가 낮은 프로세스가 재설정할 수 있지만, 우선순위가 높은 프로세스가 선점한다. 따라서 우선순위가 낮은 프로세스는 lock을 가지고, 우선순위가 높은 프로세스가 이것을 얻으려고 시도할 때 높은 우선순위 프로세스는 무한정 바쁜 대기가 될 것이다.

3. 세마포semaphore

■ 세마포 개념과 동작

- 다익스트라가 테스트 명령어의 문제 해결을 위해 제안
- 상호배제 및 다양한 연산의 순서도 제공
- 세마포 값은 true나 false로, P와 V연산과 관련. 네덜란드어로 P는 검사Proberen, V 증가Verhogen 의미. 음이 아닌 정수 플래그 변수
- 세마포를 의미하는 S는 표준 단위 연산 P(프로세스 대기하게 하는 wait 동작, 임계 영역에 진입하는 연산)와 V(대기 중인 프로세스 깨우려고 신호 보내는 signal 동작, 임계 영역에서 나오는 연산)로만 접근하는 정수 변수
- 예

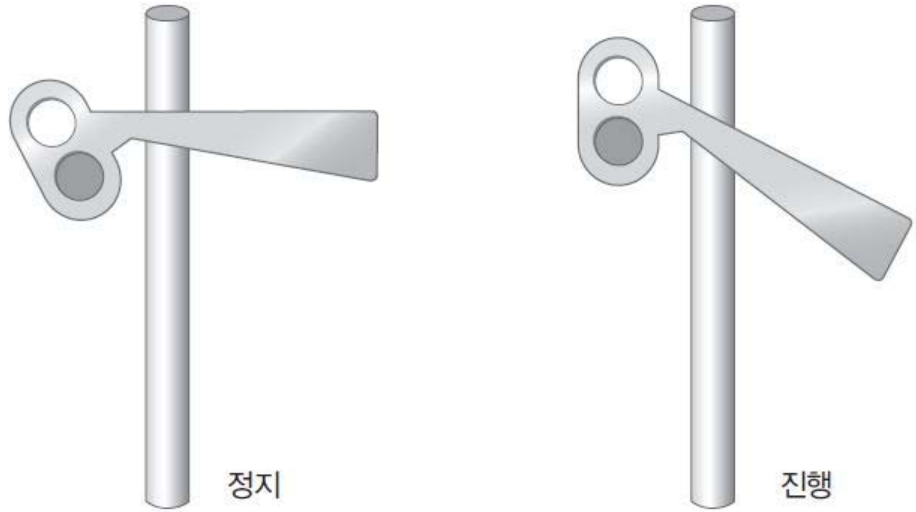


그림 4-23 세마포 예 : 열차 차단기

4. 모니터monitor

■ 모니터의 개념과 구조

- 세마포의 오용으로 여러 가지 오류가 쉽게 발생하면 프로그램 작성 곤란
  이런 단점 극복 위해 등장
- 핸슨Hansen 제안, 호Hoare 수정한 공유 자원과 이것의 임계 영역 관리 소프트웨어 구성체
- 사용자 사이에서 통신하려고 동기화하고, 자원에 배타적으로 접근할 수 있도록 프로세스가 사용하는 병행 프로그래밍 구조
- 모니터의 구조

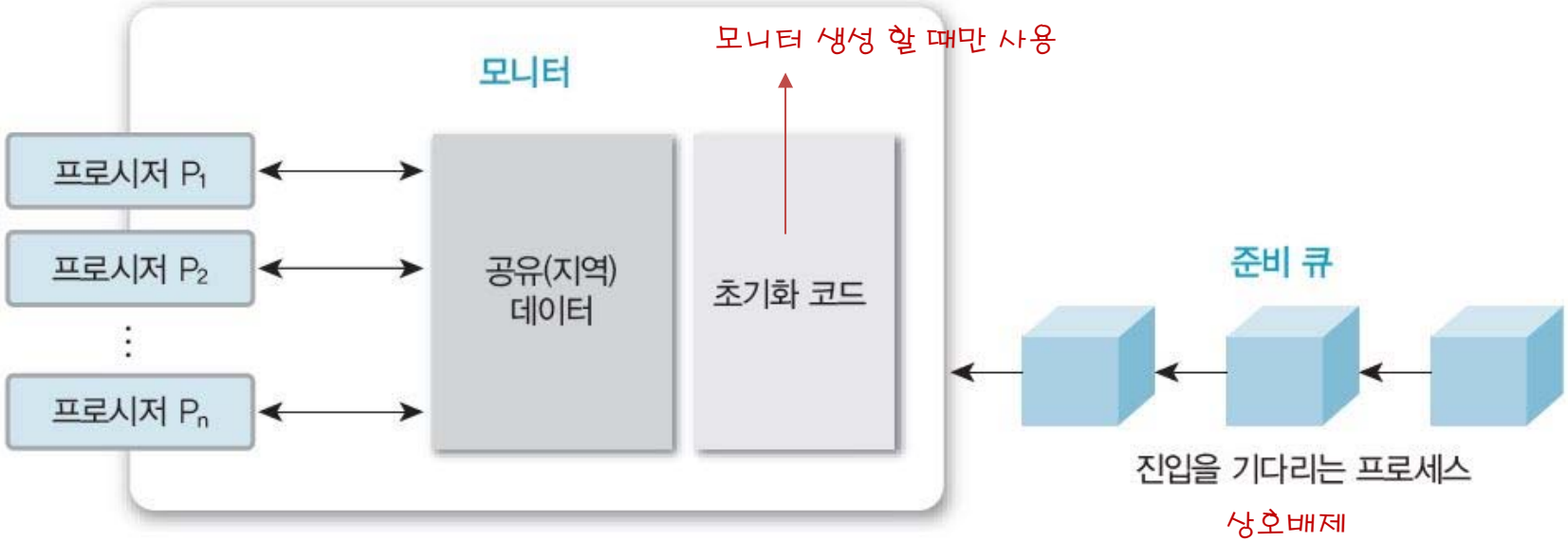


그림 4-30 모니터의 구조

4. 모니터monitor

조건 변수가 있는 모니터의 구조

- 모니터는 여러 측면에서 임계 영역과 비슷, 프로세스 실행 동안 상호배제와 동기화 제공
- 강력함 떨어져 동기화 방법 추가 정의해야 함
- 모니터 외부에 있는 프로세스가 모니터에 있는 프로세스 수행할 때까지 외부에서 기다려야 할 때는 특정 조건에 따라 실행 재개 결정
- 모니터는 조건 변수와 프로세스를 대기할 수 있는 상황 연관 시킴
- 하나 이상의 모니터 조건 변수 정의하여 모니터 안에서 작업 동기화

```
condition x, y;
```

4. 모니터monitor

- 조건 변수가 있는 모니터의 구조의 예

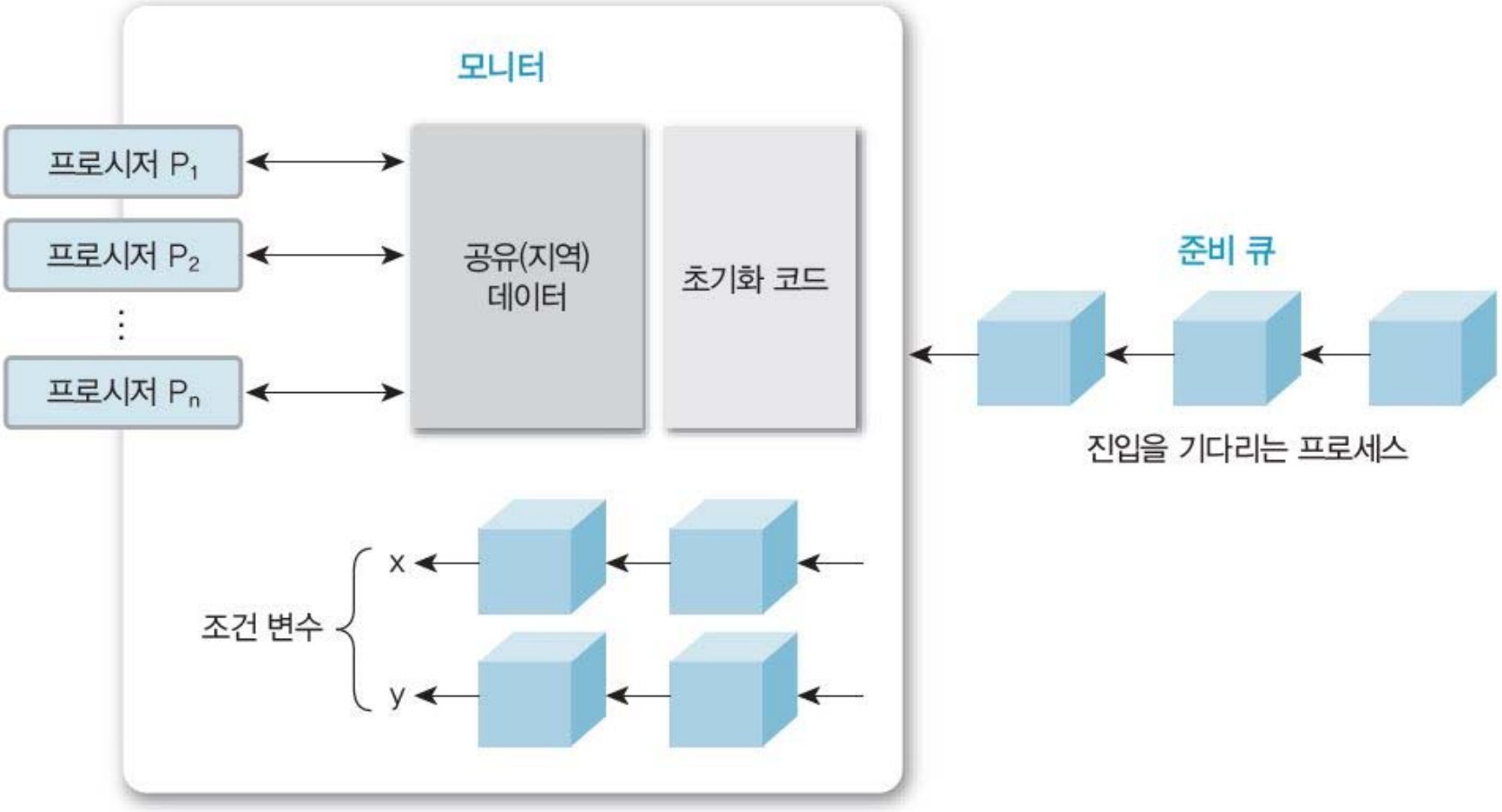


그림 4-31 조건 변수가 있는 모니터의 구조

4. 모니터monitor

모니터와 세마포 비교

- 모니터의 조건 변수에서 x.wait와 x.signal 연산은 계수 세마포에서 P와 V 연산과 비슷
- signal 연산이 다른 프로세스의 차단을 해제하는 동안 wait 연산이 프로세스의 실행을 차단할 수 있는데, 이는 약간 차이가 있음. 프로세스가 P 연산을 실행하면 계수 세마포가 0보다 클 수 있어 해당 프로세스를 반드시 차단하지 않음. 반면 wait 연산 실행하면 항상 프로세스 차단. 세마포의 V 연산 호출하면 일단 (무조건) counter 증가, 대기 작업 유무 확인. 대기 작업이 있으면 큐에서 대기 작업 꺼내고, 해당 작업 실행([예제 4-13] 참조)
- 반면 모니터는 signal 연산(세마포의 V 연산에 대응) 수행 시 대기 중인 작업 없으면 signal 호출은 아무런 효과 발생 않음([예제 4-16] 참조)
- 세마포는 V 연산으로 사용자가 지연 없이 실행 재개, 반면 모니터는 signal 연산으로 모니터 잠금 해제할 때만 다시 시작. 따라서 병렬 프로그래밍에서는 세마포보다 모니터가 더 오류가 적고 쉽게 작성
- 모니터의 주된 단점은 프로그래밍 언어의 일부로 구현하고 컴파일러가 그 코드 생성해야 한다는 점. 이는 컴파일러가 병행 프로세스에서 임계 영역에 접근 제어할 수 있는 운영 체제를 이해해야 한다는 부담. 또 병행성 지원하는 자바Java, C#, 비주얼 베이직Visual Basic, 에이다Ada 등 언어만 모니터 지원