Lecture Note #9 - Chapter 26~33. Concurrency 등

- 시스템 구조에 따른 프로세스 관리의 구분

- 다중프로그래밍(Multiprogramming)
  - ✓ 단일 처리기 상에서 다수의 프로세스 관리

- 멀티프로세싱(Multiprocessing)
  - ✓ 다중처리기 상에서 다수의 프로세스 관리

- 분산처리(Distributed processing)
  - ✓ 다수의 분산된 컴퓨터 시스템들 상에서 수행되는 다수의 프로세스 관리
  - ✓ 최근 급격히 확산되고 있는 클러스터(cluster) 시스템이 대표적인 예

- 병행성

- 프로세스 관리에서 핵심적인 주제

- 프로세스의 병행 처리(concurrent processing), 프로세스 간 통신, 자원에 대한 공유 및 경쟁, 프로세스 활동들의 동기화, 프로세스에 대한 처리기 시간 할당 등 다양한 이슈들을 포함

- 다중처리기와 분산 시스템과 같이 다수의 처리기가 있는 시스템 뿐만 아니라 단일 처리기 다중프로그래밍 시스템에서도 발생되는 이슈

- 병행성의 발생 상황

- 다수의 응용
  - ✓ 다수의 활동 중인 응용들이 처리 시간을 동적으로 공유할 수 있도록 다중프로그래밍이 개발

- 구조화된 응용
  - ✓ 모듈화된 설계 원칙과 구조적인 프로그래밍의 확장 결과, 일부 응용들은 병행 프로세스들의 집합으로 구현

- 운영체제 구조
  - ✓ 시스템 프로그래머도 구조적인 프로그래밍 기법을 사용하며, 그 결과 운영체제도 다수의 프로세스나 쓰레드들로 구현

- 병행성과 관련 있는 주요 용어

임계영역(critical section)	공유 자원에 접근하는 각 프로세스 내부의 코드 영역으로, 다른 프로세스가 해당 코드 영역을 수행하는 동안 수행하지 않아야 하는 코드 영역.
교착상태(deadlock)	두 개 이상의 프로세스들이 서로 다른 프로세스가 어떤 일을 해 줄 때까지 기다리고 있어, 관련 프로세스 모두가 더 이상 진행을 할 수 없는 상태.
라이브락 (livelock)	두 개 이상의 프로세스들이 다른 프로세스의 상태 변화에 따라 자신의 상태를 변화시키는 작업만 수행하고 있는 상태. 각 프로세스들이 열심히 작업하고 있지만, 수행되는 작업은 유용한 것이 아닌 반복적인 상태 변화일 뿐임
상호배제(mutual exclusion)	한 프로세스가 공유 자원에 접근하는 임계영역 코드를 수행하고 있을 경우, 다른 프로세스들은 해당 자원에 접근하는 각자의 임계영역 코드를 수행할 수 없다는 조건.
경쟁조건(race condition)	두 개 이상의 프로세스가 공유 데이터 항목을 읽고 쓰는 상황으로, 최종 수행 결과는 프로세스들의 상대적인 수행 순서에 따라 달라질 수 있음
기아(starvation)	수행 가능한 프로세스가 스케줄러에 의해 무한정 간과되고 있는 상황(수행 가능하지만 선택되지 않는 상황)

가. 병행성의 원리

- 시스템에서 프로세스들은 병행 처리됨 단일 처리기 시스템에서는 프로세스들이 인터리빙(*interleaving*), 즉 여러 프로세스가 서로 번갈아 수행

- 이때 번갈아 수행되는 시간 간격이 매우 짧기 때문에 사용자에게는 프로세스들이 마치 동시에 수행되는 것 같은 효과를 제공

- 인터리빙은, 비록 실제로 병렬 처리되는 것도 아니고 프로세스들 간의 문맥 교환 비용도 발생하지만, 처리 효율과 구조적 프로그래밍에 유익함

- 다중처리기 시스템에서는 프로세스들의 인터리빙뿐만 아니라 오버래핑(overlapping)도 지원

✓ 여러 처리기 상에서 프로세스들이 실제로 병렬 수행되는 것이다.

• 인터리빙과 오버래핑의 문제점

- 단일 처리기 시스템의 경우 프로세스들의 상대적 수행 속도를 예측할 수 없다는, 다중프로그래밍의 기본 특성에서 다음과 같은 문제 발생

✓ 전역 자원의 공유의 어려움

· 예를 들어, 두 개의 프로세스가 같은 전역 변수를 사용하는 경우

· 한 프로세스가 전역 변수에 데이터를 쓰고 다른 프로세스가 그것을 읽는다면, 어떤 순서로 읽기 및 쓰기가 진행되는지에 따라 수행 결과가 달라짐

✓ 운영체제가 자원을 최적으로 할당하기 어려움

· 예를 들어 프로세스 A가 특정 입출력 채널을 요청하여 운영체제가 할당해 주었을 경우

· 그런데 프로세스 A가 그 채널을 사용하기도 전에 보류될 수 있으며, 이 경우 다른 프로세스들은 해당 채널을 사용할 수 없게 됨

· 결국 가용한 자원이 프로세스의 수행 순서에 따라 사용될 수 없는 상황이 발생 => 교착 상태 (6장)

✓ 프로그래밍 오류를 찾아내는 것이 어려워짐

· 어떤 프로그래밍 오류는 특정 수행 순서에서 나타나는데, 프로세스의 수행 순서가 바뀔 경우 그 오류는 나타나지 않을 수도 있음

· 이렇게 간헐적으로 재연되는 오류는 디버깅을 어렵게 함

• 병행 처리의 예

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

- 이 프로시저는 키보드로부터 한 번에 한 문자씩 입력받아 해당 문자를 모니터로 출력하는 프로그램의 기본 요소를 보여 줌

- 입력된 문자는 변수 chin에 저장되고, 변수 chout에 치환된 후, 스크린으로 보내짐

- 어떤 프로그램도 이 프로시저를 호출하여 사용자 입력을 사용자 스크린에 출력할 수 있음

- 단일 사용자를 지원하는 단일 처리기 다중프로그래밍 시스템의 경우

✓ 사용자는 한 응용에서 다른 응용으로 옮겨 다닐 수 있으며, 이들 응용이 동일 키보드와 동일 스크린을 사용한다고 가정

✓ 각 응용이 프로시저 echo를 필요로 하므로, 이를 모든 응용이 접근할 수 있는 메모리 영역에 적재하여 공유 프로시저로 설정 가능

· 이는 단지 하나의 복사본만을 사용할 수 있게 하여 공간 절약의 유익을 제공

- 공유의 문제점

✓ 1. 프로세스 P1이 프로시저 echo를 호출. echo는 getchar을 호출하여 사용자가 입력한 문자 'x'를 리턴하고, 이는 변수 chin에 저장. 이때 P1이 인터럽트 당해 preemptive 됨.

✓ 2. 프로세스 P2가 활성화되어, 프로시저 echo를 호출. 사용자가 'y'를 입력하여 변수 chin에 문자 'y'가 저장되고, 화면에 'y'가 출력.

✓ 3. 얼마 후 프로세스 P1의 수행이 다시 재개되지만, 이전에 자신이 변수 chin에 저장했었던 값 'x'는 이미 P2에 의해 'y'로 덮어 씌어 진 이후임. 결과적으로 변수 chin에 저장된 값 'y'가 변수 chout에 치환된 후 스크린에 출력됨.

- 결국 첫 번째 문자 'x'는 유실되고, 두 번째 문자 'y'는 두 번 출력. 이 문제는 전역 변수 chin을 두 개 이상의 프로세스가 공유하기 때문에 발생 => 한 프로세스가 전역 변수를 갱신한 후 일시 중지되고, 이때 다른 프로세스가 같은 전역 변수를 접근하여 수정하면 결국 기존 프로세스가 갱신했던 값을 잃게 됨

- 해결 안 => 만일 한 시점에 오직 한 프로세스만이 그 프로시저 내부에 존재할 수 있다고 제한

✓ 1. 프로세스 P1이 프로시저 echo를 호출한다. echo는 getchar을 호출하여 사용자가 입력한 문자 'x'를 리턴하고, 이는 변수

chin에 저장. 이때 P1이 인터럽트 당해 preemptive됨.

✓ 2. 프로세스 P2가 활성화되어, 프로시저 echo를 호출. 그러나 P1이 보류된 상태일지라도 아직 echo 안에 있기 때문에, P2는 프로시저 echo에 진입 기능할 때까지 대기하며 보류됨.

✓ 3. 얼마 후, P1의 수행이 재개되고 echo의 수행이 완료. 그럼 변수 chin에 저장되었던 문자 'x'가 출력됨.

✓ 4. P1이 프로시저 echo에서 빠져나왔을 때, P2는 보류 상태에서 깨어남. 이후 P2의 수행이 재개되고, 프로시저 echo는 성공적으로 호출됨.

- 한 번에 하나의 프로세스만 프로시저 echo에 진입할 수 있고, 그 수행이 끝나 echo를 빠져 나간 후에야 다른 프로세스가 echo에 진입할 수 있도록 원칙을 세워 강제한다면 문제가 발생하지 않음

- 병행성 문제가 단일 처리기에서도 발생할 수 있음을 알 수 있음

- 다중처리기 시스템의 경우에도 공유 자원의 보호라는 동일한 문제가 발생

- 공유되는 전역 변수에 대한 접근 제어가 없는 경우의 예

✓ 1. 두 개의 프로세스 P1과 P2가 각각 서로 다른 처리기에서 동시에 수행되고 있음. P1과 P2는 모두 프로시저 echo를 호출함.

✓ 2. 다음 사건들이 순서대로 발생함. 같은 줄에 기술된 사건들은 병렬적으로 수행되는 것을 의미함.

Process P1	Process P2
.	.
chin = getchar();	.
.	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);
.	.

✓ 수행 결과 프로세스 P1에서 입력받은 문자는 화면에 출력되기 전에 손실되고, 프로세스 P2에서 입력받은 문자가 화면에 두 번 출력됨

✓ 결국 위의 예는 단일 처리기 시스템에서 다중프로그래밍에 의해 발생하였던 문제가 다중처리기 시스템에서도 발생함을 보여주는 것임.

✓ 결국, 전역 변수에 대한 접근 제어, 즉 특정 시점에 오직 하나의 프로세스만 프로시저 echo 내부에 존재할 수 있다는 원칙을 적용하면 다음과 같이 수행됨

· 1. 두 개의 프로세스 P1과 P2가 각각 서로 다른 처리기에서 동시에 수행되고 있음. P1이 프로시저 echo를 호출함.

· 2. P1이 프로시저 echo 내부에서 수행하는 동안, P2가 프로시저 echo를 호출함. 하지만 P1이 echo 내부에 존재(P1은 수행중인 상태일 수도 있고 보류 상태일 수도 있음)하기 때문에, P1은 프로시저 echo로 진입하지 못하고 블록됨. 결국 P2는 프로시저 echo가 사용 가능해질 때까지 대기하며 보류되는 것임.

· 3. 얼마 후, 프로세스 P1은 프로시저 echo의 수행을 완료하고 빠져 나옴. P1이 프로시저 echo를 빠져 나오자마자 P2는 블록 상태에서 깨어나며, 다시 프로시저 echo의 수행을 시도함.

- 단일 처리기 시스템의 경우 문제 발생 이유

✓ 문제가 발생한 이유는 인터럽트가 프로세스의 어떤 곳에서도 명령 수행을 중지시킬 수 있다는 것임

- 다중처리기 시스템의 경우, 문제가 발생한 이유

✓ 인터럽트의 비동기적인 발생뿐만 아니라 두 개의 프로세스들이 동시에 수행되면서 같은 전역 변수를 접근하려 하 기 때문임

· 해결 안 : 공유 자원에 대한 접근을 제어하는 것임

• 경쟁조건

- 다수의 프로세스나 쓰레드가 공유 자원을 동시에 읽거나 쓰려고 하는 상황

- 경쟁조건이 발생하면 최종 수행 결과는 프로세스들의 수행 순서에 따라 달라짐

✓ 첫 번째 예에서는 두 개의 프로세스 P1과 P2가 전역변수 a를 공유함

· 수행 중 P1은 전역변수 a를 값 1로 수정하며, P2는 전역변수 a를 값 2로 수정함

· 결국 두 프로세스 P1과 P2가 전역 변수 a를 접근하기 위해 서로 경쟁하고 있는 상태이며, 이 경쟁에서 진 프로세스, 즉 마지막으로 변수에 접근한 프로세스의 수정 결과가 전역 변수 a의 최종 저장 값이 됨

· 구체적으로 만일 P2의 수정 연산이 더 늦게 수행되었다면 전역변수 a를 값 2를 갖게 됨

✓ 두 번째 예제에서는 b = 1, c = 2로 초기화 된 두 전역 변수를 공유하는 두 프로세스 P3과 P4를 고려할 경우

· 수행 중 P3은 b = b + c 연산을 수행한다. 반면 P4는 c = b + c 연산을 수행함

· 이 예에서도 수행 결과는 P3과 P4의 수행 순서에 따라 결정됨

· 만일 P3이 연산을 먼저 수행하게 되면 최종 수행 결과는 b = 3, c = 5가 되고, P4가 연산을 먼저 수행하게 되면 최종 수행 결과는 b = 4, c = 3이 됨.

• 운영체제의 고려 사항들

- 모든 프로세스들을 추적할 수 있어야 함

✓ 프로세스 제어블록(PCB: Process Control Block)을 이용하여 구현

- 활동 중인 각 프로세스에게 다양한 자원들을 할당하고 해제할 수 있어야 함

✓ 처리기 시간

✓ 메모리

✓ 파일

✓ 입출력 장치

- 각 프로세스에게 할당된 자원과 데이터들을 다른 프로세스들의 예기치 못한 간섭으로부터 보호해야 함

- 프로세스의 기능과 수행 결과는 그 프로세스의 상대적 수행 속도(동시에 수행되는 다른 프로세스들과의 상대 속도)에 대해 독립적이어야 함

• 자원에 대한 프로세스 간 경쟁

- 병행 프로세스가 동일한 자원을 동시에 접근하려는 경우 경쟁이 발생

✓ 두 개 이상의 프로세스들이 수행 중에 특정 자원을 사용하려 함.

✓ 각 프로세스는 다른 프로세스의 존재를 모름.

✓ 또한 각 프로세스는 다른 프로세스의 수행 결과에 영향을 받지 않음

✓ 따라서 각 프로세스는 사용한 자원의 상태를 변화시켜서는 안 됨

✓ 이러한 자원에는 입출력 장치, 메모리, 처리기 시간, 클록 등이 있다.

- 경쟁 관계에 있는 프로세스들은 서로 아무런 정보도 교환하지 않음

✓ 하지만, 프로세스의 수행은 경쟁 관계인 다른 프로세스의 행위에 영향을 줄 수 있음

✓ 예를 들어 두 개의 프로세스들이 같은 자원을 접근하려 한다면, 운영체제는 한 프로세스에게만 자원을 할당해 줄 것이고 나머지 하나는 기다릴 수밖에 없음

✓ 따라서 자원을 할당받지 못한 프로세스의 수행 속도는 떨어짐

✓ 극단적인 경우 기다리던 프로세스가 요청했던 자원을 영원히 할당받지 못하여 무한 대기 상태에 빠질 수도 있음

- 경쟁 관계에 있는 프로세스들이 존재하는 경우 상호배제(mutual exclusion), 교착상태(deadlock), 기아(starvation)라는 세 가지 제어 문제가 발생

✓ 임계자원(*critical resource*)

· 두 개 이상의 프로세스가 동시에 사용할 수 없는 자원

✓ 임계영역(*critical section*)

· 프로그램 코드 중 임계 자원에 접근하는 부분

✓ 예) 프린터 자원

· 프린터에 두 개 이상의 프로세스가 동시에 출력하려 한다면, 출력 결과는 프로세스들의 출력 내용들이 함께 인쇄되어 알아볼 수 없는 문서가 됨

· 임계 자원 : 프린터는 두 개 이상의 프로세스가 동시에 사용할 수 없는 임계 자원

· 임계 영역 : 각 프로세스에서 프린터로 명령과 데이터를 보내거나 프린터의 상태를 확인하는 프로그램 코드 부분이 임계영역

- 상호배제

✓ 한 시점에 단 하나의 프로세스만이 임계영역에 들어갈 수 있다는 것을 의미

✓ 예) 프린터 자원의 상호 배제 보장

· 특정 프로세스가 전체 내용을 출력할 때까지 프린터 자원에 대한 독점적인 제어를 유지하는 것

✓ 일반적으로 상호배제는 프로그래머가 직접 작성

✓ 상호배제의 구체적인 내용은 프로그램마다 다름

✓ 운영체제가 구체적인 요구 조건을 명확하게 파악하기 어려움

✓ 운영체제가 각 프로그램의 상호배제를 모두 이해하고 강제해 줄 것이라고 기대할 수는 없음

- 상호배제의 보장은 교착상태와 기아라는 제어 문제를 발생시킴

✓ 교착상태

✓ 기아

· 교착상태는 아니지만 특정 프로세스가 오랜 기간 동안 자원을 사용하지 못하는 상태

• 상호배제를 위한 요구조건

- 상호배제가 강제되어야 함

✓ 동일한 자원이나 공유 객체에 대한 임계영역을 가지고 있는 모든 프로세스들 중에서 어느 한 시점에는 반드시 단 하나의 프로세스만 임계영역에 진입할 수 있음

- 임계영역이 아닌 곳에서 수행이 멈춘 프로세스는 다른 프로세스의 수행을 간섭해서는 안 됨

- 임계영역에 접근하고자 하는 프로세스의 수행이 무한히 미루어져서는 안 됨

✓ 교착상태 및 기아가 일어나지 않아야 함

- 임계영역이 비어 있을 때, 임계영역에 진입하려고 하는 프로세스는 즉시 임계영역에 들어갈 수 있어야 함

- 처리기의 개수나 프로세스의 상대적 수행 속도에 대한 가정은 없어야 함

- 일단 임계영역에 들어간 프로세스는 유한 시간 내에 임계영역에서 나와야 함

• 임계영역 문제

- n processes all competing to use some shared data

- Each process has a code segment, called critical section, in which the shared data is accessed.

- Problem

✓ 한 프로세스가 임계 구역에서 실행되면, 다른 프로세스의 임계 구역에서의 실행이 불가능하는 것을 보장

- Basic Attempt

✓ Only 2 processes, P0 and P1

✓ General structure of process Pi (other process Pj)

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

✓ Processes may share some common variables to synchronize their actions.

- Solution 1

✓ turn varible 사용

P0	P1
`while (turn != 0) do {nothing };` *`critical section;`* `turn = 1;`	`while (turn != 1) do {nothing };` *`critical section;`* `turn = 0;`

✓ Shared variable turn indicates who is allowed to enter next, can enter if turn = me

✓ On exit, point variable to other process

✓ Deadlock if other process never enters

- Solution 2

✓ busy waiting

P0	P1
while flag[1] do {nothing } ; flag[0] = true; *critical section;* flag[0] = false;	while flag[0] do {nothing } ; flag[1] = true; *critical section;* flag[1] = false;

✓ Each process has a flag to indicate it is in the critical section
✓ Fails mutual exclusion if processes are in lockstep

- Solution 3

✓ Busy Waiting Modified

P0	P1
flag[0] = true; while flag[1] do {nothing } ; *critical section;* flag[0] = false;	flag[1] = true; while flag[0] do {nothing } ; *critical section;* flag[1] = false;

✓ Deadlocks if processes are in lockstep

- Solution 4

✓ Busy Flag Again

P0	P1
flag[0] = true; while (flag[1]) { flag[0] = false; delay flag[0] = true; } *critical section;* flag[0] = false;	flag[1] = true; while (flag[0]) { flag[1] = false; delay flag[1] = true; } *critical section;* flag[1] = false;

✓ Livelock if processes are in lockstep
✓ It is possible for each process to set their flag, check other processes, and reset their flags.
✓ This scenario will not last very long so it is not deadlock.
✓ It is undesirable

나. 상호 배제: 하드웨어 지원

• 상호배제를 보장하기 위한 소프트웨어 알고리즘

- Dekker의 알고리즘
- 이러한 소프트웨어적인 접근 방법은 높은 수행 부하와 논리적 오류의 위험이 크다는 단점이 있음
- 소프트웨어적인 접근 방법에 대한 이해는 병행 처리의 기본 개념과 병렬 프로그램을 개발할 때 발생할 수 있는 잠재적인 문제를 파악하는데 큰 도움이 됨.

• 상호배제를 보장하기 위한 하드웨어

- 인터럽트 금지(Interrupt Disable)

✓ 단일 처리기에서 병행 처리되는 프로세스들은 실제로 동시에 수행되는 것이 아니라 번갈아 가며 수행
✓ 또한 프로세스는 운영체제 서비스를 호출하거나 인터럽트 되어질 때까지 계속됨
✓ 결국 인터럽트가 발생하지 않으면 그 동안 한 프로세스의 계속적인 수행을 보장할 수 있음
✓ 그러므로 상호배제를 보장하기 위한 가장 간단한 방법은 프로세스가 인터럽트 되지 않도록 하는 것임
✓ 이를 위해서 시스템 커널에서는 인터럽트를 허용하거나 허용하지 않게 하는 기본적인 인터페이스를 제공

while(true)

```
{
        /* 인터럽트 금지 */;
        /* 임계영역 */;
        /* 인터럽트 허용 */;
        /* 임계영역 이후 코드 */;
}
```

· 임계영역에서는 인터럽트가 발생할 수 없기 때문에 상호배제는 보장됨
· 문제점
  ☞ 부하가 큼. 인터럽트가 불허되면 그 사이에 발생하는 외부 이벤트에 대한 처리와 다른 프로세스에 대한 스케줄링 등 모든 기능이 중지되기 때문에 시스템의 수행 효율이 눈에 띄게 감소될 수 있음
  ☞ 다중처리기 시스템에서는 올바르게 상호 배제를 보장할 수 없음
  ☞ 두 개 이상의 처리기를 가지는 컴퓨터 시스템에서는 인터럽트가 금지된 상황에서도 서로 다른 프로세스가 동시에 공유 자원에 접근할 수 있기 때문임

- 특별한 기계 명령어
✓ 처리기들이 주기억장치를 공유하는 다중처리기 환경의 경우, 처리기들은 주종(master/slave) 관계가 아니라 동등한 관계에서 독립적으로 동작함
✓ 이러한 환경에서 처리기 간의 상호 배제를 보장할 수 있는 인터럽트 기법은 없음
✓ 하드웨어 수준에서는 특정 메모리 주소가 접근되고 있을 때 같은 위치에 대한 다른 접근 요청은 차단됨
✓ 이를 기반으로 처리기 설계자들은 두 개의 기능을 원자적으로 처리하는 다양한 기계어를 제안
· 예) 같은 메모리 위치에 대한 읽기와 쓰기, 또는 같은 메모리 위치에 대한 읽기와 테스트를, 하나의 명령어 반입 사이클을 이용하여 처리하는 기계 명령어를 제안
· 이 기계 명령어가 수행되는 동안 같은 메모리 위치를 접근하려는 다른 명령어들은 블록됨
· 일반적으로 이러한 기능은 하나의 명령어 사이클 내에 처리됨
✓ Test and Set 명령어

```
boolean testset(int i)
{
        if(i == 0) {
                i = 1;
                return true;
        } else {
                return false;
        }
}
```

· 이 명령어는 인자 i의 값을 점검
· 만약 그 값이 0이라면 1로 고친 후, true를 리턴한다. 그렇지 않으면, 인자 i의 값을 그대로 두고, false를 리턴
· testset 함수 전체는 원자적으로 처리됨.
· 즉 수행 도중에 인터럽트 당하지 않음

✓ Exchange 명령어

```
void exchange(int register, int memory)
{
        int temp;
        temp = memory;
        memory = register;
        register = temp;
}
```

· exchange 명령어는 레지스터의 값과 메모리에 저장된 값을 서로 교체하는 기능을 수행
· 예) 인텔 IA-32(Pentium)와 IA-64(Itanium)의 XCHG 명령어

• 기계 명령어 접근 방법의 특성

- 단일 처리기 시스템뿐만 아니라 공유 메모리를 사용하는 다중처리기 시스템에서도 사용 가능함

- 간단하고 검증이 쉬움

- 서로 다른 변수를 사용하면 여러 개의 임계영역을 지원할 수 있음

• 기계 명령어 문제점

- 바쁜 대기를 사용함
  ✓ 따라서 임계영역에 진입하고자 기다리는 동안 프로세스는 처리기 시간을 계속 사용해야 한다.

- 기아상태에 빠질 수 있음
  ✓ 프로세스가 임계영역에서 빠져 나왔을 때 대기하고 있던 프로세스가 여러 개라면, 그 중 한 프로세스만 다시 임계 영역에 진입할 수 있다. 이때 프로세스들의 특성이나 기다린 대기 시간 등을 고려하지 않으므로 무한정 기다리게 되는 프로세스가 생길 수 있다.

- 교착상태에 빠질 수 있음
  ✓ 프로세스 P1은 testset이나 exchange 같은 특별한 명령어를 수행한 후 임계영역에 들어감
  ✓ 이 때 P1보다 더 높은 우선순위를 갖는 프로세스 P2가 생성되었고, 운영체제가 P2를 스케줄함
  ✓ 이때, 만약 P2가 P1과 같은 자원을 쓰려고 시도한다면, 상호배제 조건에 의해 접근이 실패되고, 바쁜 대기를 수행함.
  ✓ 그런데 P1은 다시 스케줄링 될 수 없음. 이는 P1보다 우선순위가 높은 P2가 계속 수행 상태(또는 수행 가능 상태)에 있기 때문임
  ✓ 결국 P2와 P1은 발생하지 않을 사건을 서로 기다리고 있는 상태, 즉 교착 상태에 빠진 것임

다. 세마포어

• 운영체제와 프로그래밍 언어 수준에서 병행성을 위해 제공되는 기법

- 세마포어(semaphores) / 모니터(monitor) / 메시지 전달(message passing)

• Dijkstra 기법의 기본적인 원리 - 세마포어

- 두 개 이상의 프로세스들은 간단한 형태의 신호를 이용해 협력

- 한 프로세스가 특정 신호를 수신할 때까지 정해진 위치에서 중지하도록 강제하는 것

- 보다 복잡한 프로세스들 간의 상호작용에 대한 요구 조건도 적절한 구조의 신호를 이용하면 만족될 수 있음

- 신호를 보내고 받기 위해 세마포어라 불리는 특수 변수들을 사용
  ✓ 세마포어 s를 통해 신호를 전송하기 위해 프로세스는 프리미티브 semSignal(s)를 수행
  ✓ 세마포어 s를 통해 신호를 수신하기 위해 프로세스는 프리미티브 semWait(s)를 수행
  ✓ 만일 특정 신호를 받으려는 프로세스에게 아직 해당 신호가 전달되지 않았다면, 전달될 때까지 프로세스의 수행은 보류됨

• 세마포어

- 정수 값을 가지는 변수

- 세 가지 연산을 통해 접근.
  ✓ 세마포어 초기화 : 세마포어는 음이 아닌 값으로 초기화
  ✓ semWait 연산 : 세마포어 값을 감소시킴, 만일 값이 음수가 되면 semWait를 호출한 프로세스는 블록됨, 음수가 아니면 프로세스는 계속 수행될 수 있음
  ✓ semSignal 연산 : 세마포어 값을 증가시킴, 만약 값이 양수가 아니면 semWait 연산에 의해 블록된 프로세스들을 깨움

```
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 수행 가능으로 전이시키고 ready queue에 연결
  }
}
```

- 이진 세마포어(binary semaphore 또는 mutex)

- 보다 제한된 버전 중 하나

- 이진 세마포어는 오직 0과 1만 유지

- 다음 세 가지 연산에 의해 정의

✓ 세마포어 초기화

✓ 이진 세마포어는 0이나 1로 초기화

✓ semWaitB 연산

· 세마포어 값을 확인

· 만일 값이 0이면 semWaitB를 호출한 프로세스는 블록

· 만일 값이 1이면, 값을 0으로 변경시키고 프로세스는 계속 수행

✓ semSignalB 연산

· 블록되어 있는 프로세스가 존재하는지 확인

· 만일 블록되어 있는 프로세스가 존재하면 그 프로세스를 깨움

· 반면 블록되어 있는 프로세스가 존재하지 않으면 세마포어 값을 1로 설정

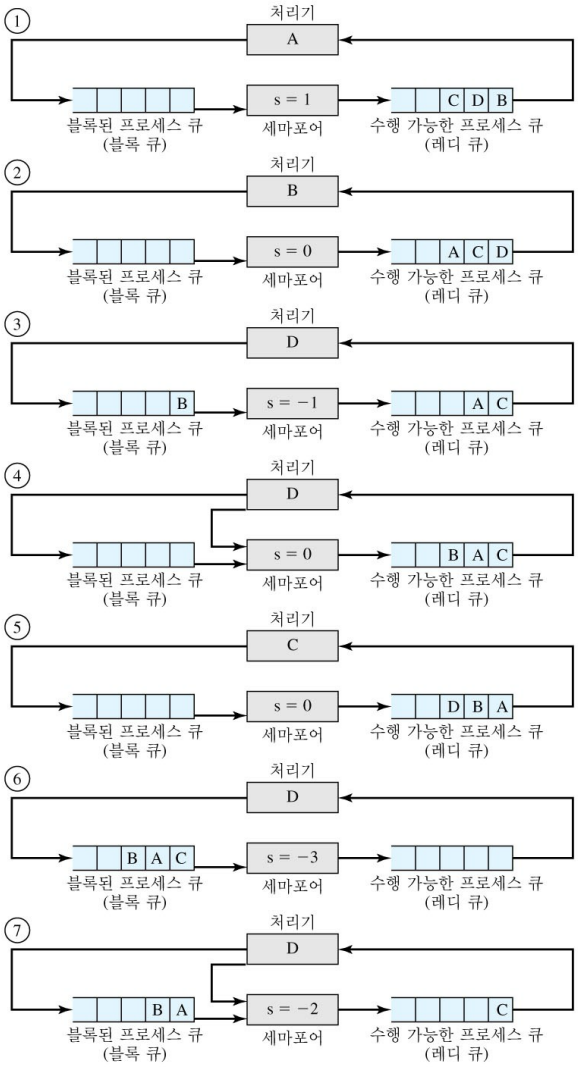
```
struct binary_semaphore {
  enum {zero, one} value;
  queueType queue;
}

void semWaitB(binary_semaphore s)
{
  if (s.value == 1)
    s.value = 0;
  else {
    요청한 프로세스를 s.queue에 연결
    요청한 프로세스를 블록 상태로 전이시킴
  }
}
void semSignal(binary_semaphore s)
{
  if (s.queue is empty())
    s.value = 1;
  else {
    s.queue에 연결되어 있는 프로세스를 큐에서 제거
    프로세스의 상태를 수행 가능으로 전이시키고 ready queue에 연결
  }
}
```

- 이진 세마포어(binary semaphore 또는 mutex)의 장점

- 이진 세마포어 vs. 이진이 아닌 세마포어 = 카운팅 세마포어 = 범용 세마포어
- 원칙적으로 이진 세마포어가 범용 세마포어에 비해 구현하기가 쉬움
- 그 표현력은 범용 세마포어와 동일

- 범용 세마포어와 이진 세마포어 모두 세마포어에서 블록된 프로세스들을 관리하기 위해 큐(queue)를 사용

- 문제점 : 큐에 연결된 프로세스들이 여러 개 있을 때 어떤 프로세스부터 깨워야 하는가?
- 가장 공정한 정책은 선입선출(FIFO)
- 가장 오래 블록되어 있던 프로세스를 먼저 깨워주는 것 => 강성 세마포어(strong semaphore)
  - ✓ 장점
    - 기아 상태가 발생하지 않음
    - 직관적이며 구현하기도 편리
    - 실제 많은 운영체제에서 사용
- 반면 프로세스들이 큐에서 제거되는 순서를 특별히 명시하고 있지 않은 세마포어 => 약성 세마포어(weak semaphore)
- 강성 세마포어의 예



✓ 프로세스 A, B, C는 프로세스 D의 수행 결과에 의존하여 수행

✓ 프로세스 D가 데이터를 생성하면 프로세스 A, B, C 중에서 하나가 그 데이터를 소비하는 관계

· 세마포어 s는 D의 생성 결과가 존재하는지의 여부를 나타냄

☞ s = 1이면 D가 생성한 데이터 하나가 존재한다는 의미

☞ s = 2이면 두 개의 데이터가 존재한다는 의미

☞ 반면 s = -1이면 D의 데이터 생성을 기다리며 블록된 프로세스가 하나 있다는 의미

✓ 초기에 D가 생성한 하나의 데이터가 있고, 모든 프로세스가 수행 가능한 상태이고, 프로세스 A가 스케줄링 되었다고 가정 ①

· D가 생성한 데이터가 하나 있기 때문에 s = 1이고 프로세스 B, C, D는 준비 큐에 연결되어 있으며, 프로세스 A는 수행 중 프로세스 A는 D가 생성한 데이터를 소비하기 전에 semWait(s)를 호출

· D가 생성한 데이터가 하나 있기 때문에 s = 1이고 프로세스 B, C, D는 준비 큐에 연결되어 있으며, 프로세스 A는 수행 중

②

· 프로세스 A는 D가 생성한 데이터를 소비하기 전에 semWait(s)를 호출 => s는 0으로 감소하고, A는 수행을 계속할 수 있음
· 그 이후 A는 준비 큐로 들어가고, B가 스케줄링

③

· B 역시 데이터를 소비하기 전에 semWait(s)를 호출
· 이미 s = 0이기 때문에 프로세스 B는 단지 s 값만 -1로 변경할 뿐 더 이상 수행되지 못하고 블록됨
· 프로세스 D가 스케줄링

④

· 프로세스 D가 수행을 완료하면 새로운 데이터를 하나 생성
· D는 semSignal(s)를 호출 => s를 1 증가시키고 블록되어 있던 프로세스 B를 준비 큐로 이동

⑤⑥

· 수행을 완료한 D는 준비 큐로 들어가고, C가 스케줄링
· 프로세스 C는 데이터를 소비하기 전에 semWait(s)를 호출 => 이미 s = 0이기 때문에 프로세스 C는 s 값을 1 감소시키고 블록됨
· 이후 A, B도 데이터를 사용하기 전에 semWait(s)를 호출 => 이미 s가 음수이므로 s 값을 1 감소시키고 블록됨 => s= -3

⑦

· D가 수행되면서 새로운 데이터를 하나 생성하고 semSignal(s)를 호출 => s를 1 증가시키고 블록되어 있던 프로세스 C는 깨어나 준비 큐로 이동
· 이후 수행이 계속 진행되면 A와 B도 블록에서 깨어나게 됨

세마포어를 이용한 상호배제

```
/* 상호배제 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));
}
```

<세마포어 s를 이용한 상호배제>

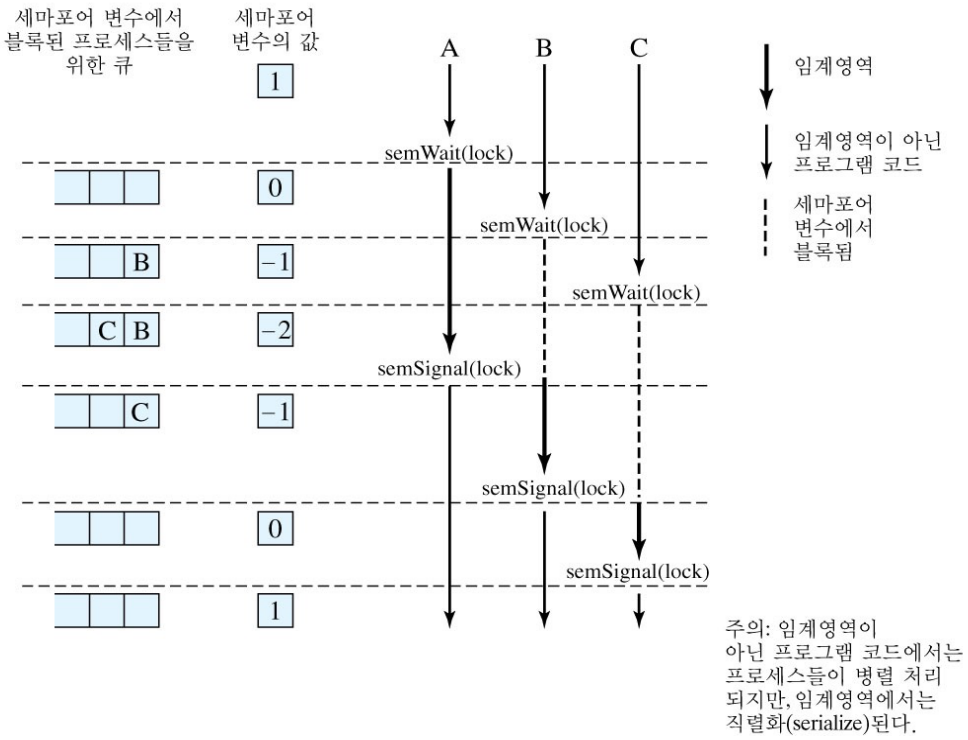
- 동일한 자원을 접근하려는 n개의 프로세스가 존재하며, 각 프로세스 i는 P(i)를 수행
- 프로세스가 공유 자원을 접근하려는 코드 부분이 임계영역으로 정의
- 각 프로세스에서는 임계영역에 들어가기 직전에 semWait(s)를 호출

- 만일 세마포어 s의 값이 양수가 아니라면 프로세스는 블록
- 만일 세마포어 s의 값이 1이라면 0으로 감소시킨 후 임계영역에 들어감
  ✓ 세마포어 s의 값이 더 이상 양수가 아니기 때문에 다른 프로세스들은 임계영역에 진입할 수 없음

- 세마포어 s를 이용한 상호배제 예

- 세마포어는 1로 초기화
- 따라서 최초로 semWait를 수행하는 프로세스가 s의 값을 0으로 바꾸고 임계영역에 들어감

- 임계영역에 들어가고자 하는 다른 프로세스들은 블록되고, s의 값은 1씩 감소
- 즉 프로세스들이 임계영역의 진입을 시도할 때마다, s의 값은 하나씩 감소됨
- 처음에 임계영역에 들어갔던 프로세스가 빠져 나오면, s는 1 증가되고, 블록되어 있던 프로세스들 중에서 하나가 대기 큐에서 나와서 준비 큐로 이동 =〉 즉 블록 상태에서 수행 가능 상태로 전이
- 프로세스가 운영체제에 의해 스케줄되면 결국 임계영역에 들어가게 됨
- 이 예에서 우리는 한 순간에 하나의 프로세스만 임계 영역에 들어감을 알 수 있음



<세마포어 s를 이용한 상호배제>

- 세마포어를 이용해 상호 배제를 구현한 프로세스들의 수행 과정
- A, B, C라는 세 개의 프로세스는 세마포어 변수 lock에 의해 보호되는 공유 자원에 접근
- 우선 프로세스 A가 semWait(lock)를 수행
- 세마포어 변수는 1로 초기화 되어있기 때문에 A는 즉시 임계영역에 들어갈 수 있음 =〉 세마포어 값 = 0
  ✓ 프로세스 A가 임계영역에 있는 동안 semWait(lock)을 호출한 프로세스 B와 C는 블록
- 프로세스 A가 임계영역을 벗어날 때 semSignal(lock)을 수행
- 먼저 블록되었던 프로세스 B가 깨어나 임계영역에 들어감
- 임계 영역에 여러 개의 프로세스가 들어갈 수 있는 경우에도 상호 배제를 보장
- 예를 들어 공유 자원 3개가 있어 동시에 세 프로세스가 임계 영역에 진입할 수 있는 경우에도 상호 배제, 즉 최대 세 개의 프로세스는 임계 영역에 진입할 수 있지만 그 이상의 프로세스가 진입을 시도할 경우 블록시키는 요구 조건을 만족시킴
- 이를 위해서는 단지 세마포어의 초기 값을 임계 영역에 동시에 진입하도록 허용하는 최대의 프로세스 수로 설정
- s.count ≥ 0 : s.count는 현재 임계 영역에 블록되지 않고 진입할 수 있는 프로세스 수
- s.count 〈 0: s.count의 절대 값은 s.queue에 블록되어 있는 프로세스의 수

생산자/소비자 문제

- 데이터를 만드는 생산자와 데이터를 사용하는 소비자로 구성

- 하나 이상의 생산자가 데이터(예를 들어 레코드나 문자들)를 생성하여 버퍼에 저장하고, 한 소비자가 있어 한 번에 하나씩 버퍼에서 데이터를 꺼내 소비
- 요구 조건

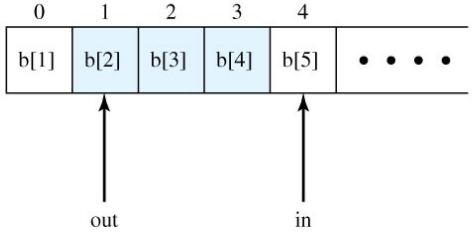
- 버퍼 접근이 중첩되어서는 안 된다는 것
- 한 시점에 생산자나 소비자 프로세스들 중 하나만 버퍼에 접근할 수 있도록 제어

- 우선 공유 버퍼는 무한하며, 버퍼를 구성하는 각 원소들이 선형 배열로 구성되어 있다고 가정
- 생산자와 소비자의 작업의 가상 코드

```
producer:
while (true) {
        /* 데이터 v를 생산 */;
        b[in] = v;
        in++;
}
```

```
consumer:
while(true) {
        while (in <= out)
                /* 대기 */;
        w = b[out];
        out++;
        /* 데이터 w를 소비 */
}
```

- b는 공유 버퍼를 의미
- 생산자는 데이터를 생성하고 버퍼에 저장
  ✓ 이때 저장되는 위치가 in이라는 이름의 인덱스가 가리키는 곳이며, 저장 후 in은 증가
- 소비자는 버퍼에서 데이터를 꺼내 소비
  ✓ 이때 꺼내는 위치는 out이라는 이름의 인덱스가 가리키는 곳이며, 꺼낸 후 out은 증가
- 소비자가 주의해야 할 것은 유효한 데이터가 없는 빈 공간에서 데이터를 꺼내지 않는 것
  ✓ 다시 말해서, 소비자는 버퍼에서 데이터를 읽기 전에 생산자가 앞서가고 있는지(in 〉 out)를 확인해야 함



주의: 음영 부분은 버퍼에서 유효한 데이터가 차지한 공간을 나타낸다.

생산자와 소비자 문제를 이진 세마포어로 구현

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore 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);
}
```

- 인덱스 in, out을 따로 관리하는 대신 변수 n을 사용
- 변수 n은 버퍼에서 유효한 데이터의 수, 즉 n = in - out
- 세마포어 s는 상호배제를 보장하기 위해 사용되며, 세마포어 delay는 버퍼가 비어 있을 때 소비자를 블록시키기 위해 사용
- 생산자는 데이터를 자유롭게 버퍼에 추가 할 수 있음
- 단지 추가하기 전에 상호 배제를 위해 **semWaitB(s)**를 호출하고 추가한 후에 **semSignalB(s)**를 호출하면 됨
- 이것은 버퍼에 데이터를 추가할 때 소비자가 버퍼 사용하는 것(또는 다른 생산자가 버퍼 사용하는 것)을 방지하기 위함
- 그리고 생산자는 임계영역 내에 있는 동안 **n**의 값을 증가시킴
- 증가 이후 **n** = **1**이 되면(증가 이전에 **n** = **0**이었다면) 데이터를 버퍼에 추가하기 바로 전에 버퍼가 비어 있었다는 것을 의미함
- 따라서, 생산자는 **semSignalB(delay)**를 호출하여 버퍼에 데이터가 채워졌다는 사실을 소비자에게 알려줌
- 그러면 소비자는 임계영역으로 진입하여 데이터를 꺼내고 **n**을 감소시킴
- 만일 생산자가 소비자에 비해 먼저 수행될 수 있다면(실제로 이 경우가 일반적이다), **n**이 대부분 양수이기 때문에 소비자가 세마포어 **delay**에서 블록되는 경우는 매우 드물 것임
- 따라서 생산자와 소비자 모두 원활하게 수행됨

• 위 프로그램의 문제점

- 소비자가 버퍼의 데이터를 모두 소진하면 생산자가 새로운 데이터를 생산할 때까지 세마포어 delay에 대기하야 함.
  - ✓ 프로그램에서 문장 if (n == 0) semWaitB(delay)가 이 역할을 담당
  - ✓ 라인 14에서 소비자는 semWaitB(delay)를 수행하지 않음

✓ 사실 소비자는 버퍼에 있는 데이터를 모두 사용하였기 때문에 n을 0으로 설정하였지만(라인 8), 그것을 확인하고 semWaitB(delay)를 호출하기 전에 (라인 14) 생산자가 n을 이미 증가시킴(라인 11)

✓ 그 결과 semSignalB는 이전의 semWaitB와 서로 일치하지 않게 됨

✓ 라인 20에서 n이 -1인 것은 소비자가 버퍼에서 존재하지 않는 데이터를 소비했다는 것을 뜻함

✓ 해결책 1 : 조건 문장 if (n == 0) semWaitB(delay)를 소비자의 임계영역 내로 옮김 =〉 교착상태발생 가능

	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	-1	0
21		semSignalB(s)	1	-1	0

표에서 하얀 영역은 세마포어에 의해 제어되는 임계영역을 나타냄

• 문제점 해결 방법 1

- 이 문제를 제대로 해결하기 위한 방법은 소비자의 임계영역 내에서 보조 변수를 사용하는 것

- 이 변수는 이후에 사용하기 위해 소비자의 임계영역 내에서 설정됨

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
  while (true) {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n == 1)
      semSignalB(delay);
    semSignalB(s);
  }
}
void consumer()
{
  int m; /* 지역 변수 */
  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);
}
```

• 문제점 해결 방법 2

- 보다 깔끔한 해결 방법은 범용 세마포어를 사용

- 이 해결책에서 n은 세마포어로, 그 값은 버퍼에 존재하는 유효 데이터의 개수를 나타냄

- 여기서 프로그램을 옮겨 쓰는 과정에서 실수로 인해 생산자 프로그램의 semSignal(s)와 semSignal(n)의 위치가 서로 바뀌었다고 가정할 경우

✓ 세마포어 s에 의해 보호되는 임계영역 내부에서 소비자나 다른 생산자의 개입 없이 semSignal(n) 연산이 수행

✓ 프로그램 수행에 영향이 있는가?

· 소비자는 어떤 경우든 두 세마포어를 기다려야하기 때문에 수행 과정에는 영향이 없음

```
/* program producerconsumer */
semaphore n = 0;
semaphore s = 1;

void producer()
{
  while (true) {
    produce();
    semWait(s);
    append();
```

```
    semSignal(s);
    semSignal(n);
  }
}
void consumer()
{
  while (true) {
    semWait(n);
    semWait(s);
    take();
    semSignalB(s);
    consume();
  }
}
void main()
{
  parbegin(producer, consumer);
}
```

- 소비자 프로그램에서 semWait(n)과 semWait(s) 연산의 순서가 바뀐 경우를

- 이 경우에는 세마포어 s에 의해 보호되는 임계영역 내부에서 세마포어 n에 대한 semWait를 수행하게 되며, 이로 인해 수행 과정에서 심각한 오류를 야기할 수 있음

- 버퍼가 비어있을 때(n.count = 0) 소비자가 임계영역에 들어가면 세마포어 s를 확보한 채 세마포어 n에서 블록됨

- 한편 생산자는 세마포어 s에 대한 허가를 얻을 수 없어 버퍼에 데이터를 추가하기 위한 임계 영역에 진입하지 못함

✓ 교착상태

- 유한 버퍼를 이용한 생산자 소비자 문제

- 순환 큐로 구현된 버퍼 구조

✓ 인덱스 in과 out의 위치는 버퍼 크기에 의해 제한

✓ 즉 인덱스의 위치가 버퍼 크기보다 커지는 경우 다시 0으로 설정

✓ 이때 다음의 관계가 만족되어야 함

블록됨	블록에서 깨어남
생산자: 가득 찬 버퍼에 데이터를 추가하려 할 때	소비자: 데이터가 추가될 때
소비자: 빈 버퍼에서 데이터를 꺼내려 할 때	생산자: 데이터가 꺼내졌을 때

- 유한 버퍼를 사용하는 경우, 생산자와 소비자 함수 (변수 **in**과 **out**의 초기값은 0).

```
Producer:
while(true)
{
    /*데이터 v를 생산 */
    while((in + 1) % n == out)
        /* 대기 */;
    b[in] = v;
    in = (in + 1) % n;
}
```

```
Consumer:
        while(true)
        {
    while(in == out)
        /* 대기 */;
            w = b[out];
            out = (out +1) % n;
    /* 데이터 w를 소비 */;
        }
```

✓ 범용 세마포어를 이용한 해결책

· 세마포어 e는 버퍼에서 빈 공간의 수

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */
semaphore n = 0;
semaphore s = 1;
semaphore 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();
    semSignalB(s);
    semSignalB(e);
    consume();
  }
}
void main()
{
  parbegin(producer, consumer);
}
```

<유한 버퍼에서 범용 세마포어를 이용한 생산자 소비자 문제 해결 방법>

라. 모니터

- 세마포어의 장점

- 상호배제의 보장과 프로세스 사이의 조정을 위해 사용되는 강력하고 유연한 프리미티브를 제공

- 세마포어의 단점

- 올바른 프로그램을 만든다는 것은 어려운 일

- semWait와 semSignal 연산이 프로그램 전체에 산재해 있고, 이런 연산들이 전체적으로 수행에 어떠한 영향을 미치는지 파악하기가 어렵기 때문에 발생

- 모니터

- 세마포어의 단점 해결

- 모니터는 프로그래밍 언어 수준에서 제공되는 구성체

- 세마포어와 동일한 기능을 제공하며 보다 사용하기 쉬움

- 모니터에 대한 개념은 [HOAR74]에서 처음으로 제안

- Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java 등 다양한 프로그램 언어에 구현되어 있으며, 프로그램 라이브러리로 구현되기도 하였음

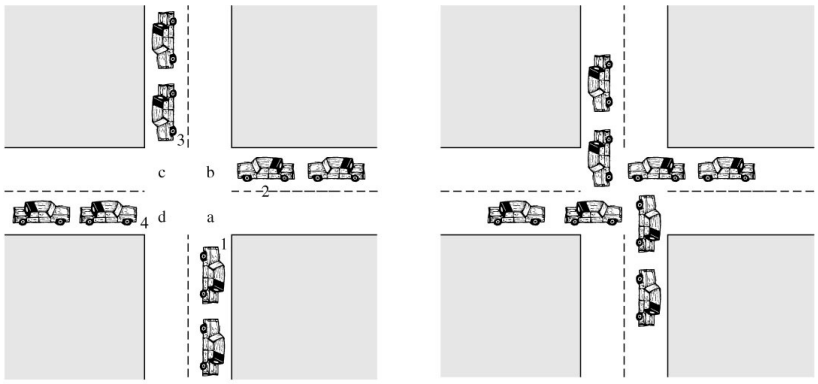
- 프로그래머는 자신이 보호하려는 어떠한 객체에도 모니터 락(monitor lock)을 설정할 수 있음

- 예를 들어 리스트 구조의 경우, 프로그래머는 리스트 구조 전체에 하나의 모니터 락을 설정할 수도 있고, 각 리스트마다 모니터 락을 설정할 수도 있으며, 각 리스트에 존재하는 각 원소마다 모니터 락을 설정할 수도 있음

마. 교착상태

- 교착상태

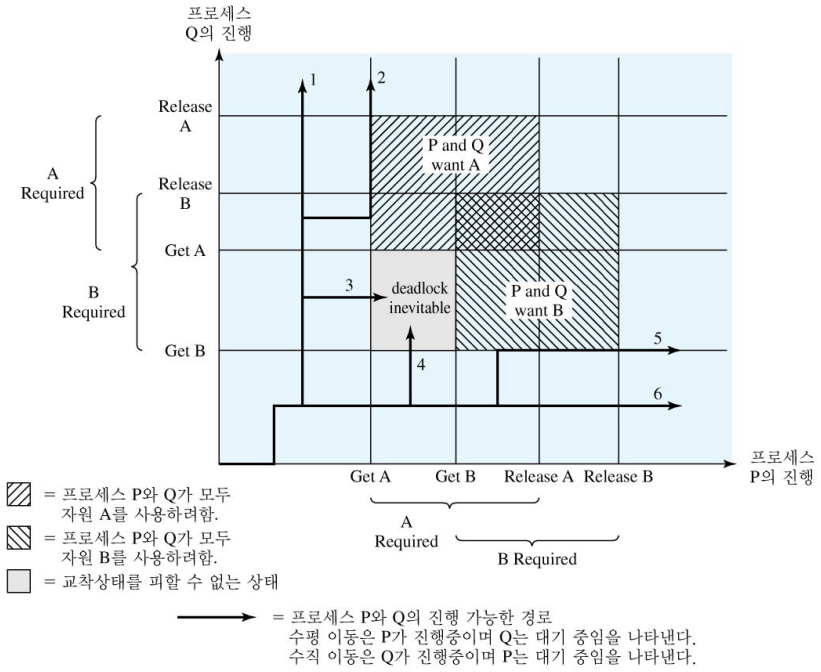
- 프로세스들의 집합이 더 이상 진행을 못하고 영구적으로 블록되어 있는 상태
- 교착상태는 시스템 자원에 대한 경쟁 도중에 발생할 수도 있고 프로세스 간 통신 도중에 발생할 수도 있음
- 집합 내의 한 프로세스가 특정 사건의 발생을 기다리며 대기하고 있고 (예를 들어 요청한 자원이 가용해지기를 기다림), 이 사건이 집합 내의 다른 블록된 프로세스에 의해 발생될 수 있을 때 이 프로세스의 집합은 교착 상태가 됨
- 교착상태가 영구적인 이유
  - ✓ 기다리던 사건이 결코 발생하지 않기 때문
- 다른 병행 프로세스 관리 문제들과는 달리, 모든 교착상태에 적용될 수 있는 범용 해결책은 아직 없음
- 교착상태는 두개 이상의 프로세스들이 서로 충돌되는 자원 요구를 할 때 발생
- 교착상태의 대표적인 예
  - ✓ 교통 교착상태(traffic deadlock)

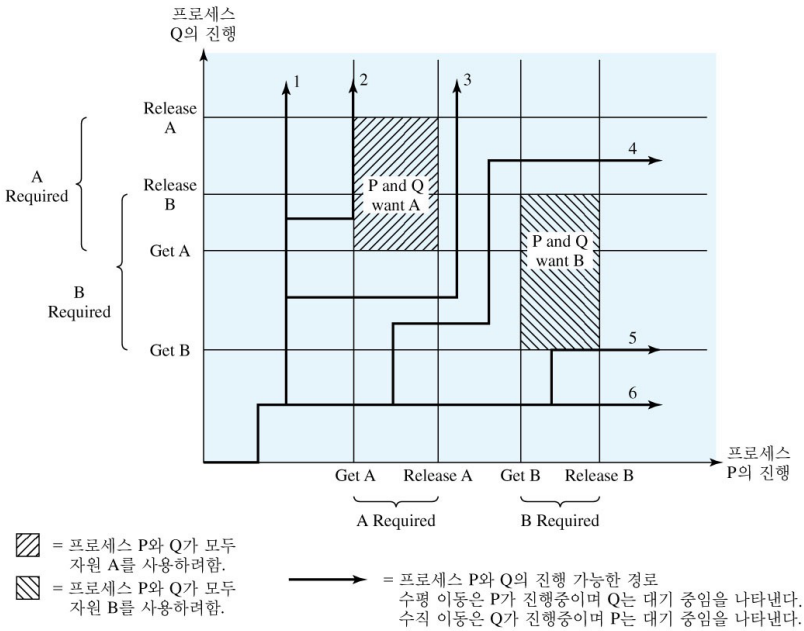


(a) 교착상태 가능성 존재 (b) 교착상태 발생

- 프로세스와 컴퓨터 자원 관계에서 교착상태

- P와 Q라는 두개의 프로세스가 A와 B라는 자원을 경쟁적으로 사용하며 수행하는 과정
- 결합 진행 다이어그램(Joint Progress Diagram)
- 각 프로세스는 일정 기간 동안 두개의 자원을 동시에 배타적으로 사용해야 함
- 진행경로 시나리오
  - ✓ 1. 프로세스 Q가 자원 B를 획득하고 그 이후에 다시 자원 A를 획득한다. 수행 완료 이후 프로세스 Q는 두 자원을 반납한다. 이제 프로세스 P가 수행되면 두 자원을 사용할 수 있다.
  - ✓ 2. 프로세스 Q가 자원 B를 획득하고 그 이후에 다시 자원 A를 획득한다. 이때 프로세스 P가 수행된다. 하지만 P는 자원 A의 획득에 실패하고 블록된다. 프로세스 Q가 수행을 완료하고 자원들을 반납한다. 블록되었던 프로세스 P가 깨어나면 두 자원을 사용할 수 있다.
  - ✓ 3. 프로세스 Q가 자원 B를 획득하고 프로세스 P가 자원 A를 획득한다. 프로세스 Q는 자원 A의 획득을 시도하다가 블록되고, 프로세스 P는 자원 B의 획득을 시도하다가 블록된다. 그런데 프로세스들이 서로를 기다리며 블록되어 있기 때문에 더 이상 진행될 수 없다. 교착상태가 발생한 것이다.
  - ✓ 4. 프로세스 P가 자원 A를 획득하고 프로세스 Q가 자원 B를 획득한다. 프로세스 P는 자원 B의 획득을 시도하다가 블록되고, 프로세스 Q는 자원 A의 획득을 시도하다가 블록된다. 이 경우도 교착상태이다.
  - ✓ 5. 프로세스 P가 자원 A를 획득하고 그 이후에 다시 자원 B를 획득한다. 이때 프로세스 Q가 수행된다. 하지만 Q는 자원 B의 획득에 실패하고 블록된다. 프로세스 P가 수행을 완료하고 자원을 반납한다. 블록되었던 프로세스 Q가 깨어나면 두 자원을 사용할 수 있다.
  - ✓ 6. 프로세스 P가 자원 A를 획득하고 그 이후에 다시 자원 B를 획득한다. 수행 완료 이후 프로세스 P는 자원을 반납한다. 이제 프로세스 Q가 수행되면 두 자원을 사용할 수 있다.





- 프로세스가 사용하는 자원

- 재사용 가능한 (reusable) 자원
  - ✓ 재사용 가능한 자원은 프로세스의 사용에 의해 없어지지 않는 자원
  - ✓ 프로세스가 사용한 후 다른 프로세스가 다시 사용할 수 있도록 반납
  - ✓ 처리기, 입출력 채널, 주/보조 메모리, 장치, 파일이나 데이터베이스나 세마포어와 같은 자료 구조 등
- 소모성(consumable) 자원
  - ✓ 생성되었다가 사용된 이후 소멸되는 자원
  - ✓ 특정한 유형의 소모성 자원에는 개수의 제한이 없음

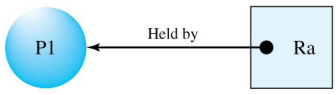
✓ 블록되지 않은 생산 프로세스(producing process)는 자원을 몇 개라도 생산할 수 있음
✓ 자원이 소비 프로세스(consuming process)에 의해 사용되면, 그 자원은 사라짐
✓ 인터럽트, 시그널, 메시지, I/O 버퍼에 존재하는 정보 등

- 모든 유형의 교착상태를 효과적으로 해결하는 단일한 전략은 없음

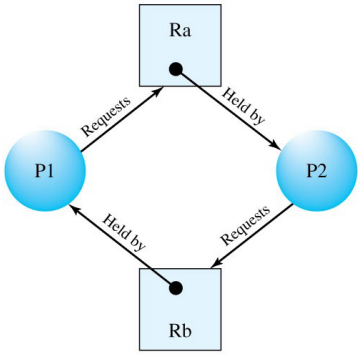
접근 방법	자원 할당 정책	구체적인 기법	장점	단점
예방	보수적. (자원 할당이 가능하더라도 조건에 따라 할당하지 않을 수 있다.)	모든 자원을 한꺼번에 요구	순간적으로 많은 일을 하는 프로세스에게 적합 선점이 불필요	효율이 나쁨 프로세스 시작을 지연시킬 가능성 있음 프로세스는 사용할 모든 자원을 미리 알고 있어야 함
		선점 가능	자원 상태의 저장과 복구가 간단한 자원에는 적용하기 쉬움	선점 부하
		자원 할당 순서	컴파일 시점에 강제할 수 있음 시스템의 설계 시점에 문제를 해결했기 때문에 동적 부하 없음	점진적인 자원 할당이 안 됨
회피	예방과 발견의 중간 정도.	교착상태가 발생하지 않는 안전한 경로를 최소한 하나는 유지	선점이 불필요	운영체제는 자원에 대한 미래 요구량을 미리 알고 있어야 함 오랜 기간 지연 발생의 가능성 있음
발견	적극적. (자원 할당이 가능하면 즉시 할당한다.)	주기적으로 교착상태 발생 여부 파악	프로세스 시작을 지연시키지 않음 온라인 처리 가능	선점에 의한 손실 발생

자원 할당 그래프

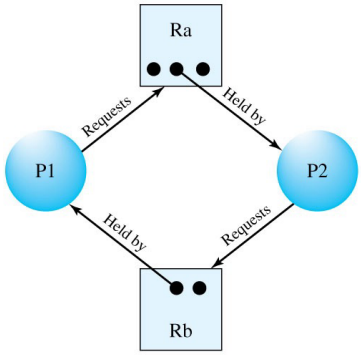
(a) 프로세스가 자원을 요청하고 있다.



(b) 자원이 프로세스에게 할당되었다.



(c) 환형 대기



(d) 교착상태가 아닌 경우

교착상태 조건

- 1. 상호배제(mutual exclusion) 조건: 한 순간에 한 프로세스만이 자원을 사용할 수 있다. 즉 한 프로세스에 의해 점유된 자원을 다른 프로세스들이 접근할 수는 없다.
- 2. 점유대기(hold and wait) 조건: 자원을 기다리는 프로세스가 이미 다른 자원을 할당하고 있다.
- 3. 비선점(no preemption) 조건: 프로세스에 의해 점유된 자원을 다른 프로세스가 강제적으로 빼앗을 수 없다.
- 4. 환형 대기(circular wait) 조건: 프로세스들 간에 닫힌 연결(closed chain)이 존재한다. 즉, 자원 할당 그래프에서 환형이 만들어 지는 것이다. 닫힌 연결에서 블록된 프로세스가 자원을 점유하고 있는데, 이 자원을 체인 내부의 다른 프로세스가 원하며 대기하고 있다
- 조건 1~3은 교착상태가 발생할 수 있는 필요조건
- 조건 4까지 만족되면 교착상태가 발생할 수 있는 필요충분조건
- 사실 조건 4는 조건 1~3의 결과에 의해 발생

- 조건 1~3의 복잡한 상호작용의 결과 해결할 수 없는 환형 대기 상태가 발생
- 교착상태의 정의가 바로 해결할 수 없는 환형 대기 상태
- 환형 대기 상태가 해결될 수 없는 이유는 조건 1~3이 지켜지기 때문
- 결국 위 4가지 조건이 교착상태가 발생할 수 있는 필요충분조건

바. 교착상태 예방

- 교착상태 예방 전략 에서 사용하는 전략은 운영체제를 설계할 때 교착상태가 발생할 가능성을 없애는 것

- 교착상태가 발생하기 위한 4가지 필요충분조건 중에 하나를 설계 단계에서 배제하는 것
- 간접 예방 vs. 직접 예방
  - ✓ 간접 예방 : 조건 1~3 중에 하나를 허용하지 않는 것
  - ✓ 직접 예방 : 환형대기를 허용하지 않는 것

상호배제

- 상호배제 조건은 공유 자원의 일관성을 유지하기 위해 반드시 필요하기 때문에, 자원 접근에서 상호배제가 필요하면 운영체제가 이를 지원해 주어야 함

- 예) 파일의 경우, 다수의 읽기 접근은 허용되지만 쓰기 접근은 한 시점에 하나만 배타적으로 허용. 다수의 프로세스들이 쓰기 권한을 얻으려고 할 때 교착상태 발생 가능

점유 대기

- 프로세스는 자신이 사용할 모든 자원을 한순간에 요청
- 만일 모든 자원을 할당받을 수 있으면 계속 수행한다. 반면 하나의 자원이라고 할당받을 수 없으면, 어떠한 자원도 할당받지 않은 채 대기
- 비효율적

- 모든 자원을 할당받기 위해 오랜 기간 대기 가능
- 한꺼번에 할당 받은 자원 중 일부는 실제 수행이 끝날 때쯤에 사용될 수 있음. 실제로 이용되지 않으면서 점유될 가능성 있음
- 프로세스가 미래에 사용될 모든 자원 미리 예측 불가능

Non Preemption

- 방법 1 : 자원을 점유한 프로세스가 다른 자원을 요청했을 때 할당받을 수 없다면, 일단 자신이 점유한 자원을 일단 반납 후 프로세스는 원래 자원과 새로 원하는 자원을 함께 요청하는 방법
- 방법 2 : 한 프로세스에서 다른 프로세스가 점유한 자원을 원하면, 운영체제는 다른 프로세스가 점유한 자원을 강제적으로 반납

시키고 그것을 원하는 프로세스에게 할당

- 프로세스들이 서로 다른 우선순위를 갖고 있을 때 교착상태를 예방 가능
- 자원의 상태를 저장하고 복구하기 쉬운 자원에 적용
- CPU

환형대기 Circular Wait

- 자원들의 할당 순서를 정해 환형 대기조건 제거 가능
- 자원 사용의 비효율성과 프로세스의 오랜 대기를 야기

사. 교착상태 회피(avoidance, prevention)

- 회피 방법은 교착상태 발생 조건 중 1~3은 허용
- 자원을 할당할 때 교착상태가 발생 가능한 상황으로 진행하지 않도록 고려
- 예방 방법에 비해 더 많은 병행성을 제공(효율성이 높음)
- 교착상태 회피는 프로세스가 자원을 요청하고 그 자원이 사용 가능할 때 그대로 할당해 주지 않음
- 할당 전에 이 할당이 교착상태를 발생시킬 가능성이 있는지 조사
- 만일 교착상태를 발생시킬 가능성이 있다면 자원을 할당하지 않음
- 문제점 - 현재 자원의 가용 개수와 프로세스의 자원 요구량을 미리 알고 있어야 가능
- 교착상태 회피 기법

- 프로세스 시작 거부 : 프로세스가 시작하려 할 때 요구하는 자원 할당이 교착상태 발생의 가능성이 있으면, 프로세스를 시작시키지 않는 방법
- 자원 할당 거부 : 수행 중인 프로세스가 요구하는 추가적인 자원 할당이 교착상태 발생의 가능성이 있으면, 자원을 할당하지 않는 방법

프로세스 시작 거부

- 시스템에 n개의 프로세스들과 m개의 다른 유형에 자원이 있다고 가정
- 벡터와 행렬을 정의하고, 이를 이용하여 교착상태 회피

자원 = R = $(R_1, R_2, \ldots, R_m)$	시스템에 존재하는 자원의 전체 개수
가용 = V = $(V_1, V_2, \ldots, V_m)$	시스템에 존재하는 자원 중 현재 사용가능한 자원의 개수
요구 = C = $\begin{vmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ & & \ldots & \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{vmatrix}$	C_{ij}는 프로세스 i가 자원 j를 요구하고 있음을 의미
할당 = A = $\begin{vmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ & & \ldots & \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{vmatrix}$	A_{ij}는 프로세스 i가 자원 j를 할당받아 점유하고 있음을 의미

- 행렬 C와 A는 프로세스가 요구하고 있는 자원과 현재 프로세스가 점유하고 있는 자원의 개수
- 각 행은 각 프로세스에 대응
- 교착상태 회피를 적용하기 위해서 운영체제는 프로세스 수행 전에 이 정보들을 미리 알고 있어야 함

✓ $R_i = V_i + \sum_{i}^{n} A_{ij}$, for all j. 모든 자원은 가용하거나 할당되어 있음(이 식은 가용 자원과 할당된 자원의 합이 결국 그 자원의 전체 개수와 동일함을 의미)

✓ 2. $C_{ij} \leq R_i$, for all i, j. 프로세스는 자원의 전체 개수 보다 더 많은 개수의 자원을 요구할 수 없음

✓ 3. $A_{ij} \leq C_{ij}$, for all i, j. 프로세스는 자신이 요구한 자원보다 더 많은 개수의 자원을 할당할 수 없음

- 모든 프로세스들이 요구한 자원의 개수가 전체 자원 개수보다 적으면 교착상태가 발생하지 않음
- 교착상태 회피를 위한 프로세스 시작 거부 방법은 다음의 식을 만족할 때 새로운 프로세스의 시작을 허가

$$R_j \geq C_{(n+1)j} + \sum_{i}^{n} C_{ij}, \textbf{\textit{ for all }} j.$$

✓ $\sum_{i}^{n} C_{ij}$: 현재 존재하는 모든 프로세스들이 요구하는 자원의 개수

✓ $C_{(n+1)j}$: 새로 수행하려는 프로세스가 요구하는 자원의 개수

✓ 새로운 프로세스와 기존의 프로세스들이 요구하는 자원의 개수가 그 자원의 전체 개수보다 적을 경우에 수행을 허용

✓ 비효율적 : 최악의 경우, 즉 모든 프로세스들이 동시에 최대 자원을 요구하여 사용함을 가정

자원 할당 거부

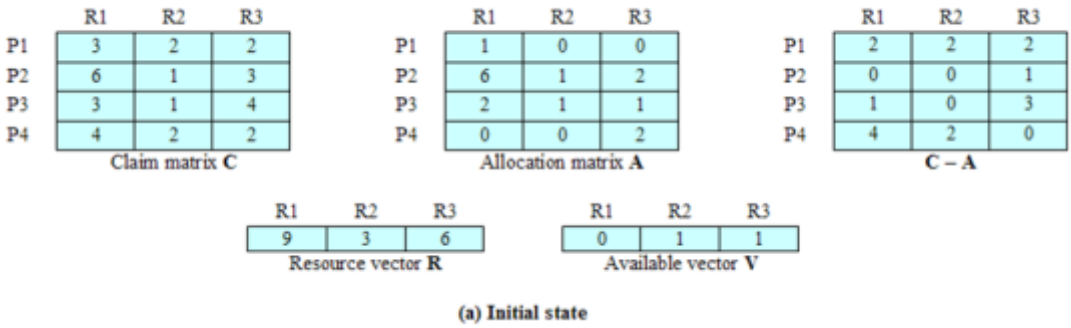
- 은행원 알고리즘
- 시스템에 고정된 수의 프로세스와 자원이 존재한다고 가정
- 시스템의 상태를 안전한 상태(safe state)와 그렇지 않은 상태(unsafe state)로 구분

- 시스템의 상태란 프로세스와 자원의 할당과 요구 관계를 의미
- 시스템의 상태란 이전에 살펴보았던 2개의 벡터와 행렬, 즉 자원, 가용이라는 벡터(R, V)와 요구와 할당이라는 행렬(C, A)로 표현
- 안전한 상태란 교착상태가 발생하지 않도록 프로세스에게 자원을 할당할 수 있는 진행 경로가 존재함을 의미

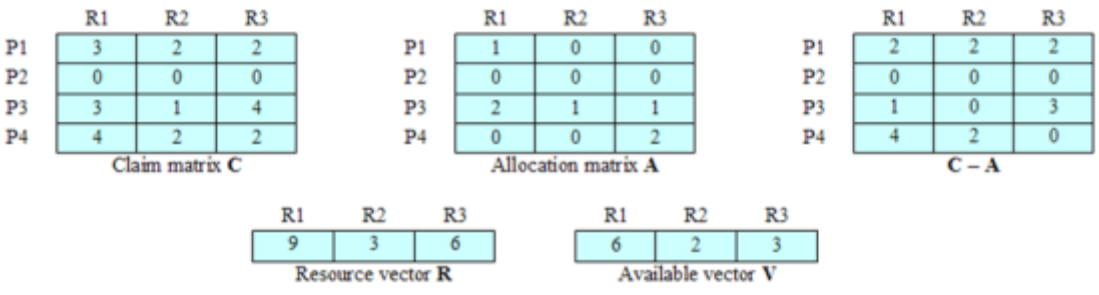
✓ 모든 프로세스가 자신의 자원 요구량을 모두 만족하여 수행을 마칠 수 있음을 의미

- 안전하지 않은 상태란 그러한 진행 경로가 없는 상태이다.

- 안전한 상태 예) 그림 6.7



(a) Initial state



(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

- 안전하지 않은 상태

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

- 교착상태 회피의 장점

- 교착상태 예방에 비해 자원 할당이 더 자유로움

- 시스템에서 자원 효율이 높음

- 교착상태 회피의 단점

- 각 프로세스들이 사용할 최대 자원 요구량을 미리 운영체제에게 알려 줘야 함.

- 프로세스들은 서로 독립적이어야 함

✓ 즉 프로세스들 중에 한 프로세스는 다른 프로세스에 비해 먼저 수행되어야 한다는 것과 같은 수행 순서 종속 관계가 없어야 함

- 자원 개수가 고정되어 있어야 함

- 자원을 선점한 채 종료하는 프로세스는 없어야 함

아. 교착상태 발견(detection)

- 예방 전략

- 상당히 보수적인 방법

- 교착상태가 발생하지 않도록 하기 위해 자원 접근에 대한 제한과 프로세스의 수행 제한

- 발견 전략

- 좀 더 낙관적인 방법

- 자원 접근에 대한 제한이나 프로세스의 행위에 제한 없음

- 즉 자원 할당이 가능한 상황이면 항상 할당함

- 단, 주기적으로 시스템에 환형 대기조건이 발생했는지 검사하고 발생하면 해결

교착상태 발견 알고리즘

- 자원 할당이 요구될 때마다 매번 수행 vs. 주기적인 수행

- 자원 할당이 요구될 때마다 매번 수행하면 빨리 교착 상태 발견 가능하고 알고리즘 간단

- 처리 비용

- 알고리즘

- 할당 행렬, 가용 벡터 및 요청 행렬 Q(request matrix Q) 사용

- Qij는 프로세스 i에 의해 요청된 자원 j의 개수

- 교착상태가 아닌 프로세스들을 찾아 표시(mark)

- 초기에는 모든 프로세스들이 표시되어 있지 않음

✓ 1. 할당 행렬에서 행의 값이 모두 0인 프로세스를 우선 표시

✓ 2. 임시 벡터 W를 만들고 현재 사용 가능한 자원의 개수(결국 가용 벡터의 값)를 벡터 W의 초기 값으로 설정

✓ 3. 표시되지 않은 프로세스들 중에서 요청 행렬 Q의 특정 행의 값이 모두 W보다 작은 프로세스를 찾음. 즉 $Q_{ik} \leq W_k$ for $1 \leq k \leq m$인 i를 찾고 프로세스 i를 표시. 만일 이러한 프로세스가 없으면 알고리즘을 종료.

✓ 4. 단계 3의 조건을 만족하는 행을 Q에서 찾으면, 할당 행렬에서 그 행에 대응되는 값으로 W를 갱신. 즉 3 단계에서 i를 찾았으면 $W_k = W_k + A_{ik}$ for $1 \leq k \leq m$을 수행한다. 그리고 3 단계를 다시 수행.

- 이 알고리즘이 종료된 이후에도 표시(mark)되지 않은 프로세스가 존재한다면, 교착상태 발생

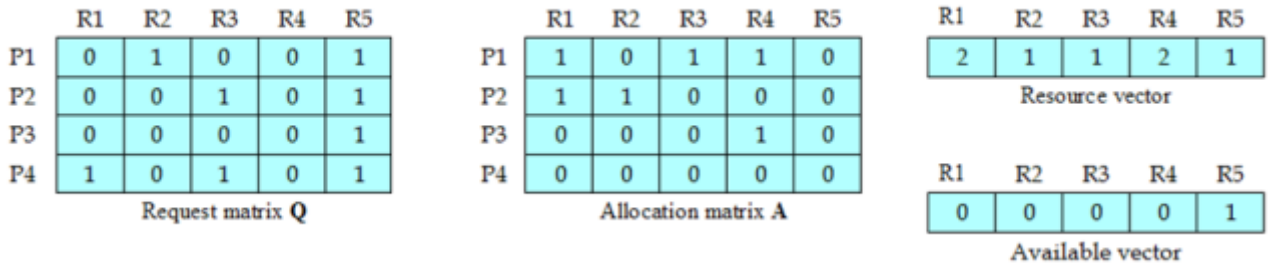
- 각 표시되지 않는 프로세스들은 완료되지 못함

- 이 알고리즘의 원리는 우선 현재 가용 자원으로 수행이 완료될 수 있는 프로세스를 찾고 그런 프로세스가 있다면 그 프로세스가 완료된 이후 반납할 자원을 가용 자원에 추가.

- 그리고 다시 이 가용 자원으로 수행이 완료될 수 있는 프로세스를 찾음.
- 이러한 과정을 반복한 후 알고리즘이 종료 되었을 때 모든 프로세스가 완료되면 교착상태가 없는 것이고, 그렇지 않으면 교착상태가 있는 것임

• 단점

- 현재 교착상태의 존재 여부만 파악할 뿐, 이후에 더 진행되면 교착상태가 생길지 여부는 파악하지 못함



<교착 상태 발견 예>

- ✓ 1. P4는 할당받은 자원이 없다. 따라서 P4를 표시(mark)
- ✓ 2. W=(0 0 0 0 1)로 초기화
- ✓ 3. P3의 요청을 W가 만족할 수 있기 때문에 P3을 표시하고 W=W+(0 0 0 1 0)를 수행 => W=(0 0 0 1 1)이 된다.
- ✓ 4. 표시가 되지 않은 프로세스들 중에서 W로 요청을 만족할 수 있는 프로세스가 더 이상 없음 => 알고리즘 종료
  - 알고리즘 종류 후에도 P1과 P2가 표시가 되지 않은 상태로 남아있으며, 결국 P1과 P2가 교착상태에 있음을 발견 가능

교착상태 회복 알고리즘

• 교착상태에 포함되어 있는 모든 프로세스들을 중지시킴.

- 실제 많은 운영체제에서 채용하고 있는 방법 중에 하나

• 교착상태에 포함되어 있는 각 프로세스의 수행을 롤백시킴. 즉 미리 정의된 특정 체크포인트 시점까지 되돌린 후 다시 수행

- 단점 : 재수행시 다시 교착상태 발생 가능. 병행 처리의 비결정인 특성으로 인해 가능성은 낮음

• 교착상태가 없어질 때까지 교착상태에 포함되어 있는 프로세스들 하나씩 종료

- 종료시킬 프로세스의 선택은 비용이 가정 적은 것부터
  - ✓ 지금까지 사용한 처리기 시간이 적은 프로세스부터
  - ✓ 지금까지 생산한 출력량이 적은 프로세스부터
  - ✓ 이후 남은 수행시간이 가장 긴 프로세스부터
  - ✓ 할당받은 자원이 가장 적은 프로세스부터
  - ✓ 우선순위가 낮은 프로세스부터
- 하나씩 종료 시킨 후 교착상태 발견 알고리즘을 다시 수행시켜보면 아직 교착상태가 존재하는지 여부를 알 수 있음

• 교착상태가 없어질 때까지 교착상태에 포함되어 있는 자원을 하나씩 선점시킴.

- 자원 선택은 3에서와 같이 비용이 가장 적은 자원부터 선택
- 하나씩 선점한 후 교착상태 발견 알고리즘을 수행시켜 아직 교착상태가 존재하는지 파악
- 자원을 선점 당한 프로세스는 자원을 할당 받기 전 시점으로 롤백하고, 그 시점부터 다시 수행

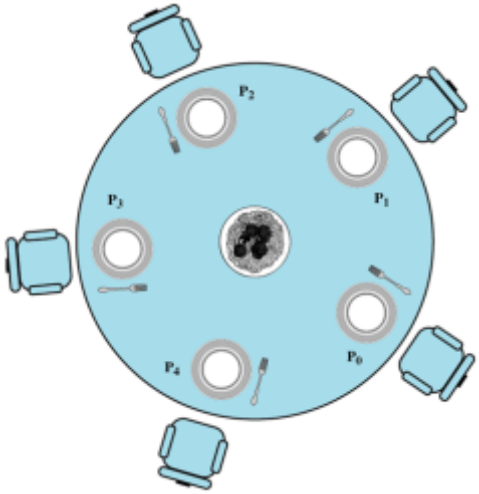
자. 식사하는 철학자 문제

- 고려사항

- 임계 영역 준수

✓ 두 명의 철학자가 동시에 같은 포크를 사용할 수는 없음

- 교착상태나 기아에 빠지면 안됨



세마포어를 이용한 해결방법

- 철학자는 우선 왼쪽에 있는 포크를 먼저 집고, 그 이후에 오른쪽에 있는 포크를 집음
- 식사를 마친 이후에 철학자는 두개의 포크들을 식탁에 다시 내려놓음
- 문제점

- 교착상태가 발생 가능성

```
/* 식사하는 철학자 문제 */
semaphore fork[5] = {1};
int i;
void philosopher(int i)
{
  while (true) {
    think();
    wait(fork[i]);
    wait(fork[(i+1) mod 5]);
    eat();
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
  }
}
void main()
{
  parbegin(philosopher(0),    philosopher(1),    philosopher(2),    philosopher(3),
philosopher(4));
}
```

<세마포어를 이용한 식사하는 철학자 문제 해결 방법: 첫 번째 버전>

- 해결안

- 5개의 포크를 더 구입
- 철학자들에게 하나의 포크로 먹는 방법을 가르침
- 다른 방법으로는 한 번에 최대 4명까지 철학자가 식탁에 앉을 수 있도록 제한

```
/* 식사하는 철학자 문제 */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher(int i)
{
  while (true) {
    think();
    wait(room);
    wait(fork[i]);
    wait(fork[(i+1) mod 5]);
    eat();
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
    signal(room);
  }
}
void main()
{
  parbegin(philosopher(0),    philosopher(1),    philosopher(2),    philosopher(3),
philosopher(4));
}
```

<세마포어를 이용한 식사하는 철학자 문제 해결 방법: 두 번째 버전>