5장 병행성: 상호배제와 동기화

5장의 학습 목표

- 병행성(concurrency)의 원리와 주요 용어를 이해한다.
- 경쟁상태(race condition)의 문제점에 대해 이해한다.
- 경쟁상태 해결을 위한 운영체제 고려사항을 이해한다.
- 상호배제(mutual exclusion)의 필요성 및 이를 지원하기 위한 하드웨어 수준 접근방법을 이해한다.
- 세마포어를 정의하고 동작 방식을 이해한다.
- 모니터를 정의하고 동작 방식을 이해한다.
- 메시지 전달을 이용한 상호배제 기법을 이해한다.
- 상호배제 문제의 대표적인 예인 생산자/소비자 문제와 판독자/기록자 문제를 이해하고 해결 방법을 이해한다.

목 차

5.1 병행성 원리

- **멀티프로세스**
  - 프로세스/쓰레드 관리를 위한 현대 운영체제의 설계 핵심 주제
    - Multiprogramming
    - Multiprocessing
    - Distributed Processing
  - Eg) 인터리빙, 오버랩핑
    - 동일한 문제 야기

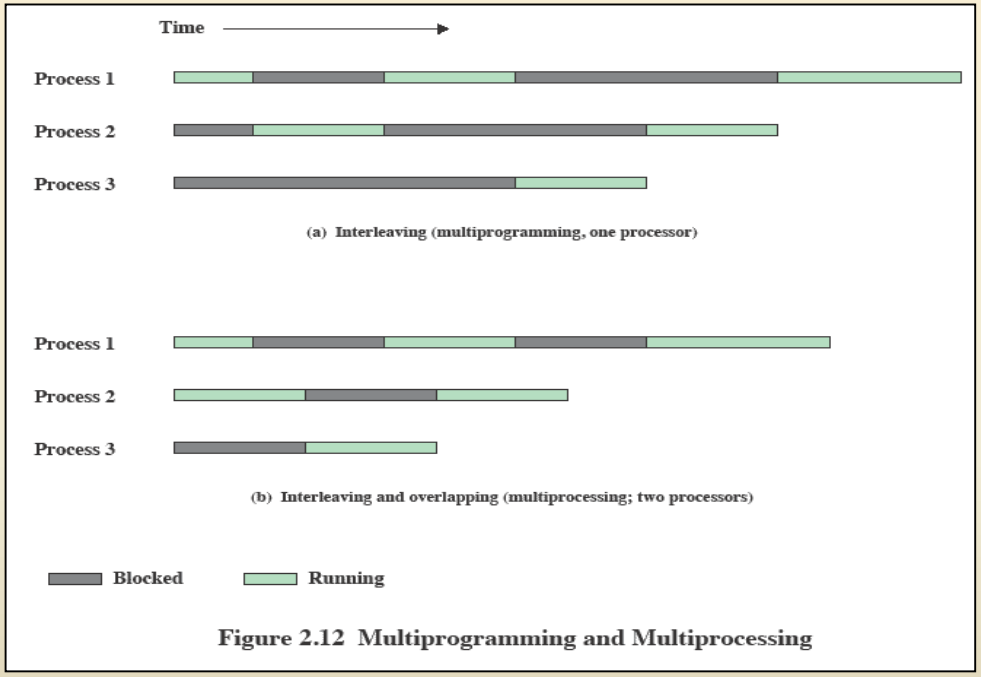


Figure 2.12 Multiprogramming and Multiprocessing

  - Big Issue is Concurrency(병행성)
    - Managing the interaction of all of these processes

병행성 원리

- **병행성: 다음과 같은 3가지 상황에서 발생**

다수의 응용

다수의 활동 중인 응용들 간에 처리시간의 동적 공유를 위해 멀티프로그래밍이 발전

구조화된 응용

모듈화된 설계 원칙과 구조적인 프로그램이 발전되면서 일부 응용이 병행 프로세스의 집합

운영체제 구조

운영체제도 다수의 프로세스와 쓰레드의 집합으로 구현

병행성 원리

- **병행 처리의 문제점**
  - 전역 자원의 공유가 어렵다.
  - 운영체제가 자원을 최적으로 할당하기 어렵다.
  - 프로그래밍 오류를 찾아내는 것이 어렵다.

- **인터리빙이나 오버래핑으로 인해 발생하는 문제점은 단일처리기 시스템에서나 다중처리기 시스템에서 동일**
  - 단일처리기: 프로세스 수행의 상대적인 속도 예측이 어려움
    - 다른 프로세스의 행동에 종속
    - OS의 스케줄링 정책에 의존
    - OS의 인터럽트 처리 방법에 따라 달라짐

병행성 원리

- **병행성과 관련된 주요 용어**

표 5.1 병행성과 관련 있는 주요 용어

원자적 연산(atomic operation)	하나 또는 여러 개의 명령어들로 구성된 함수 또는 액션으로 더 이상 분할할 수 없는 단위. 따라서 다른 어떤 프로세스도 중간 상태를 볼 수 없으며, 연산을 중단 시킬 수 없다. 이 명령어들은 모두 수행되거나 하나도 수행되지 않음이 보장된다. 원자성은 병행 프로세스들에게 고립(isolation)을 보장한다.
임계영역(critical section)	공유 자원을 접근하는 프로세스 내부의 코드 영역. 다른 프로세스가 이 영역에 있을 때, 이 프로세스 또한 이 영역을 수행할 수 없다.
교착상태(deadlock)	두개 이상의 프로세스들이 더 이상 진행을 할 수 없는 상태. 각 프로세스가 다른 프로세스의 진행을 기다리면서 대기하고 있을 때 발생한다.
라이브락 (livelock)	두개 이상의 프로세스들이 다른 프로세스의 상태 변화에 따라 자신의 상태를 변화시키는 작업만 수행하고 있는 상태. 각 프로세스들이 열심히 수행은 하고 있지만, 실제 수행하는 작업은 유용한 작업이 아니다.
상호 배제(mutual exclusion)	한 프로세스가 공유 자원을 접근하는 임계영역 코드를 수행하고 있으면 다른 프로세스들은 공유 자원을 접근하는 임계영역 코드를 수행할 수 없다는 조건.
경쟁상태(race condition)	두개 이상의 프로세스가 공유 자원을 동시에 접근하려는 상태. 최종 수행 결과는 프로세스들의 상대적인 수행 순서에 따라 달라질 수 있다.
기아(starvation)	특정 프로세스가 수행 가능한 상태임에도 불구하고 매우 오랜 기간 동안 스케줄링 되지 못하는 경우.

☞ 그 외: 공유 자원(Shared Resource), 동기화(Synchronization)

병행성 원리

- **병행성의 다른 예들**
  - 공유 함수(shared functions)

```
// shared functions
void echo()
{
        chin = getchar();
        chout = chin;
        putchar(chout);
}
```

```
Process P1
  .
  chin = getchar();
  .
  chout = chin;
  putchar(chout);
  .
  .
```

```
Process P2
  .
  .
  chin = getchar();
  chout = chin;
  .
  putchar(chout);
  .
```

  - 연관된 공유 데이터 집합(a = b 라는 일관성 유지 필요)

```
// coordinated data set
Process P1
  a = a + 1;
  b = b + 1;
  ...
Process P2
  b = b * 2;
  a = a * 2;
```

```
// Real Execution
  ...
  a = a + 1;
  b = b * 2;
  b = b + 1;
  a = a * 2;
  ...
```

병행성 원리

- **경쟁상태(Race condition)**
  - 다중 프로세스/쓰레드가 공유 데이터를 읽거나 쓸 때 발생
  - 최종 결과는 수행의 순서에 의해 결정됨
    - 경쟁(race)의 패자(loser)가 가장 마지막으로 데이터를 수정하며, 결국 최종 결과를 결정함
- **운영체제 고려사항**
  - 다양한 프로세스의 행위를 추적할 수 있어야 함
  - 각 프로세스에게 자원을 할당하거나 반납 받을 수 있어야 함
  - 한 프로세스가 소유한 자원이나 데이터를 다른 프로세스의 간섭으로부터 보호할 수 있어야 함
  - 수행 결과가 프로세스들의 수행 순서와 독립적일 수 있도록 보장해야 함

병행성 원리

- **프로세스 상호작용: 경쟁, 공유, 통신**

표 5.2 프로세스 상호작용

인식 정도 (Degree of Awareness)	관계 (Relationship)	한 프로세스가 다른 프로세스에게 미치는 영향	잠재적인 제어 문제
서로를 인식하지 못함	경쟁	• 한 프로세스의 수행 결과는 다른 프로세스들의 행위와는 독립 • 프로세스의 타이밍에 영향을 받을 수 있음	• 상호 배제 • 교착상태 (재사용 가능한 자원) • 기아상태
서로를 간접적으로 인식 (예. 자원 공유)	공유를 통한 협력	• 한 프로세스의 수행 결과는 다른 프로세스들로부터 얻은 정보에 의해 영향을 받을 수 있음 • 프로세스의 타이밍에 영향을 받을 수 있음	• 상호 배제 • 교착 상태 • (재사용 가능 자원) • 기아상태 • 데이터 일관성
서로를 직접적으로 인식 (예. 프로세스 간 통신)	통신을 통한 협력	• 한 프로세스의 수행 결과는 다른 프로세스들로부터 얻은 정보에 의해 영향을 받을 수 있음 • 프로세스의 타이밍에 영향을 받을 수 있음	• 교착 상태 • (소모성 자원) • 기아상태

병행성 원리

- **자원 경쟁**
  - 병렬 프로세스들이 같은 자원을 사용하려고 경쟁하면 충돌이 발생한다.
  - Eg. I/O 장치, 메모리, 처리기 시간, 클락 등

프로세스들이 경쟁하면 다음 3가지 제어 문제가 발생함

- **상호배제 (mutual exclusion)**
- **교착상태 (deadlock)**
- **기아 (starvation)**

상호배제

- **상호배제(mutual exclusion) 요구조건**
  - 어느 한 순간에는 오직 하나의 프로세스만이 임계영역(critical section)에 진입할 수 있다.
  - 임계영역이 아닌 곳에서 수행이 멈춘 프로세스는 다른 프로세스의 수행을 간섭해서는 안 된다.
  - 임계영역에 접근하고자 하는 프로세스의 수행이 무한히 미뤄져서는 안 된다. 즉, 교착상태(deadlock) 및 기아(starvation)가 일어나지 않아야 한다.
  - 임계영역이 비어 있을 때, 임계영역에 진입하려고 하는 프로세스가 지연되어서는 안 된다.
  - 프로세서의 개수나 상대적인 프로세스 수행 속도에 대한 가정은 없어야 한다.
  - 프로세스는 유한 시간 동안만 임계영역에 존재할 수 있다.

상호배제

- **상호배제 해결방법**
  - 어느 한 순간에는 오직 하나의 프로세스만이 임계영역에 진입
  - Illustration of Mutual Exclusion

```
/* 프로세스 1 */
void P1
{
  while (true) {
    /* 임계영역 이전코드 */;
    entercritical (Ra);
    /* 임계영역 */;
    exitcritical (Ra);
    /* 임계영역 이후코드 */;
  }
}
```

```
/* 프로세스 2 */
void P2
{
  while (true) {
    /* 임계영역 이전코드 */;
    entercritical (Ra);
    /* 임계영역 */;
    exitcritical (Ra);
    /* 임계영역 이후코드 */;
  }
}
```

• • •

```
/* 프로세스 n */
void Pn
{
  while (true) {
    /* 임계영역 이전코드 */;
    entercritical (Ra);
    /* 임계영역 */;
    exitcritical (Ra);
    /* 임계영역 이후코드 */;
  }
}
```

그림 5.1 상호 배제의 예

상호배제

- **entercritical(), exitcritical() 구현 방법**
  - 소프트웨어적 접근방법
    - 수행 부하가 높고, 논리적 오류의 위험성이 크다.
  - 하드웨어 지원
    - 인터럽트 금지(오버헤드가 크다.)
    - 특별한 기계 명령어: Test and Set, Exchange
  - 세마포어
  - 모니터
  - 메시지 전달

5.2 하드웨어 지원

- 인터럽트 금지

```
// 인터럽트 금지
while (true)
{
        /* disable interrupt */
        /* critical section */
        /* enable interrupt */
        /* remainder */
}
```

☞ multiprocessor??

- 특별한 기계 명령어

```
// 특별한 기계 명령어: compare & swap
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;

    oldval = *word;
    if (oldval ==  testval)
        *word = newval;
    return oldval;
}
```

```
// 특별한 기계 명령어: exchange
void exchange (int *register, int *memory)
{
    int temp;

    temp = *memory;
    *memory = *register;
    *register = temp;
}

// XCHG in IA, SWAP in ARM architecture
```

☞ 원자적 연산(atomic operation)

하드웨어 지원

- 특별한 기계 명령어 사용 예

```
/* 상호 배제 예제 프로그램 */
const int n = /* 프로세스 개수 */;
int bolt;
void P(int i)
{
    while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* 대기 */;
      /* 임계영역 */;
      bolt = 0;
      /* 임계영역 이후 코드 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap 명령어

```
/* 상호 배제 예제 프로그램 */
int const n = /* 프로세스 개수 */;
int bolt;
void P(int i)
{
    while (true) {
        int keyi = 1;
        do exchange (&keyi, &bolt)
        while (keyi != 0);
        /* 임계영역 */;
        bolt = 0;
        /* 임계영역 이후 코드 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange 명령어

그림 5.2 하드웨어 기법을 이용한 상호 배제

하드웨어 지원

- **특별한 기계 명령어 장점**
  - 임의 개수의 프로세스에 적용 가능
  - 단일 프로세서와 공유 메모리 기반 다중 프로세서에 모두 적용 가능
  - 간단하고 검증하기 쉬움
  - 서로 다른 변수를 사용하면 다중 임계영역 지원

- **특별한 기계 명령어 단점**
  - 바쁜 대기(Busy-waiting)
  - 기아상태 발생 가능
  - 교착상태 발생 가능
    - 예를 들어 우선 순위가 낮은 프로세스가 임계영역에 진입한 상태이고 우선 순위가 높은 프로세스가 그 임계영역에 진입하기 위해 바쁜 대기를 하고 있다면 교착상태 발생

병행성 기법

표 5.3 대표적인 병행성 기법

세마포어(Semaphore)	프로세스 간에 시그널(signal)을 주고받기 위해 사용되는 정수 값. 세마포어는 다음 3가지 원자적인 연산만을 지원한다: initialize, decrement, increment. Decrement 연산은 프로세스를 블록시킬 수 있다. 반면 increment 연산은 블록되었던 프로세스를 깨울 수 있다. 이 세마포어를 **카운팅 세마포어(counting semaphore)** 또는 **일반 세마포어(general semaphore)** 라고 한다.
이진 세마포어 (Binary Semaphore)	이 세마포어는 0 또는 1을 값으로 가질 수 있다.
뮤텍스 (Mutex)	이진 세마포어와 유사하다. 차이점은 뮤텍스에 락을 획득(값을 0으로 설정)한 프로세스가 반드시 그 락을 해제(값을 1로 설정)해야 한다는 것이다.
조건 변수 (Conditional Variable)	특정 조건이 만족될 때까지 프로세스나 쓰레드를 블록시키기 위해 사용하는 데이터 유형 (data type).
모니터 (Monitor)	추상화된 데이터 유형(abstract data type)을 만드는 프로그램 언어 생성자 (construct). 생성된 데이터 유형은 변수, 접근 함수, 초기화 함수 등을 캡슐화(encapsulation) 한다. 모니터 변수는 접근 함수를 통해서만 접근되며, 한 순간에 한 프로세스만 활동적으로 접근할 수 있다. 결국 접근 함수가 임계영역이 된다. 모니터는 대기하는 프로세스들을 위한 큐를 가지기도 한다.
이벤트 플래그 (Event Flags)	동기화에 사용되는 메모리 워드. 응용은 플래그의 각 비트에 서로 다른 이벤트를 연관시킬 수 있다. 쓰레드는 하나의 이벤트, 또는 여러 이벤트들의 조합을 대기할 수 있으며, 이를 위해 결국 대응되는 비트 또는 비트들을 검사한다. 쓰레드는 모든 비트가 만족되어야만 깨어날 수도 있으며 (AND), 하나의 비트만 만족되면 깨어날 수도 있다 (OR).
메일박스/메시지 (Mailbox/Message)	두 개의 프로세스들이 정보를 주고받는 수단. 동기화를 위해 사용될 수도 있다.
스핀락 (Spinlocks)	상호 배제 기법 중에 하나. 프로세스가 공유 자원의 가용성을 확인하기 위해 락 변수를 무한 반복으로 확인.

5.3 세마포어

- **세마포어(semaphore) 정의**
  - 상호 배제를 운영체제와 프로그래밍 언어 수준에서 지원하는 메커니즘
  - 블록(수면)과 깨움을 지원
  - 세마포어는 정수 값을 갖는 변수로 다음 3가지 인터페이스를 통해 접근할 수 있다.
    - 초기화 연산(Initialize operation): 세마포어 값을 음이 아닌 값으로 초기화한다.
    - 대기 연산(Wait operation): 세마포어 값을 감소시킨다. 값이 음수이면 호출한 프로세스는 블록 된다. 음수가 아니면 프로세스는 계속 수행될 수 있다.
    - 시그널 연산(Signal operation): 세마포어 값을 증가시킨다. 값이 양수가 아니면 블록된 프로세스를 깨운다.

카운팅(counting) 세마포어

- **여러 개의 공유 자원에 대한 액세스를 제어할 목적**
  - 일반 세마포어에 해당

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* 요청한 프로세스를 s.queue에 연결
          /* 요청한 프로세스를 블록 상태로 전이시킴 */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* s.queue에 연결되어 있는 프로세스를 큐에서 제거 */;
          /* 프로세스의 상태를 실행 가능으로 전이시키고 ready list에 연결 */;
        }
}
```

☞ P, V operations
Proberen: to try
Verhogen: to increase

그림 5.3 세마포어 프리미티브

이진(binary) 세마포어

- **이진 세마포어**
  - mutex의 역할

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWaitB(binary_semaphore s)
{
        if (s.value == one)
             s.value = zero;
        else {
                     /* 요청한 프로세스를 s.queue에 연결 */;
                     /* 요청한 프로세스를 블록 상태로 전이시킴 */;
        }
}
void semSignalB(semaphore s)
{
        if (s.queue is empty())
             s.value = one;
        else {
                     /* s.queue에서 프로세스 P를 제거 */;
                     /* 프로세스 P의 상태를 실행 가능으로 전이시키고 ready list에 연결 */;
        }
}
```

그림 5.4 이진 세마포어 프리미티브

세마포어

- 세마포어를 이용한 상호배제

```
/* 상호 배제 예제 프로그램 */
const int n = /* 프로세스 개수 */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* 임계영역 */;
          semSignal(s);
          /* 임계영역 이후 코드 */;
     }
}
void main()
{
     parbegin (P(1), P(2),…, P(n));
}
```

그림 5.6 세마포어를 이용한 상호 배제

세마포어

- **세마포어를 이용한 상호배제 동작 예 1**
  - 가정: 3개의 프로세스 존재

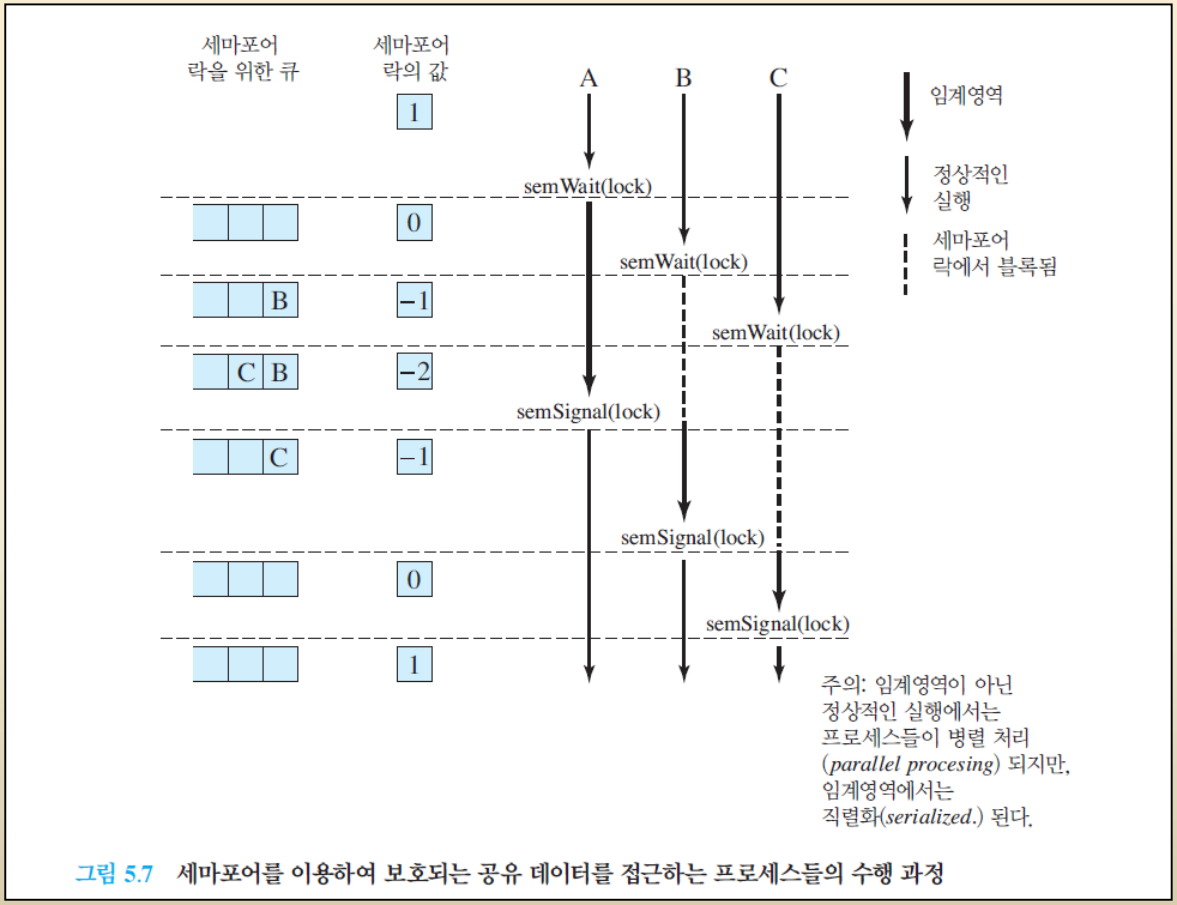


그림 5.7 세마포어를 이용하여 보호되는 공유 데이터를 접근하는 프로세스들의 수행 과정

☞ 세마포어 변수의 값과 블록된 프로세스 개수와의 관계는?

세마포어

- **세마포어를 이용한 상호배제 동작 예 2**
  - 가정: 프로세스 A,B,C는 프로세스 D가 생산한 데이터를 소비
- **strong semaphore vs. weak semaphore**

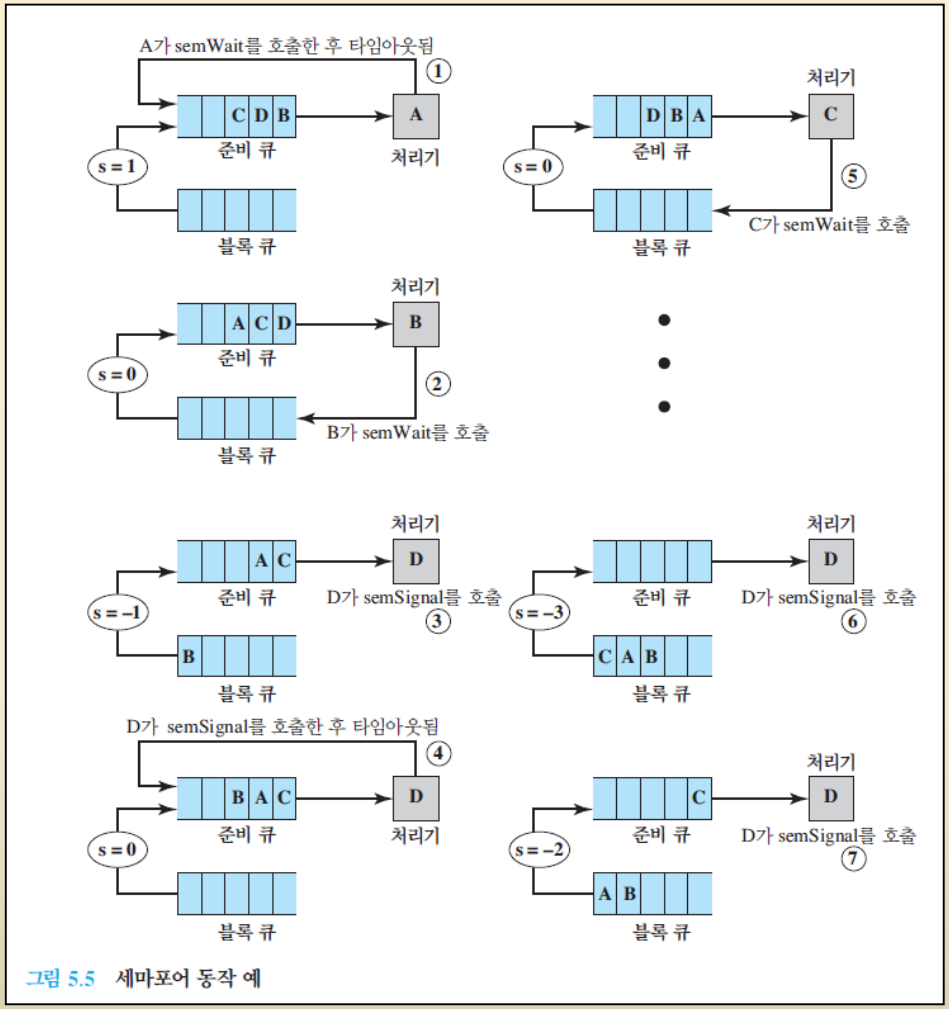


그림 5.5 세마포어 동작 예

세마포어

- **세마포어의 특징**

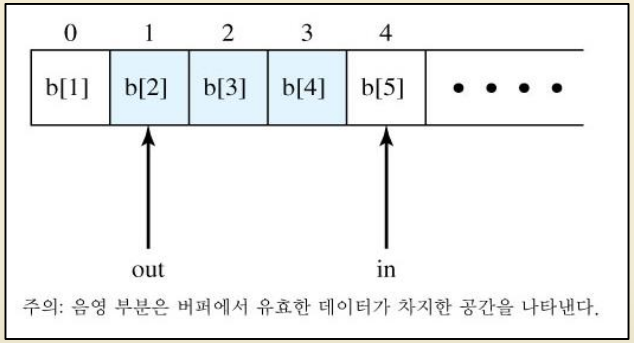
프로세스가 블록될지 여부를 세마포어를 감소시키기 전까지 알 수 없다.

프로세스가 세마포어를 증가시키고 블록되어 있던 프로세스를 깨우면, 이 두 프로세스 모두 수행가능 상태가 된다. 단일처리기 시스템에서 이 두 프로세스 중에 누가 먼저 수행될 지 알 수 없다.

세마포어에 시그널을 보낼 때, 다른 프로세스가 대기 중인지 알 필요가 없다. 즉, 깨어나는 프로세스는 없거나 하나이거나 이다.

생산자/소비자 문제

- **생산자/소비자 (producer/consumer) 문제 정의**
  - 병행 수행되는 생산자와 소비자, 생산된 item을 버퍼에 저장
    - 한 순간에 하나의 생산자 또는 소비자만 버퍼에 접근 가능
    - 생산자는 가득 찬 버퍼에 저장하면 안됨.
    - 또한 소비자는 빈 버퍼에서 꺼내면 안됨
  - 버전 1: 무한 공유 버퍼




```
// 생산자 의사 코드
producer:
  while (true) {
        /* produce item v */
        b[in] = v;
        in++;
  }
```

```
// 소비자 의사 코드
consumer:
  while (true) {
        while (in <= out)
              /*do  nothing */;
        w = b[out];
        out++;
        /* consume item w */
  }
```

생산자/소비자 문제

- 이진 세마포어를 이용한 방법: 부정확한 방법

```
/* 생산자 소비자 프로그램 */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true)  {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void consumer()
      {
            semWaitB(delay);
            while (true)   {
                  semWaitB(s);
                  take();
                  n--;
                  semSignalB(s);
                  consume();
                  if (n==0) semWaitB(delay);
            }
      }
      void main()
      {
            n = 0;
            parbegin (producer, consumer);
      }
```

그림 5.9 무한 버퍼에서 이진 세마포어를 이용한 생산자 소비자 문제 해결 방법: 부정확한 버전

생산자/소비자 문제

- 이진 세마포어를 이용한 방법: 부정확한 방법

표 5.4 그림5.9 프로그램의 수행 예

	생산자	소비자	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–	0
21		semSignalB(s)	1	–	

주의: 표에서 흰색 영역은 세마포어에 의해 보호되는 임계영역을 나타낸다.

생산자/소비자 문제

- 이진 세마포어를 이용한 방법: 정확한 방법

```
/* 생산자 소비자 프로그램 */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true)  {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void consumer()
      {
            int m; /* a local variable */
            semWaitB(delay);
            while (true)   {
                  semWaitB(s);
                  take();
                  n--;
                  m = n;
                  semSignalB(s);
                  consume();
                  if (m==0) semWaitB(delay);
            }
      }
      void main()
      {
            n = 0;
            parbegin (producer, consumer);
      }
```

그림 5.10 무한 버퍼에서 이진 세마포어를 이용한 생산자 소비자 문제 해결 방법: 정확한 버전

생산자/소비자 문제

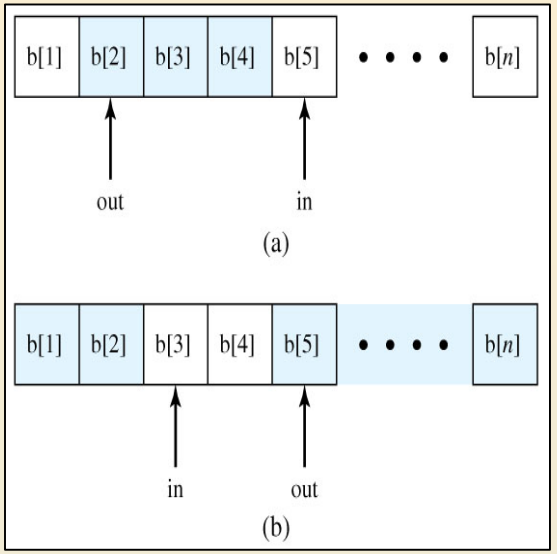
- 무한 버퍼에서 범용 세마포어를 이용한 해결 방법

```
/* 생산자 소비자 프로그램 */
      semaphore n = 0, s = 1;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  consume();
            }
      }
      void main()
      {
            parbegin (producer, consumer);
      }
```

그림 5.11 무한 버퍼에서 범용 세마포어를 이용한 생산자 소비자 문제 해결 방법

생산자/소비자 문제

- **버전 2**
  - 병행 수행되는 생산자와 소비자
  - 유한 공유 버퍼




```
// 생산자 의사 코드
producer:
  while (true) {
              /* produce item v */
              while ((in + 1) % n == out)
                    /* do nothing */;
              b[in] = v;
              in = (in + 1) % n
  }
```

```
// 소비자 의사 코드
consumer:
  while (true) {
              while (in == out)
                    /* do nothing */;
              w = b[out];
              out = (out + 1) % n;
              /* consume item w */
  }
```

생산자/소비자 문제

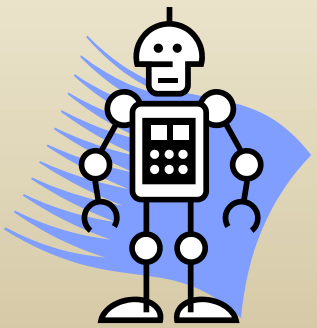
- 유한 버퍼에서 범용 세마포어를 이용한 해결 방법

```
/* 유한 버퍼를 사용하는 생산자 소비자 프로그램 */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(e);
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  semSignal(e);
                  consume();
            }
      }
      void main()
      {
                  parbegin (producer, consumer);
      }
```

그림 5.13 유한 버퍼에서 범용 세마포어를 이용한 생산자 소비자 문제 해결 방법

세마포어 구현

- semWait와 semSignal은 원자적으로 구현되어야 함
- 하드웨어 또는 펌웨어로 구현 가능
- Dekker's 또는 Peterson's 알고리즘 같은 소프트웨어적인 기법으로도 구현 가능
- 상호 배제를 위해 하드웨어 지원 기법 중에 하나를 사용



세마포어 구현

- **세마포어 구현 예**

☞ 원자성(atomicity)

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* 대기 */;
    s.count--;
    if (s.count < 0) {
        /* 요청한 프로세스를 s.queue에 연결*/;
        /* 요청한 프로세스를 블록 상태로 전이시킴 (또한
s.flag를 0으로 설정) */;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* 대기 */;
    s.count++;
    if (s.count<= 0) {
        /* s.queue에 블록된 프로세스를 큐에서 제거 */;
        /* 전이시키고 준비 큐에 연결 */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap 명령 이용

```
semWait(s)
{
    인터럽트 금지;
    s.count--;
    if (s.count < 0) {
        /* 요청한 프로세스를 s.queue에 연결 */;
        /* 요청한 프로세스를 블록 상태로 전이시킴
(또한 인터럽트 허용)*/;
    }
    else
        인터럽트 허용;;
}

semSignal(s)
{
    인터럽트 금지;
    s.count++;
    if (s.count<= 0) {
        /* s.queue에 블록된 프로세스를 큐에서 제거 */;
        /* 전이시키고 준비 큐에 연결 */;
    }
    인터럽트 허용;
}
```

(b) 인터럽트 금지 이용

그림 5.14 세마포어를 구현한 2가지 예

5.4 모니터

- **모니터 (Monitor)의 정의**
  - 상호배제를 위한 소프트웨어 모듈 (프로그래밍 언어 수준에서 제공)
    - Concurrent-Pascal, Pascal-Plus, Module-2/3, Java 등에서 지원
  - 세마포어처럼 상호 배제 기능 제공, but 사용이 훨씬 쉽다.

- **특징**
  - 지역 변수는 모니터 내부에서만 접근 가능
  - 프로세스는 모니터 프로시저 중 하나를 호출함으로써 모니터 내부로 진입
  - 한 시점에 단 하나의 프로세스만 모니터 내부에서 수행 가능

신호 기반 모니터

- **구조**
  - 하나 또는 그 이상의 프로시저
  - 지연변수
  - 조건변수
- **동기화를 위해 조건 변수 (Condition variables) 사용**
  - 모니터 내부에서 사용
  - 다음 두 함수로 접근
    - cwait(c): 호출한 프로세스를 조건 c에서 일시 중지 시킨다.
    - csignal(c): cwait에 의해서 중지되었던 프로세스를 재개시킨다.

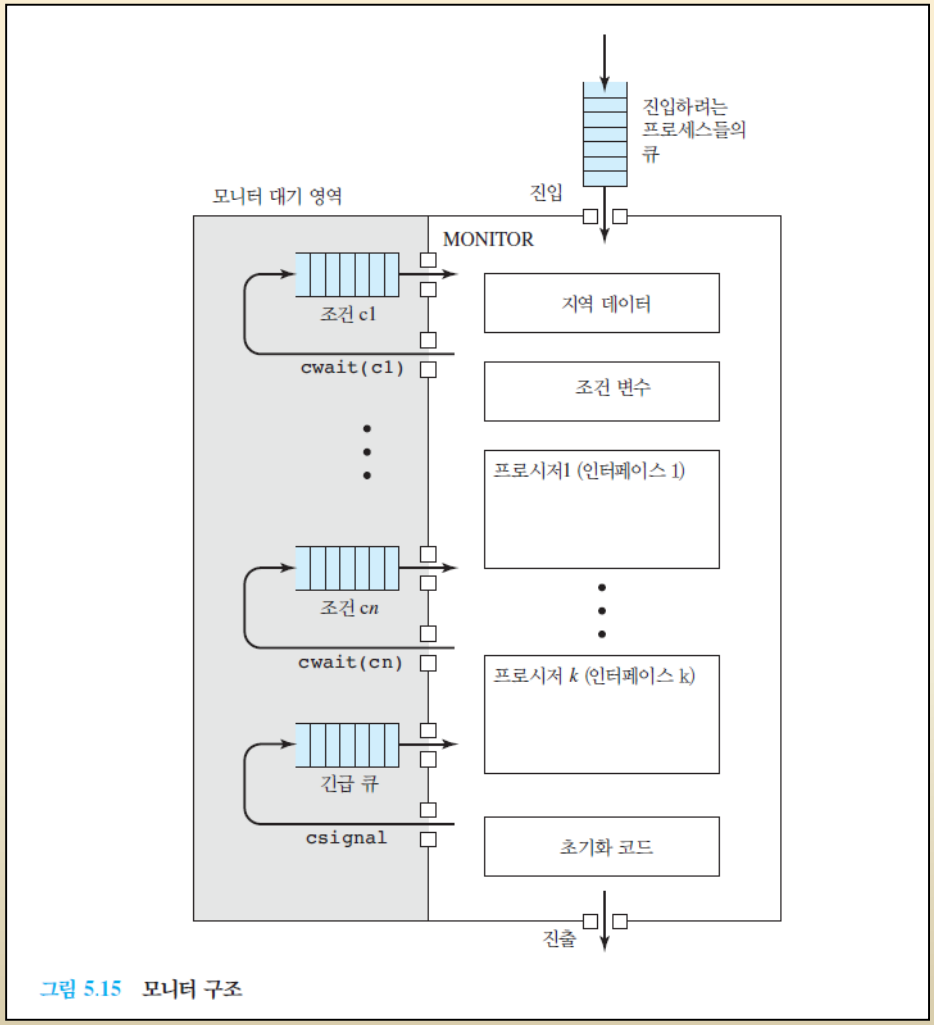


그림 5.15 모니터 구조

신호 기반 모니터

- 모니터를 이용한 생산자/소비자 문제 해결 방법

```
/* 생산자 소비자 프로그램 */
monitor boundedbuffer;
char buffer [N];                                    /* N개의 문자가 저장될 수 있는 버퍼 */
int nextin, nextout;                                /* 버퍼 포인터 */
int count;                                          /* 버퍼 내부에 추가된 문자 개수 */
cond notfull, notempty;                             /* 동기화를 위한 조건 변수 */
void append (char x)

{
      if (count == N) cwait(notfull);               /* 버퍼에 문자가 가득 찬 경우. 오버플로우 방지 */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* 버퍼에 문자 하나를 추가 */
      csignal (notempty);                           /*notempty 조건 변수에서 대기하고 있는 프로세스를 깨움 */
}
void take (char x)
{
      if (count == 0) cwait(notempty);              /* 버퍼가 빈 경우. 언더플로우 방지 */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                      /* 버퍼에서 문자 하나를 삭제 */
      csignal (notfull);                            /* notfull 조건 변수에서 대기하고 있는 프로세스를 깨움 */
}
{                                                   /* 모니터 몸체(body) */
      nextin = 0; nextout = 0; count = 0;           /* 버퍼 변수 초기화 */
}
```

신호 기반 모니터

- 모니터를 이용한 생산자/소비자 문제 해결 방법(계속)

```
void producer()
{
        char x;
        while (true) {
        produce(x);
        append(x);
        }
}
void consumer()
{
        char x;
        while (true) {
        take(x);
        consume(x);
        }
}
void main()
{
        parbegin (producer, consumer);
}
```

그림 5.16 유한 버퍼에서 모니터를 이용한 생산자 소비자 문제 해결 방법

Mesa 모니터

- **변경 사항**
  - csignal() → cnotify()
  - if → while

```
void append (char x)
{
      while (count == N) cwait(notfull);      /* 버퍼에 문자가 가득 찬 경우. 오버플로우 방지 */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;                                                  /* 버퍼에 문자 하나를 추가 */
      cnotify(notempty);                      /* 대기하는 소비자가 있으면 조건 발생을 통보 */
}

void take (char x)
{
      while (count == 0) cwait(notempty);       /* 버퍼가 비어 있는 경우. 언더플로우 방지 */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                                  /* 버퍼에서 문자 하나를 삭제 */
      cnotify(notfull);                       /* 대기하는 생산자가 있으면 조건 발생을 통보 */
}
```

그림 5.17 Mesa 모니터를 이용한 유한 버퍼 모니터 코드

5.5 메시지 전달

- **메시지 전달(message passing) 인터페이스**
  - send (destination, message)
  - receive (source, message)
  - 기본적으로 정보 교환을 위해 사용
  - 또한 상호배제와 동기화를 위해 사용 가능

☞ mailbox (port)

☞ blocking operation

표 5.5 프로세스 간 통신과 동기화를 위한메세지 전달 시스템에서 설계 특성

동기화 송신 블록킹 비블록킹 수신 블록킹 비블록킹 도착 여부 확인	**포맷** 내용 길이 고정 가변
주소 지정 직접 송신 수신 명시적 암묵적 간접 정적 동적 소유관계	**큐잉 방식** 선입선출(FIFO) 우선순위

메시지 전달

- **관련 용어**
  - Blocking, nonblocking
  - Addressing
    - Direct, indirect
  - Message format
  - Queuing discipline

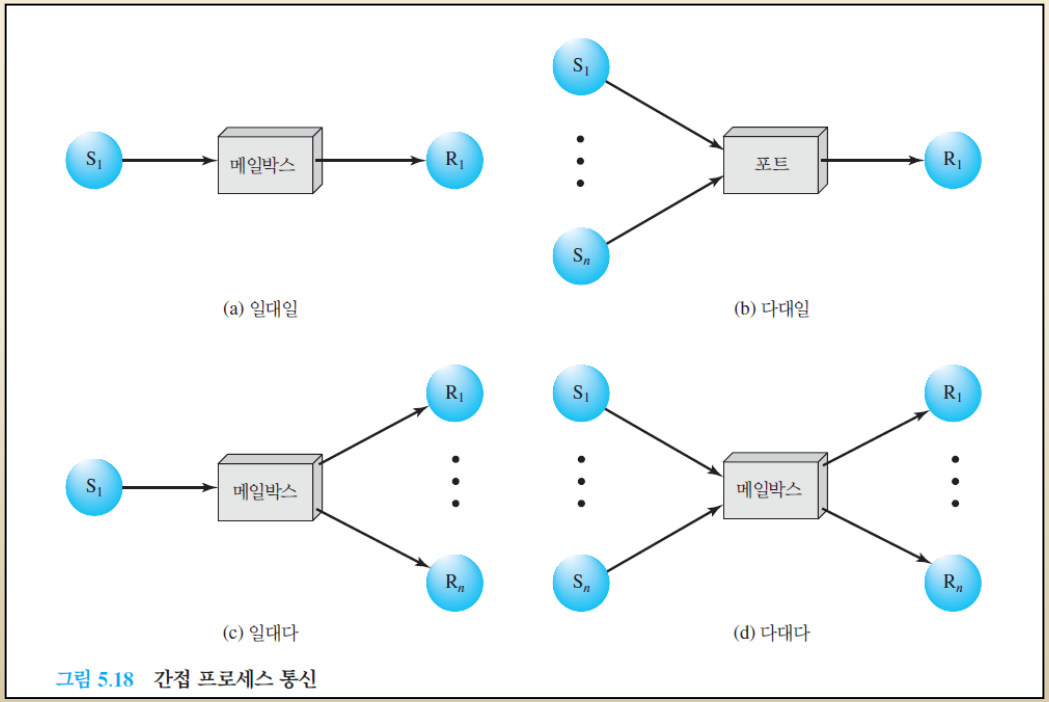
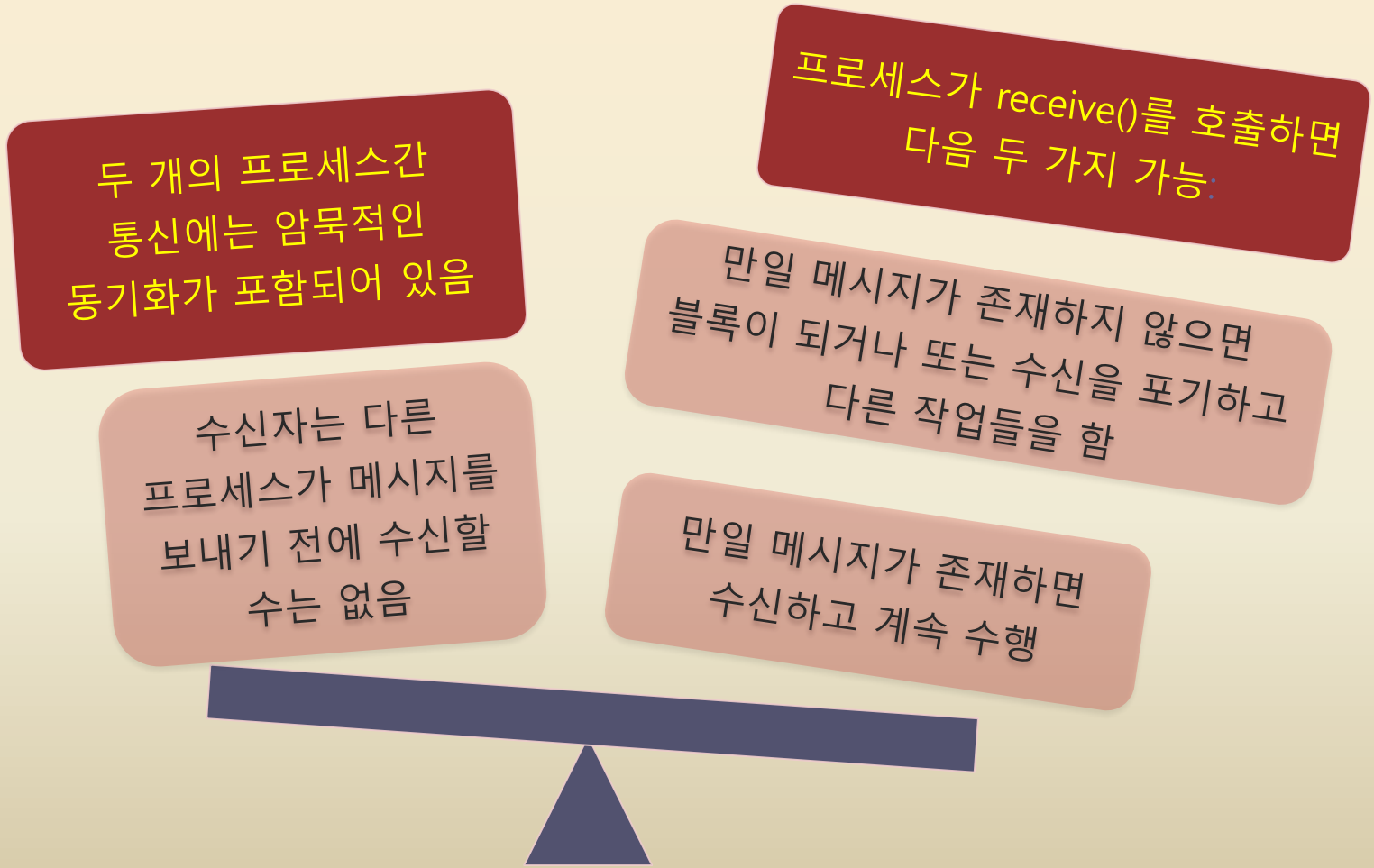


그림 5.18 간접 프로세스 통신

메시지 전달

- 동기화


두 개의 프로세스간 통신에는 암묵적인 동기화가 포함되어 있음
프로세스가 receive()를 호출하면 다음 두 가지 가능:
만일 메시지가 존재하지 않으면 블록이 되거나 또는 수신을 포기하고 다른 작업들을 함
수신자는 다른 프로세스가 메시지를 보내기 전에 수신할 수는 없음
만일 메시지가 존재하면 수신하고 계속 수행

메시지 전달

- 메시지 전달을 이용한 상호 배제

```
/* 상호 배제 프로그램 */
const int n = /* 프로세스 개수 */
void P(int i)
{
      message msg;
      while (true) {
          receive (box, msg);
          /* 임계영역 */;
          send (box, msg);
          /* 임계영역 이후 코드 */;
      }
}
void main()
{
      create_mailbox (box);
      send (box, null);
      parbegin (P(1), P(2),…, P(n));
}
```

그림 5.20 메시지를 이용한 상호 배제

메시지 전달

- 메시지 전달을 이용한 생산자/소비자 문제 해결 방법

```
const int
    capacity = /* 버퍼 용량 */ ;
    null = /* 빈 메시지 */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

그림 5.21 유한 버퍼에서 메시지를 이용한 생산자/소비자 문제 해결 방법

5.6 판독자/기록자 문제

- **판독자/기록자(readers/writers) 문제 정의**
  - 병행 수행되는 판독자와 기록자
  - 공유 자원 (파일, 데이터베이스)

- **요구 조건**
  - 여러 판독자들이 공유 데이터를 동시에 읽을 수 있다.
  - 한 시점에 오직 하나의 기록자만 공유 데이터를 변경할 수 있다.
  - 기록자가 데이터를 변경하고 있는 동안 판독자가 그 데이터를 읽을 수 없다.

☞ 생산자/소비자 문제와 차이점은?

판독자/기록자 문제

- 세마포어를 이용한 해결 방법: 판독자 우선

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

판독자/기록자 문제

- **세마포어를 이용한 해결 방법: 기록자 우선**
  - x: readcount 보호
  - rsem: writer가 먼저 들어가 있을 때 첫 번째 reader가 기다리는 세마포어
  - z: writer가 먼저 들어가 있을 때 두 번째 이후의 reader가 기다리는 세마포어
  - y: writecount 보호
  - wsem: reader건 writer건 먼저 들어온 프로세스가 갖는 세마포어

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

판독자/기록자 문제

- **세마포어를 이용한 해결 방법: 기록자 우선**
  - 프로세스 대기 상태

표 5.6 그림 5.23의 프로그램이 수행할 때 프로세스 대기 상태

시스템에 판독자들만 존재	• *wsem*: 판독자가 소유 • 대기하는 프로세스 없음
시스템에 기록자들만 존재	• *wsem*, *rsem*: 기록자가 소유 • *wsem*: 다른 기록자들이 여기에서 대기
시스템에 판독자/기록자 모두 존재 (판독자가 공유 데이터 접근)	• *wsem*: 판독자가 소유 • *rsem*: 기록자가 소유 • *wsem*: 모든 기록자들이 여기에서 대기 • *rsem*: 하나의 판독자가 여기에서 대기 • *z*: 다른 판독자들이 여기에서 대기
시스템에 판독자/기록자 모두 존재 (기록자가 공유 데이터 접근)	• *wsem*: 기록자가 소유 • *rsem*: 기록자가 소유 • *wsem*: 다른 기록자들이 여기에서 대기 • *rsem*: 하나의 판독자가 여기에서 대기 • *z*: 다른 판독자들이 여기에서 대기

메시지 전달을 이용한 판독자/기록자 문제

- **기록자 우선이라고 가정**
  - 기록자나 판독자는 제어기에게 요청 메시지를 보내고
  - 제어기로부터 OK를 받으면 임계영역으로 진입
  - 빠져나올 때 finished 메시지를 보내 나왔음을 알림.
  - 제어기는 쓰기 요청 메시지를 먼저 처리
- **count: 상호배제 구현에 활용되는 변수**
  - 최대 가능한 판독자의 수보다 큰 수로 초기화되어 있음
  - 다음의 예에서는 100으로 초기화 되어 있음
  - count > 0: 현재 대기 중인 writer는 없는 상태
  - count = 100: 현재 활동 중인 reader가 없는 상태
  - 0 < count < 100: 현재 활동 중인 reader가 있는 상태
  - count < 0: writer가 요청을 했고, 활동 중인 모든 reader가 수행을 마칠 때까지 기다리는 중. 따라서 reader의 완료 메시지만 서비스.
  - count = 0: 대기하던 writer 요청을 재개시킬 수 있는 상태

메시지 전달을 이용한 판독자/기록자 문제

```
void reader(int i)

    message rmsg;
        while (true) {
            rmsg = i;
            send (readrequest, rmsg);
            receive (mbox[i], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
}
void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()

    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

그림 5.24 메시지 전달을 이용한 판독자/기록자 문제 해결 방법