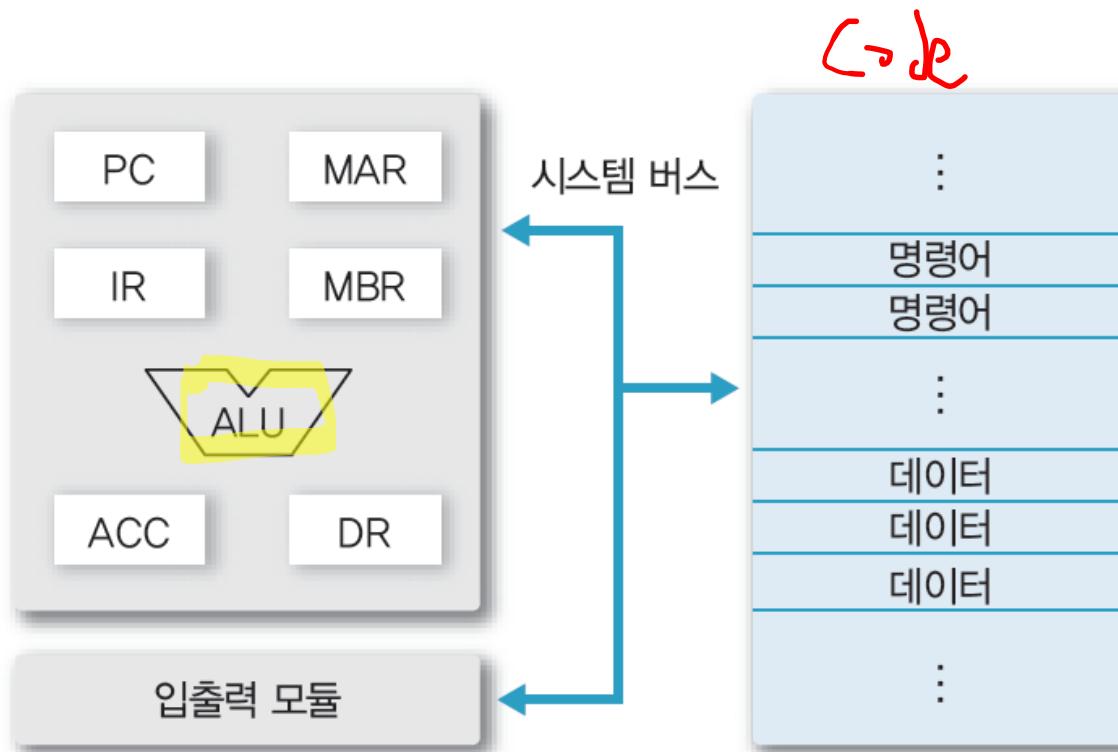


1. 프로세서

■ 프로세서의 기본 레지스터



- ALU Arithmetic Logic Unit : 산술 · 논리 연산장치

그림 1-3 프로세서의 기본 레지스터

2. 메모리

■ 메모리 계층 구조

- 1950~1960년대 너무 비싼 메인 메모리의 가격 문제 때문에 제안한 방법

- 메모리를 계층적으로 구성하여 비용, 속도, 용량, 접근시간 등을 상호 보완

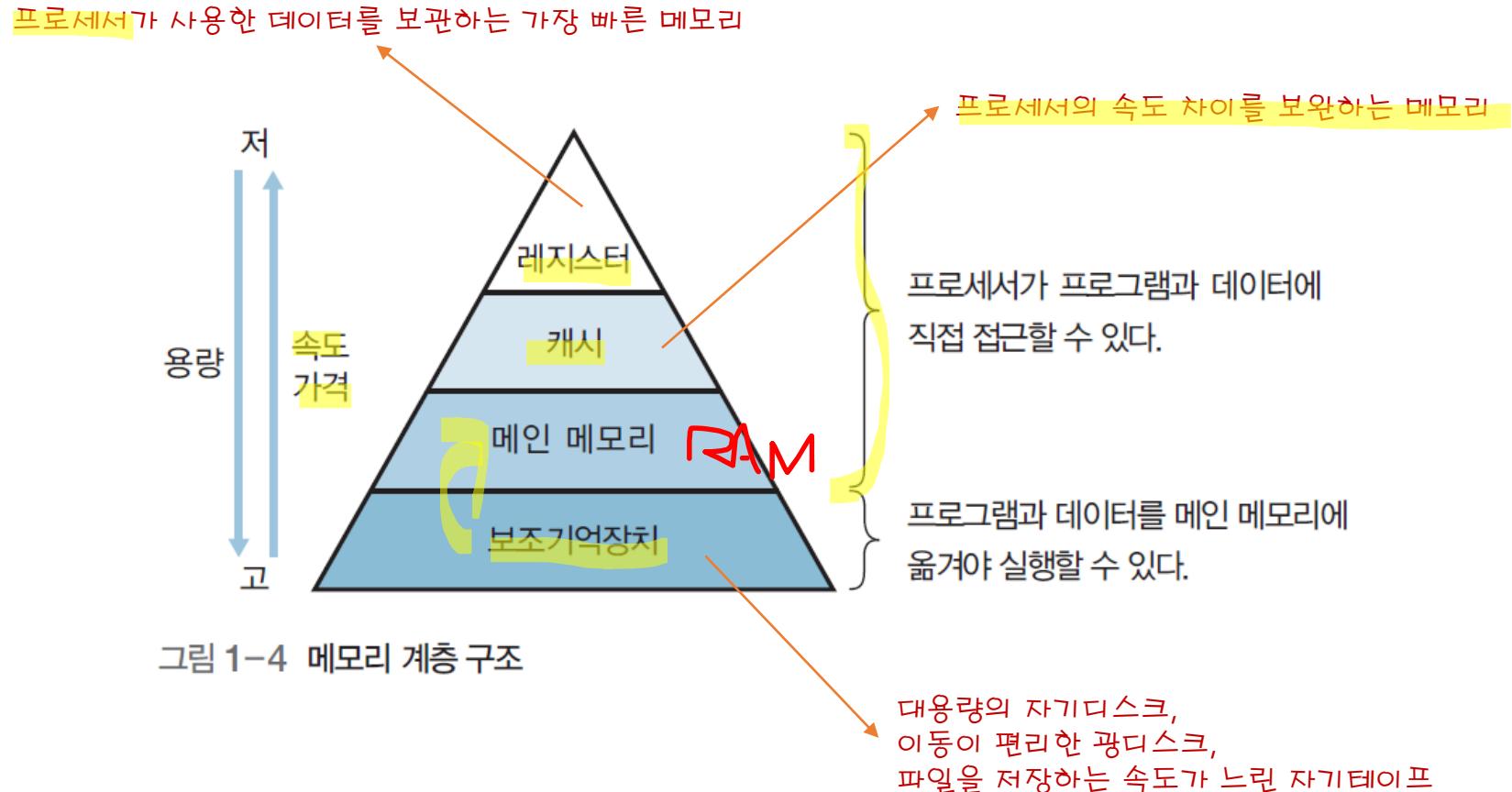


그림 1-4 메모리 계층 구조

2. 메모리

■ 레지스터

- **프로세서 내부에 있으며, 프로세서가 사용할 데이터를 보관하는 가장 빠른 메모리**

■ 메인 메모리

- **프로세서 외부에 있으면서 프로세서에서 수행할 프로그램과 데이터를 저장하거나 프로세서에서 처리한 결과 저장**
- **주기억장치 또는 1차 기억장치라고도 한다.** 저장 밀도가 높고 가격이 싼 DRAM^{Dynamic RAM}을 많이 사용

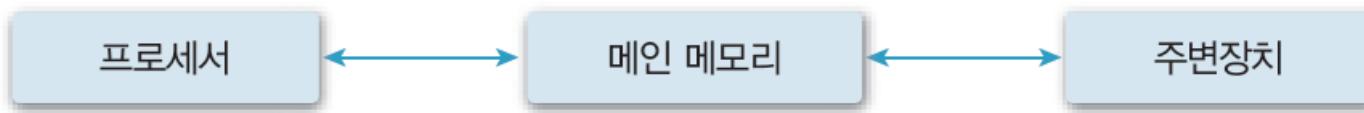


그림 1-5 메인 메모리의 역할 1

2. 메모리

■ 메인 메모리

- 다수의 셀^{cell}로 구성되며, 각 셀은 비트로 구성
- 셀이 K 비트이면 셀에 2^K 값 저장 가능
- 메인 메모리에 데이터를 저장할 때는 셀 한 개나 여러 개에 나눠서 저장
- 셀은 주소로 참조하는데, n 비트이라면 주소 범위는 $0 \sim 2^{n-1}$

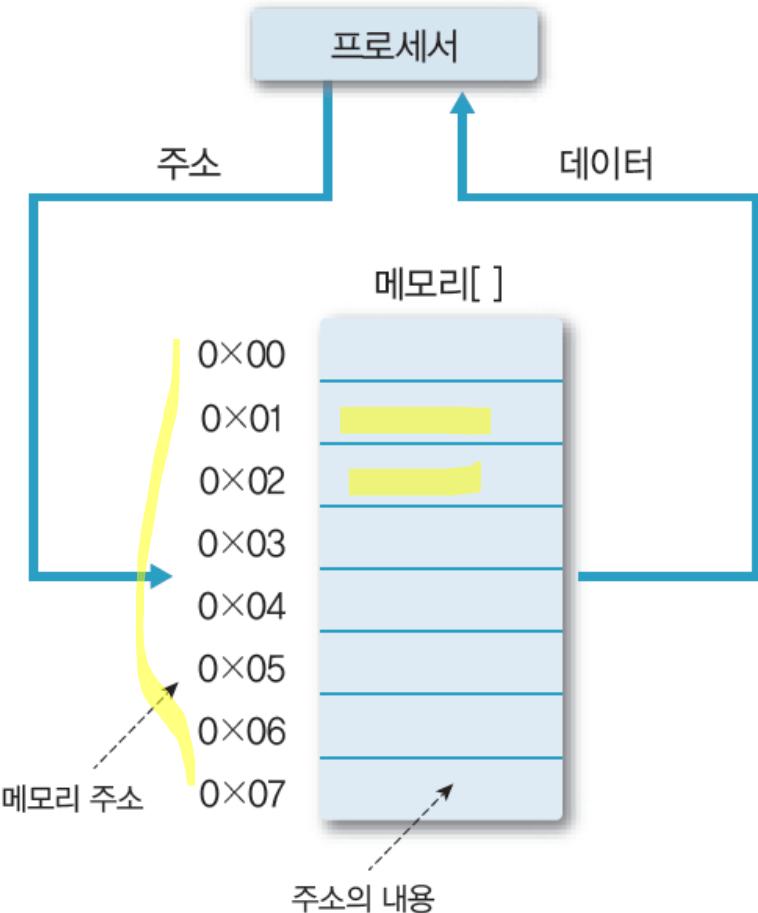


그림 1-6 메인 메모리의 주소 지정

2. 메모리

■ 메인 메모리

- 프로세서와 보조기억장치 사이에 있으며, 여기서 발생하는 디스크 입출력 병목 현상을 해결하는 역할 수행
- 프로세서와 메인 메모리 간에 속도 차이의 부담을 줄이려고 프로세서 내부나 외부에 캐시를 구현하기도 함

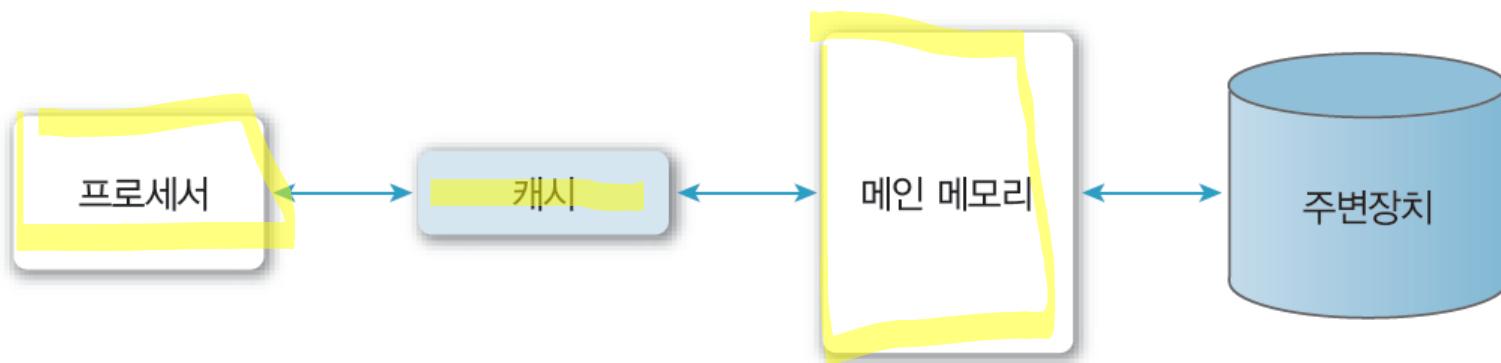


그림 1-9 메인 메모리의 역할 2 : 캐시 추가

2. 메모리

■ 캐시

- 프로세서 내부나 외부에 있으며, 처리 속도가 빠른 프로세서와 상대적으로 느린 메인 메모리의 속도 차이를 보완하는 고속 **버퍼**

- 메인 메모리에서 데이터를 블록 단위로 가져와 프로세서에 워드 단위로 전달하여 속도를 높임
- 데이터가 이동하는 통로(대역폭)를 확대하여 프로세서와 메모리의 속도 차이를 줄임

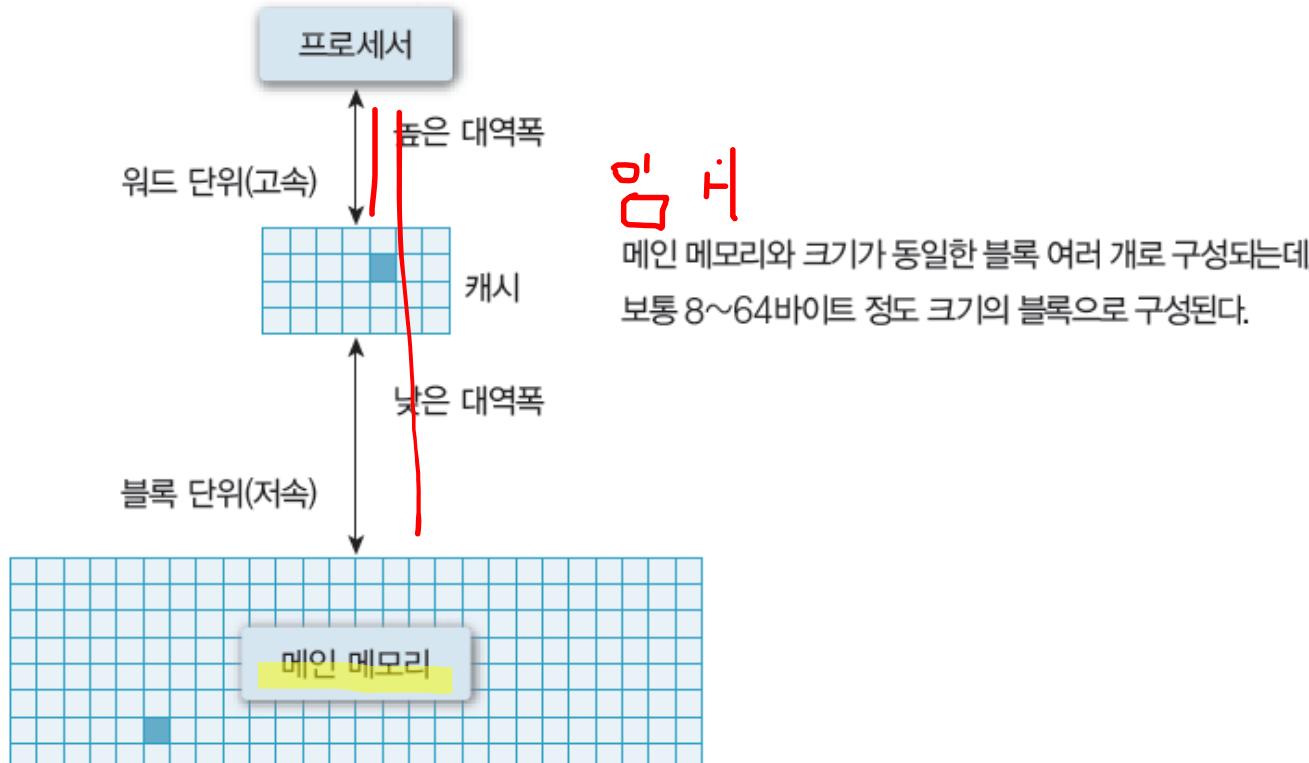


그림 1-10 캐시의 역할

2. 메모리

■ 캐시

- 캐시의 성능은 작은 용량의 캐시에 프로세서가 이후 참조할 정보가 얼마나 들어 있느냐로 좌우됨
 - 캐시 적중^{cache hit} (캐시 히트) : 프로세서가 참조하려는 정보가 있을 때
 - 캐시 실패^{cache miss} (캐시 미스) : 프로세서가 참조하려는 정보가 없을 때
- 블록의 크기는 캐시의 성능으로 좌우되는데, 실제 프로그램을 실행할 때 참조한 메모리에 대한 공간적 지역성(국부성)과 시간적 지역성(국부성)이 있기 때문
 - 공간적 지역성^{spatial locality} : 대부분의 프로그램이 참조한 주소와 인접한 주소의 내용을 다시 참조하는 특성
 - 시간적 지역성^{temporal locality} : 한 번 참조한 주소를 곧 다시 참조하는 특성

2. 메모리

■ 공간적 지역성

- 참조한 주소와 인접한 주소를 참조할 가능성이 높다
 - i.e., 프로그램을 순차적으로 수행

■ 시간적 지역성

- 한 번 참조한 주소를 곧 다시 참조하는 특성
 - i.e., for, while states

The screenshot shows a C++ code editor interface with the following code:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     int a = 1;
7     int b = 2;
8     int* pa = &a;
9     int* pb = &b;
10
11    cout << "a의 값: " << a << endl;
12    cout << "b의 값: " << b << endl;
13    cout << "pa의 값: " << pa << endl;
14    cout << "pb의 값: " << pb << endl;
15
16 }
```

The code defines two integer variables `a` and `b`, and their pointers `pa` and `pb`. It then prints the values of `a`, `b`, `pa`, and `pb` using `cout`.

Input	Output	Stderr	Compilation	Execution
1	a의 값 : 1			
2	b의 값 : 2			
3	pa의 값: 0x7ffc6d479e10			
4	pb의 값: 0x7ffc6d479e14			

2. 메모리

■ 캐시

▪ 공간적 지역성과 시간적 지역성의 발생 원인

- 프로그램이 명령어를 순차적으로 실행하는 경향이 있어 명령어가 특정 지역 메모리에 인접해 있다.
- 순환(단일 순환, 중첩 순환 등) 때문에 프로그램을 반복하더라도 메모리는 일부 영역만 참조한다.
- 대부분의 컴파일러를 메모리에 인접한 블록에 배열로 저장한다. 따라서 프로그램이 배열 원소에 순차적으로 자주 접근하므로 지역적인 배열 접근 경향이 있다.

→ 지역성은 블록이 크면 캐시의 히트율이 올라갈 수 있음을 의미하지만, 블록이 커지면 이에 따른 전송 부담과 캐시 데이터 교체 작업이 자주 일어나므로 블록 크기를 무작정 늘릴 수는 없음

2. 메모리

■ 보조기억장치

- 주변장치 중 프로그램과 데이터를 저장하는 하드웨어
- 2차 기억장치 또는 외부기억장치라고도 함
- 자기디스크, 광디스크, 자기테이프 등이 있음

3. 시스템 버스

■ 시스템 버스 system bus

- 하드웨어를 물리적으로 연결하여 서로 데이터를 주고받을 수 있게 하는 통로
- 컴퓨터 내부의 다양한 신호(데이터 입출력 신호, 프로세서 상태 신호, 인터럽트 요구와 허가 신호, 클록 clock 신호 등)를 시스템 버스로 전달
- 기능에 따라 데이터 버스, 주소 버스, 제어 버스로 구분

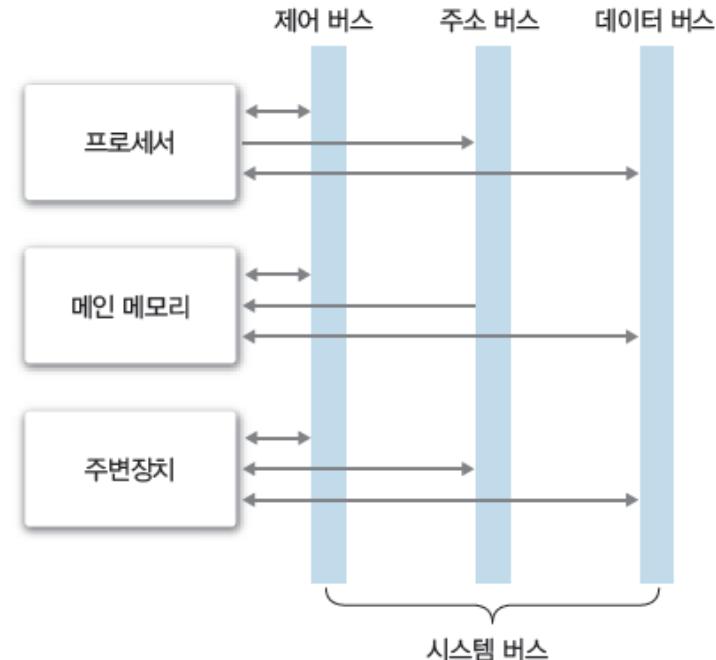
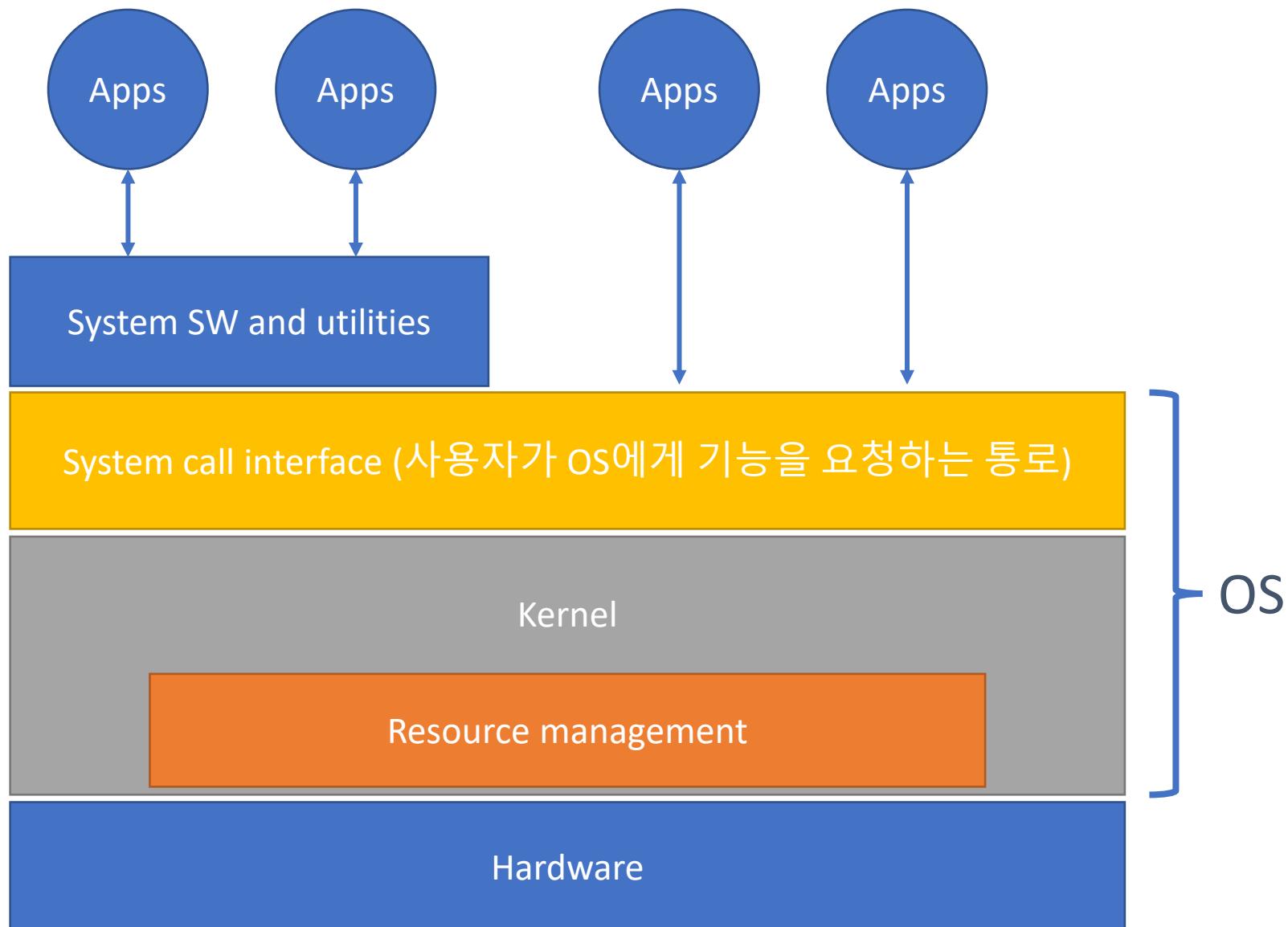


표 1-3 시스템 버스의 종류

종류	설명
데이터 버스	프로세서와 메인 메모리, 주변장치 사이에서 데이터를 전송한다. 데이터 버스를 구성하는 배선 수는 프로세서가 한 번에 전송할 수 있는 비트 수를 결정하는데, 이를 워드라고 한다.
주소 버스	프로세서가 시스템의 구성 요소를 식별하는 주소 정보를 전송한다. 주소 버스를 구성하는 배선 수는 프로세서와 접속할 수 있는 메인 메모리의 최대 용량을 결정한다.
제어 버스	프로세서가 시스템의 구성 요소를 제어하는 데 사용한다. 제어 신호로 연산장치의 연산 종류와 메인 메모리의 읽기나 쓰기 동작을 결정한다.

3. 컴퓨터 시스템의 구성



1. 자원관리

■ 프로세스?

- 커널에 등록된 실행단위 (실행중인 프로그램)
- 프로그램은 일반적으로 하드디스크 등에 저장되어 있는 실행코드를 뜻하고, 프로세스는 프로그램을 구동하여 프로그램 자체와 프로그램의 상태가 메모리 상에서 실행되는 작업 단위를 지칭
 - 하나의 프로그램을 여러 번 구동하면 여러 개의 프로세스가 메모리 상에서 실행된다.

1. 자원관리

활성 상태 보기
모든 프로세스

(x) (i) (...) CPU 메모리 에너지 디스크 네트워크 검색

프로세스 이름	% CPU	CPU 시간	스레드	대기 상태 깨움	% GPU	GPU 시간	PID	사용자
WindowServer	8.0	1:27:07.96	13	26	0.0	43:25.44	144	_windowserver
kernel_task	25.8	55:57.42	184	729	0.0	0.00	0	root
Dropbox	0.3	8:52.45	134	8	0.0	0.00	1041	crazytb
fseventsd	0.4	4:27.27	13	12	0.0	0.00	74	root
gamecontrollerd	0.1	3:48.90	3	0	0.0	0.00	1267	_gamecontroller
nosmain	0.5	3:40.55	7	4	0.0	0.00	1585	root
Finder	0.4	3:20.19	11	0	0.0	0.02	943	crazytb
iconservicesagent	0.0	3:12.97	2	0	0.0	4.08	973	crazytb
coreaudiod	1.3	3:09.19	8	0	0.0	0.00	171	_coreaudiod
launchd	0.4	2:40.17	3	0	0.0	0.00	1	root
IMDPersistenceAgent	0.0	2:36.99	2	0	0.0	0.00	1048	crazytb
mds_stores	0.1	2:31.61	8	0	0.0	0.00	307	root
Be Focused	0.3	2:19.83	4	8	0.0	0.00	1191	crazytb
syspolicyd	0.0	2:18.34	3	0	0.0	0.00	179	root
Microsoft PowerPoint	0.0	2:15.06	15	1	0.0	7.08	24812	crazytb

시스템:	13.53%
사용자:	12.74%
대기:	73.73%

CPU 로드	
--------	---

스레드:	2,053
프로세스:	496

1. 프로세스의 개념

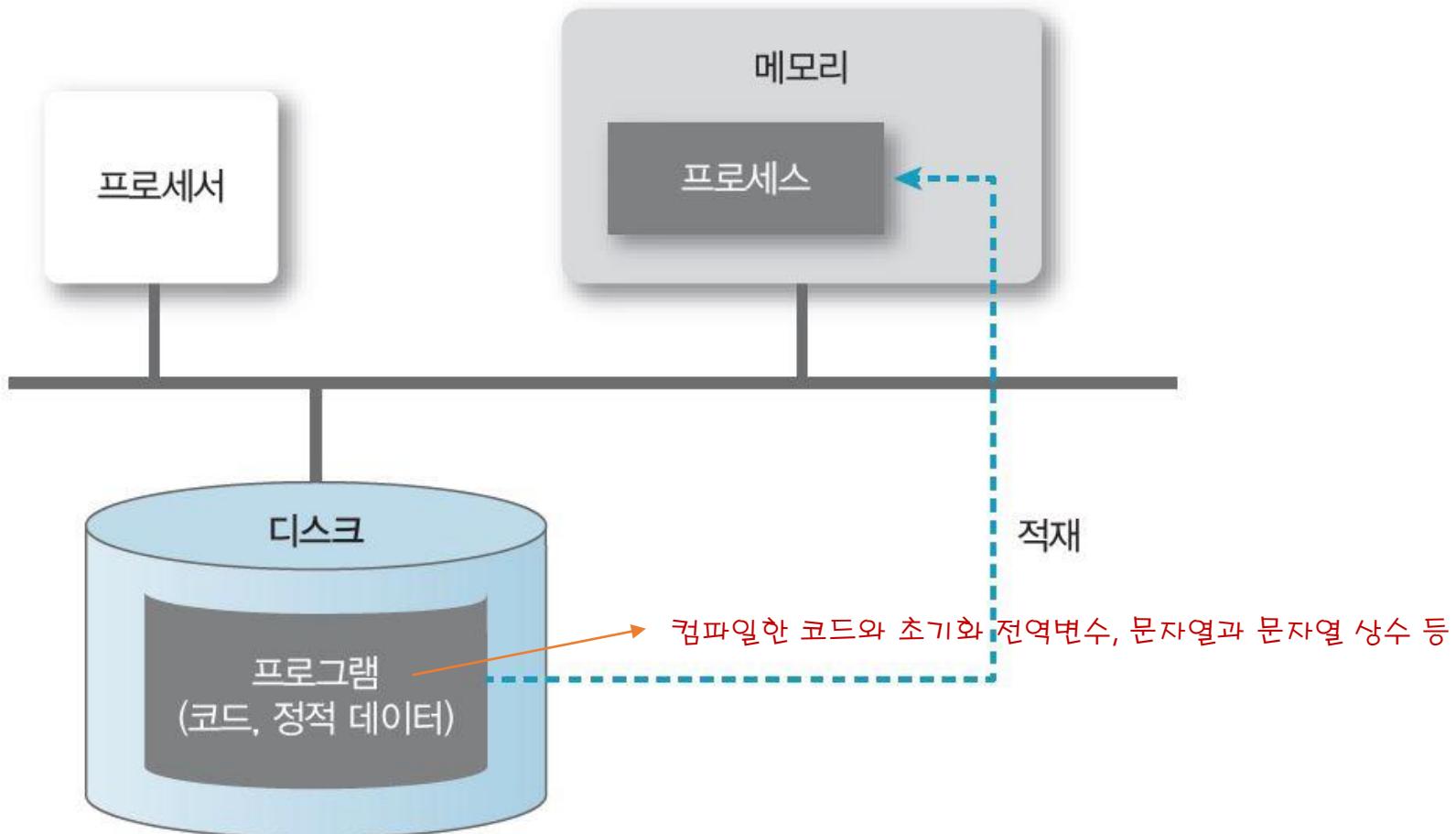


그림 3-1 프로그램과 프로세스 : 프로그램이 메모리로 적재되면 프로세스가 됨

5. 프로세스의 중단과 재시작

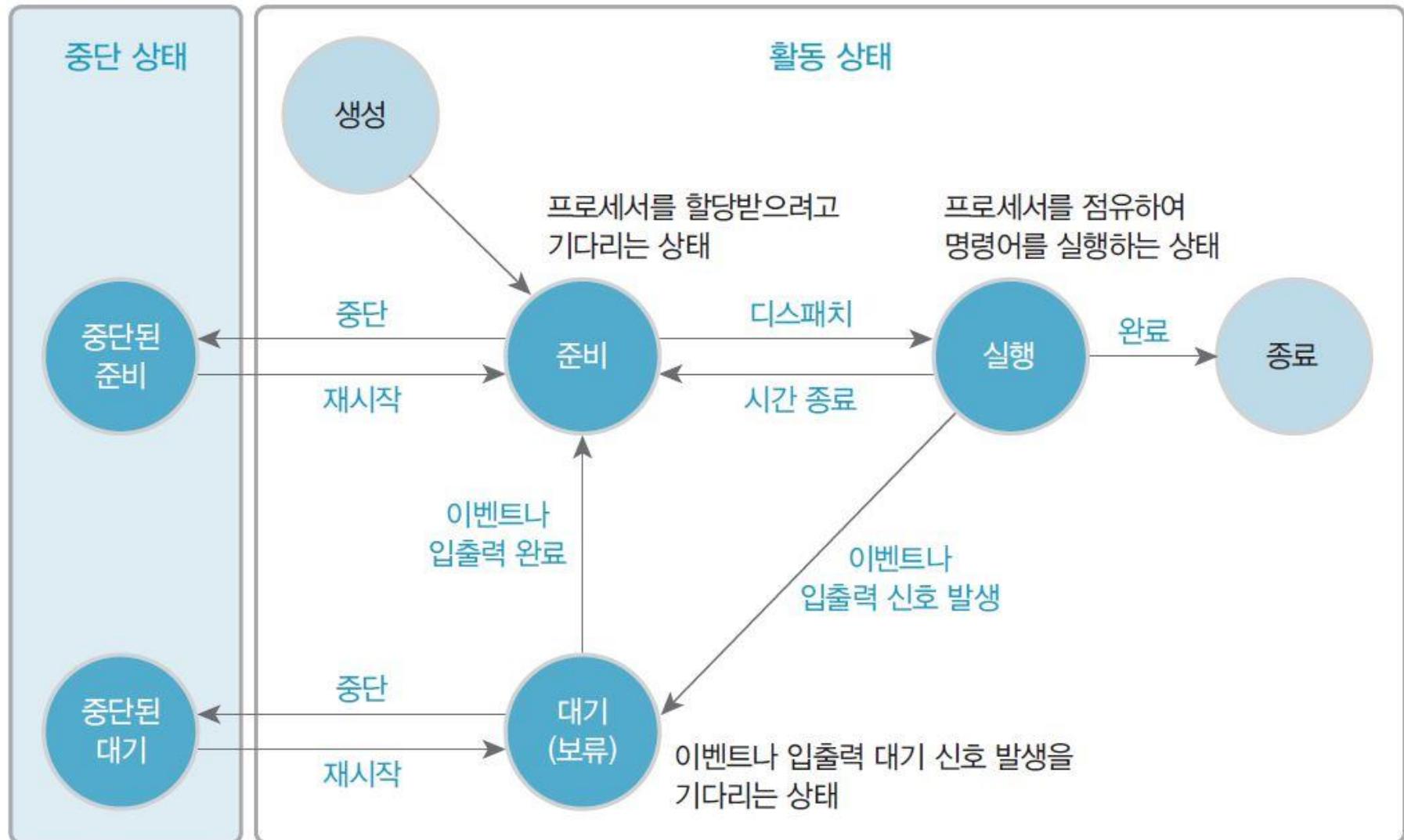


그림 3-10 중단과 재시작을 추가한 프로세스의 상태 변화

6. 프로세스의 우선 순위

■ 프로세스 스케줄러

- 프로세스 제어 블록의 우선순위 이용하여 준비 리스트의 프로세스 처리
- 준비 리스트의 프로세스는 프로세서 중심 프로세스와 입출력 중심 프로세스로 구분
- 입출력 중심 프로세스는 속도가 느리면서 빠른 응답 요구하는 단말기 입출력 프로세스에 높은 우선순위 부여, 속도가 빠른 디스크 입출력 프로세스에는 낮은 우선순위 부여
- 우선순위가 낮은 프로세스에는 시간을 많이, 우선순위가 높은 프로세스에는 적게 할당
- 입출력 중심 프로세스는 프로세서를 짧게 자주 사용하도록 하고, 프로세서 중심 프로세스 프로세서를 길게 사용하되 사용 횟수를 줄여 균형 유지

7. 프로세스의 문맥 교환

■ 프로세스의 문맥 교환 발생

- 실행 중인 프로세스에 인터럽트가 발생하면 운영체제가 다른 프로세스를 실행 상태로 바꾸고
제어를 넘겨주어 프로세스 문맥 교환 발생
- 인터럽트 발생 : 현재 실행하는 프로세스와 별도로 외부에서 이벤트(예 : 입출력 동작의 종료) 발생시
- 인터럽트 유형에 따른 루틴 분기
 - 입출력 인터럽트 : 입출력 동작이 발생 확인, 이벤트 기다리는 프로세스를 준비 상태로 바꾼 후 실행할 프로세스 결정
 - 클록 인터럽트 : 실행 중인 프로세스 할당 시간 조사, 준비 상태로 바꾸고, 다른 프로세스를 실행 상태로 전환
- 인터럽트는 인터럽트 처리 루틴을 실행한 후 현재 실행 중인 프로세스를 재실행할 수 있으므로 인터럽트가 곧 프로세스 문맥 교환으로 발전하지는 않음
- 대개 이전 프로세스의 상태 레지스터 내용 보관하고 다른 프로세스의 레지스터 적재하여 프로세스를 교환하는데, 이런 일련의 과정을 문맥 교환^{context switching}이라고 한다. 문맥 교환에서는 오버헤드가 발생하는데, 이는 메모리 속도, 레지스터 수, 특수 명령어의 유무에 따라 다르다. 그리고 문맥 교환은 프로세스가 '준비 → 실행' 상태로 바뀌거나 '실행 → 준비' 또는 '실행 → 대기' 상태로 바뀔 때 발생한다. 그 예는 [그림3 -11]과 같다.

7. 프로세스의 문맥 교환

■ 문맥 교환 context switching

- 이전 프로세스의 상태 레지스터 내용 보관하고 다른 프로세스의 레지스터 적재하여 프로세스를 교환하는 일련의 과정
- 오버헤드 발생, 이는 메모리 속도, 레지스터 수, 특수 명령어의 유무에 따라 다름
- 오버헤드는 시간 비용 소요되어 운영체제 설계 시 불필요한 문맥 교환 감소가 주요 목표
- 레지스터 문맥 교환, 작업 문맥 교환, 스레드 문맥 교환, 프로세스 문맥 교환 가능

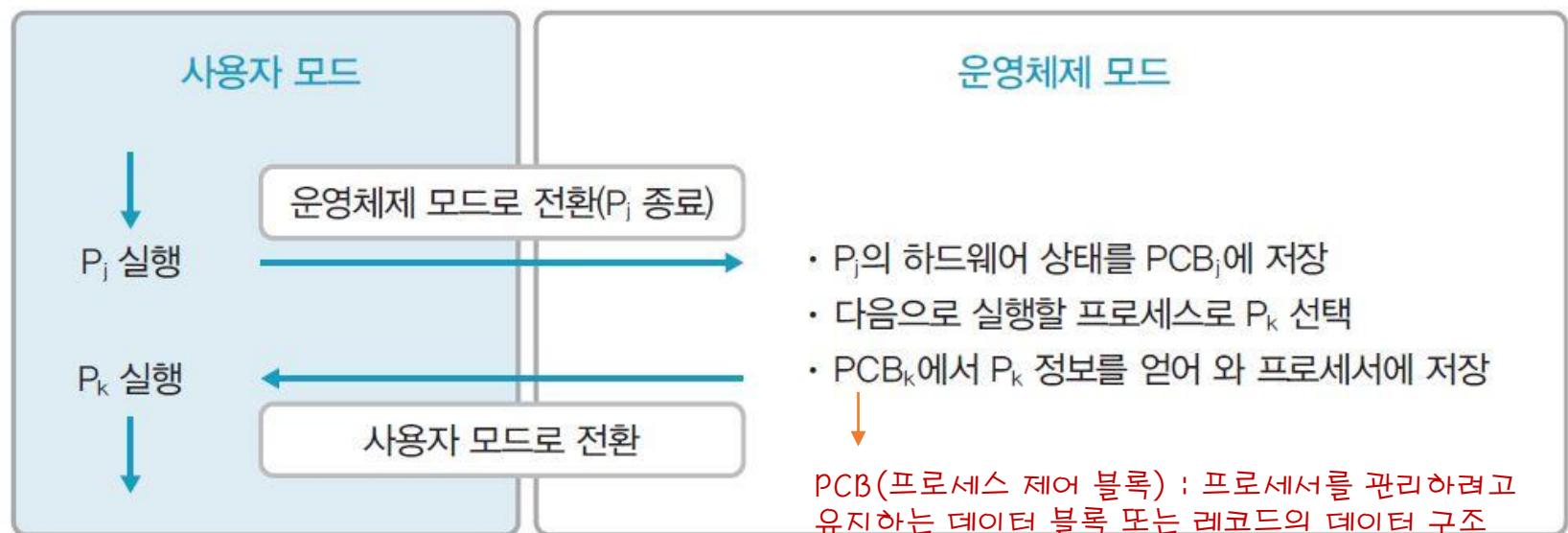


그림 3-11 문맥 교환 예

Section 03 스레드의 개념과 상태 변화(1. 스레드의 개념)

■ 스레드^{thread}의 개념

- 프로세스의 특성인 자원과 제어에서 제어만 분리한 실행 단위
- 프로세스 하나는 스레드 한 개 이상으로 나눌 수 있음
- 프로세스의 직접 실행 정보를 제외한 나머지 프로세스 관리 정보 공유
- 다른 프로시저 호출, 다른 실행 기록(별도 스택 필요)
- 관련 자원과 함께 메모리 공유 가능하므로 손상된 데이터나 스레드의 이상 동작 고려
- 경량 프로세스^{LWP, Light Weight Process}: 프로세스의 속성 중 일부가 들어 있는 것
- 중량 프로세스^{HWP, Heavy Weight Process}: 스레드 하나에 프로세스 하나인 전통적인 경우
- 같은 프로세스의 스레드들은 동일한 주소 공간 공유

1. 스레드의 개념



- SP^{Stack Pointer} : 스택 포인터
- SR^{Sequence Register} : 순서열 레지스터
- PC^{Program Counter} : 프로그램 카운터

그림 3-12 스레드의 구조

1. 스레드의 개념

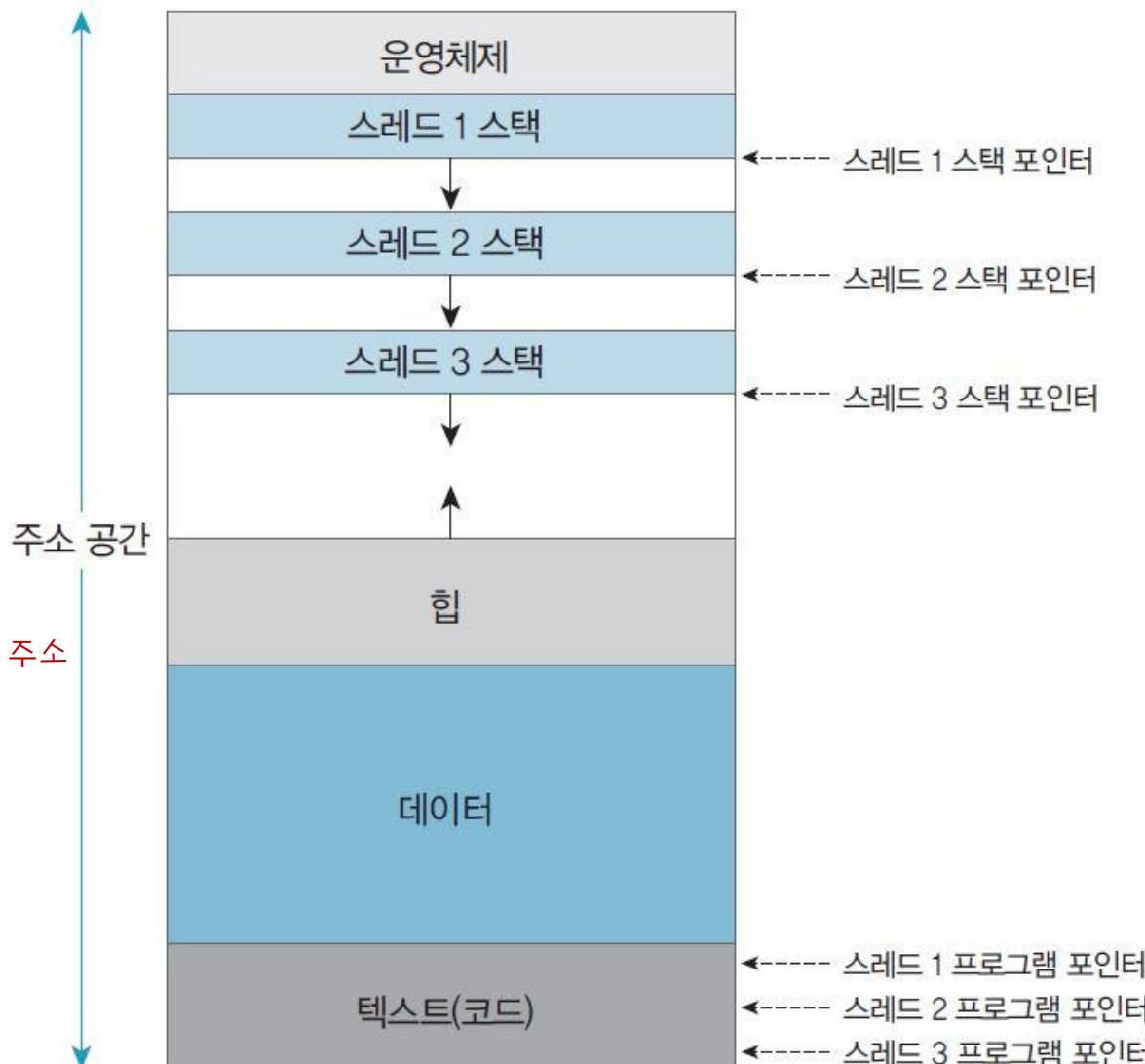


그림 3-13 스레드의 주소 공간

1. 스레드의 개념

■ 스레드 병렬 수행

- 프로세스 하나에 포함된 스레드들은 공동의 목적 달성을 위해 병렬 수행
- 프로세스가 하나인 서로 다른 프로세서에서 프로그램의 다른 부분 동시 실행

■ 스레드 병렬 수행의 이점

- 사용자 응답성 증가
- 프로세스의 자원과 메모리 공유 가능
- 경제성 좋음
- 다중 처리(멀티 프로세싱)로 성능과 효율 향상

3. 스레드의 사용 예

- 스레드를 이용하여 프로그램의 비동기적 요소를 구현한 예

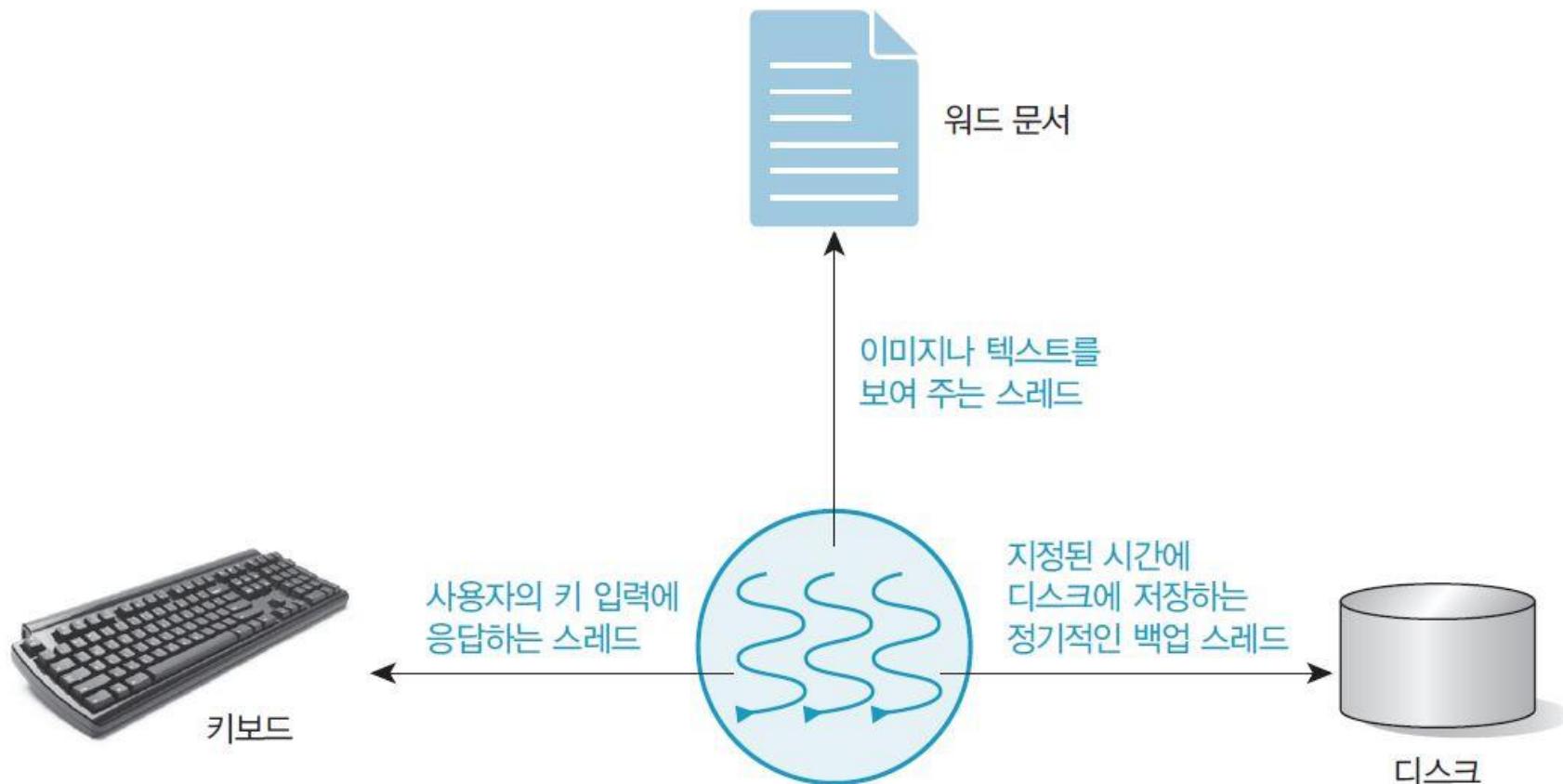


그림 3-16 다중 스레드를 이용한 워드 편집기 프로세스

1. 사용자 수준 스레드

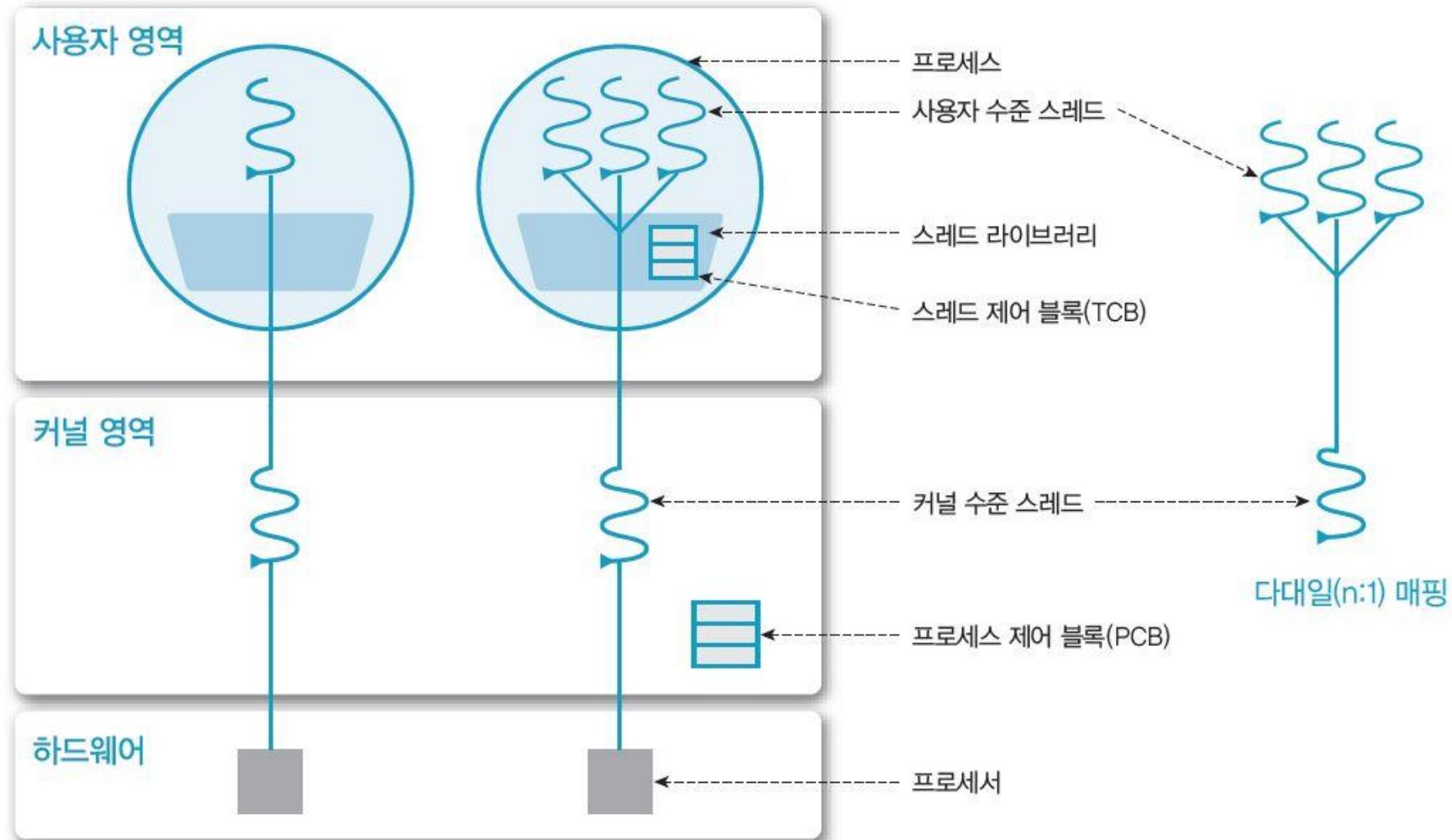


그림 3-20 사용자 수준 스레드 : 다대일($n : 1$) 매핑

1. 사용자 수준 스레드

■ 사용자 수준 스레드의 장점

- 이식성 높음 : 커널에 독립적 스케줄링을 할 수 있어 모든 운영체제에 적용
- 오버헤드 적음 : 스케줄링, 동기화 위해 커널 호출 않으므로 커널 영역 전환 오버헤드 감소
- 유연한 스케줄링 가능 : 커널이 아닌 스레드 라이브러리에서 스레드 스케줄링 제어하므로 응용 프로그램에 맞게 스케줄링 가능

■ 사용자 수준 스레드의 단점

- 시스템의 동시성 지원하지 않음 : 스레드가 아닌 프로세스 단위로 프로세서 할당하여 다중 처리 환경을 갖춰도 스레드 단위로 다중 처리 불가능
동일한 프로세스의 스레드 한 개가 대기 상태가 되면
이 중 어떤 스레드도 실행 불가
- 확장 제약이 따름 : 커널이 한 프로세스에 속한 여러 스레드에 프로세서를 동시에 할당 할 수 없어 다중 처리 시스템에서 규모 확장 곤란
- 스레드 간 보호 불가능 : 스레드 간 보호에 커널의 보호 방법 사용 불가
스레드 라이브러리에서 스레드 간 보호를 제공해야
프로세스 수준에서 보호 가능

2. 커널 수준 스레드

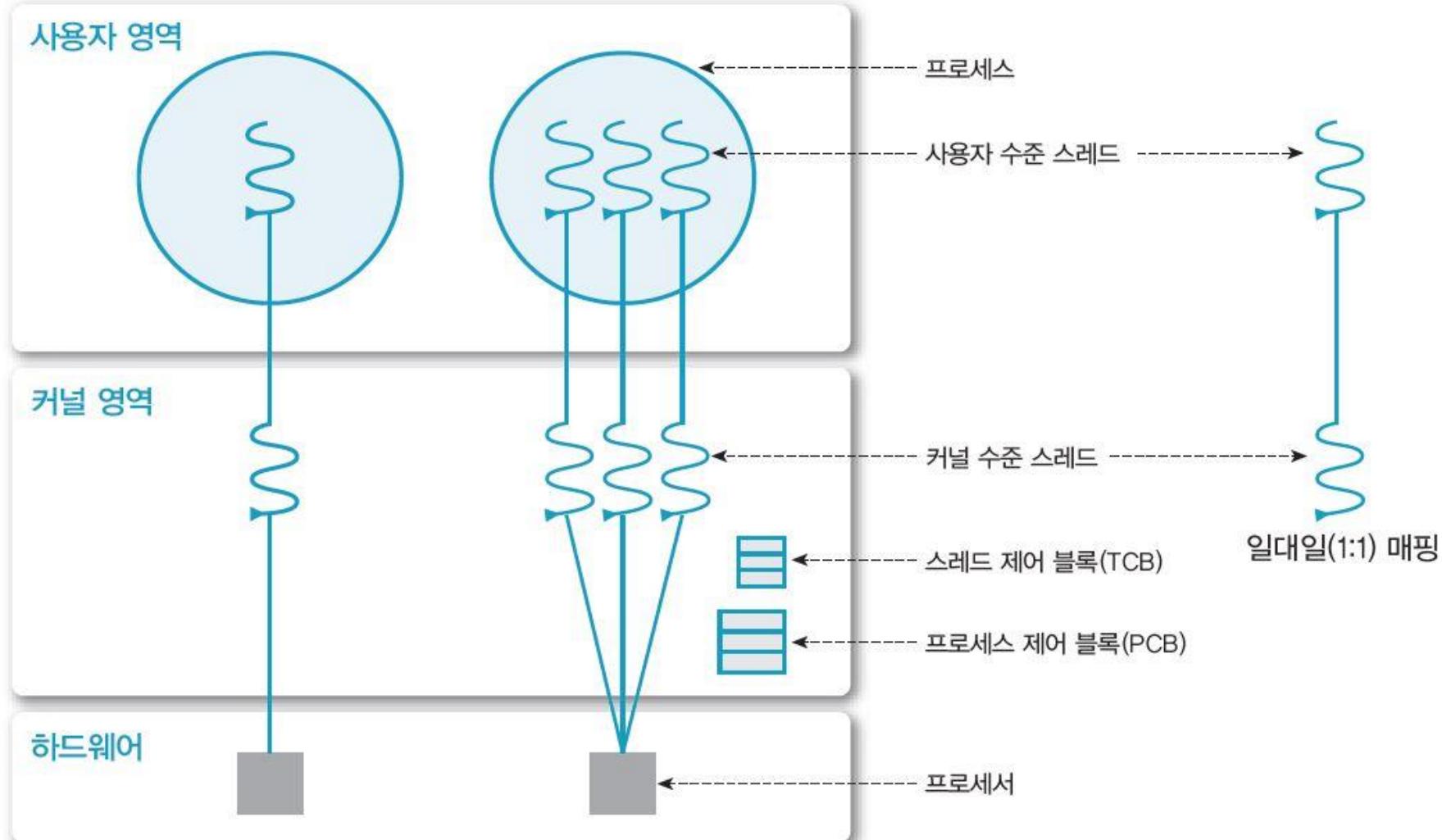


그림 3-21 커널 수준 스레드 : 일대일(1:1) 스레드 매핑

2. 커널 수준 스레드

■ 커널 수준 스레드의 개념

- 사용자 수준 스레드의 한계 극복 방법으로, 커널이 스레드와 관련된 모든 작업 관리
- 한 프로세스에서 다수의 스레드가 프로세서 할당받아 병행 수행, 스레드 한 개가 대기
- 상태가 되면 동일한 프로세스에 속한 다른 스레드로 교환 가능. 이때도 커널 개입하므로
- 사용자 영역에서 커널 영역으로 전환 필요
- 커널이 직접 스케줄링하고 실행하기에 사용자 수준 스레드의 커널 지원 부족 문제 해결
- 커널이 전체 프로세스와 스레드 정보를 유지하여 오버헤드가 커짐
- 커널이 각 스레드 개별적으로 관리하여 동일한 프로세스의 스레드 병행 수행
- 동일한 프로세스에 있는 스레드 중 한 개가 대기 상태가 되도 다른 스레드 실행 가능
- 스케줄링과 동기화를 하려면 더 많은 자원 필요
- 윈도우 NT·XP·2000, 리눅스, 솔라리스 9 이상 버전의 운영체제가 대표적

Section 01 스케줄링의 이해(1. 스케줄링의 개념)

■ 스케줄링 scheduling의 개념

- 여러 프로세스가 번갈아 사용하는 자원을 어떤 시점에 어떤 프로세스에 할당할지 결정
- 자원이 프로세서인 경우를 프로세서 스케줄링, 대부분의 스케줄링이 프로세서 스케줄링의 미
- 스케줄링 방법에 따라 프로세서를 할당받을 프로세스 결정하므로 스케줄링이 시스템의 성능에 영향 미침
- 좋은 스케줄링은 프로세서 효율성 높이고, 작업(프로세스)의 응답시간 최소화하여 시스템의 작업 처리 능력 향상
- 스케줄링이 필요 없는 프로세스(인터럽트 처리, 오류 처리, 사용자의 시스템 호출 등)의 사전 처리가 대표적
- 반면에 스케줄링이 필요한 프로세스에는 사용자 프로세스와 시스템 호출로 발생하는 시스템 프로세스가 있음

3. 스케줄링의 기준 요소

- 프로세스 버스트가 긴 예와 짧은 예

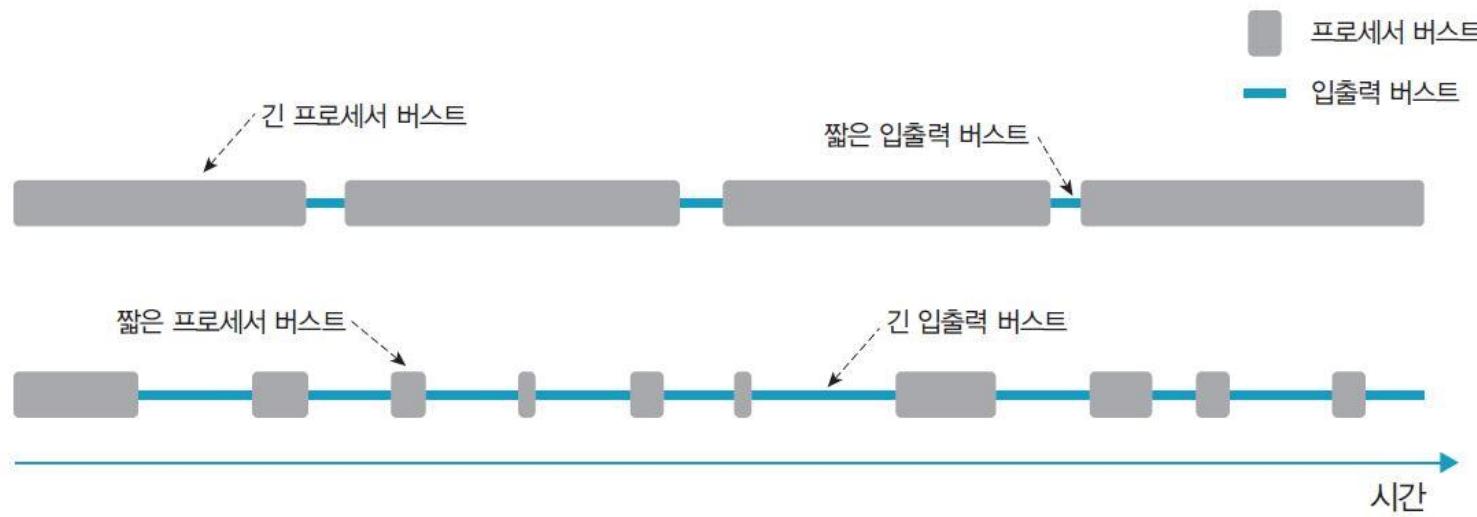


그림 6-3 프로세서 버스트가 긴 예와 짧은 예

4. 스케줄링의 단계

■ 스케줄링 수행 단계

- 1단계 작업 스케줄링 : 작업 선택
 - 실제로 시스템 자원을 사용할 작업 결정하는 작업 스케줄링. 승인 스케줄링이라고도 함
 - 작업 스케줄링에 따라 작업 프로세스들로 나눠 생성. 수행 빈도가 적어 장기 스케줄링에 해당
- 2단계 작업 승인과 프로세서 결정 스케줄링 : 사용 권한 부여
 - 프로세서 사용 권한 부여할 프로세스 결정하는 작업 승인과 프로세서 할당 스케줄링
 - 시스템의 오버헤드에 따라 연기할 프로세스 잠정적으로 결정. 1단계 작업 스케줄링과 3단계 프로세서 할당 스케줄링의 완충 역할. 수행 빈도를 기준으로 하면 중기 스케줄링에 해당, 메모리 사용성도 높이고 작업 효율성 향상시키는 스와핑(swapping)(교체) 기능의 일부로 이해 가능
- 3단계 프로세서 할당 스케줄링 : 준비 상태의 프로세스에 프로세서 할당(디스패치)
 - 디스패처(분배기)가 준비 상태에 있는 프로세스 중에서 프로세서 할당할 프로세스 결정하는 프로세스 할당 스케줄링. 단기 스케줄링에 해당

4. 스케줄링의 단계

■ 스케줄링 단계

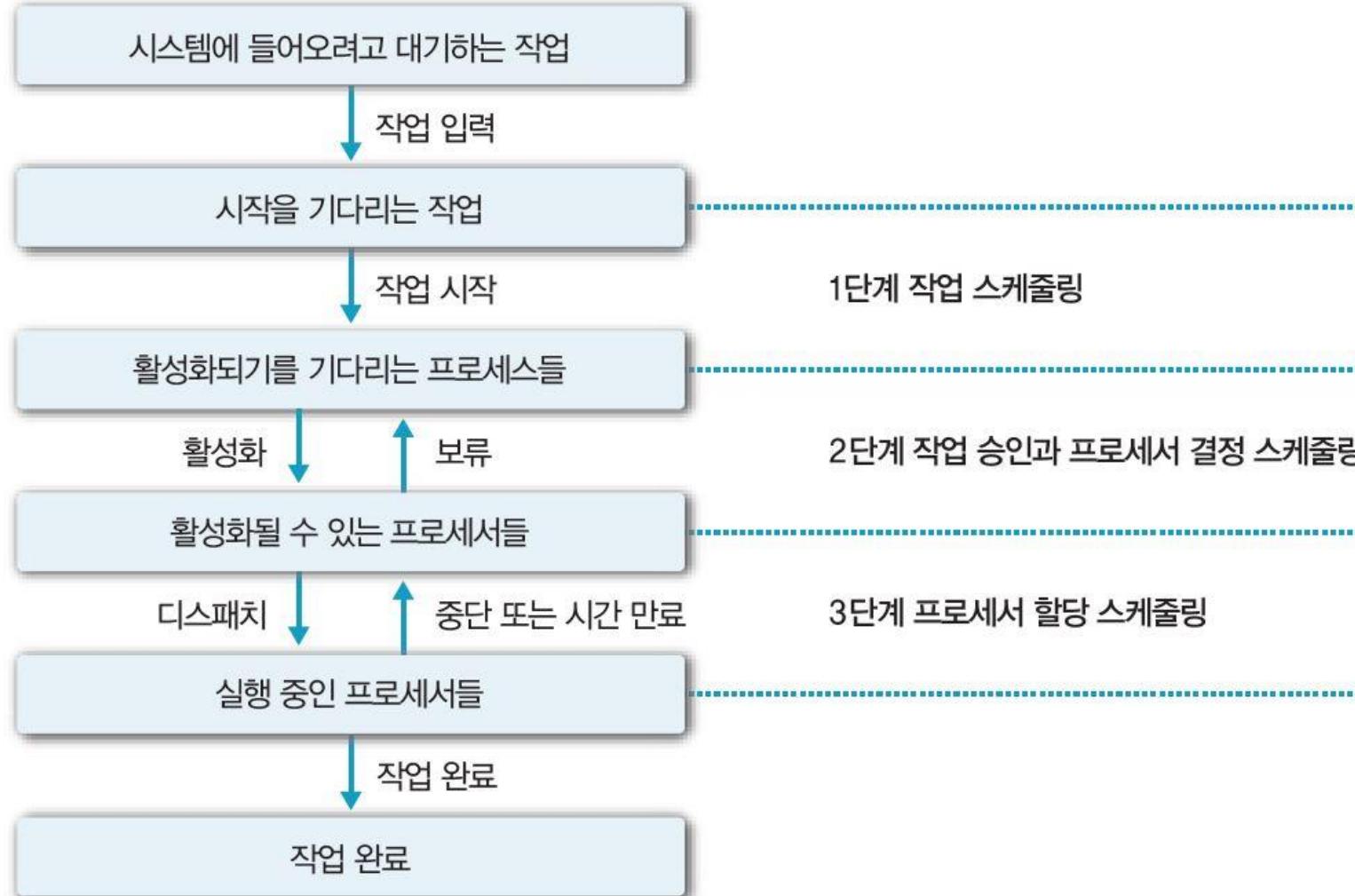


그림 6-4 스케줄링 단계

6. 스케줄링과 스케줄러

- 프로세스 상태 변화와 스케줄러의 역할

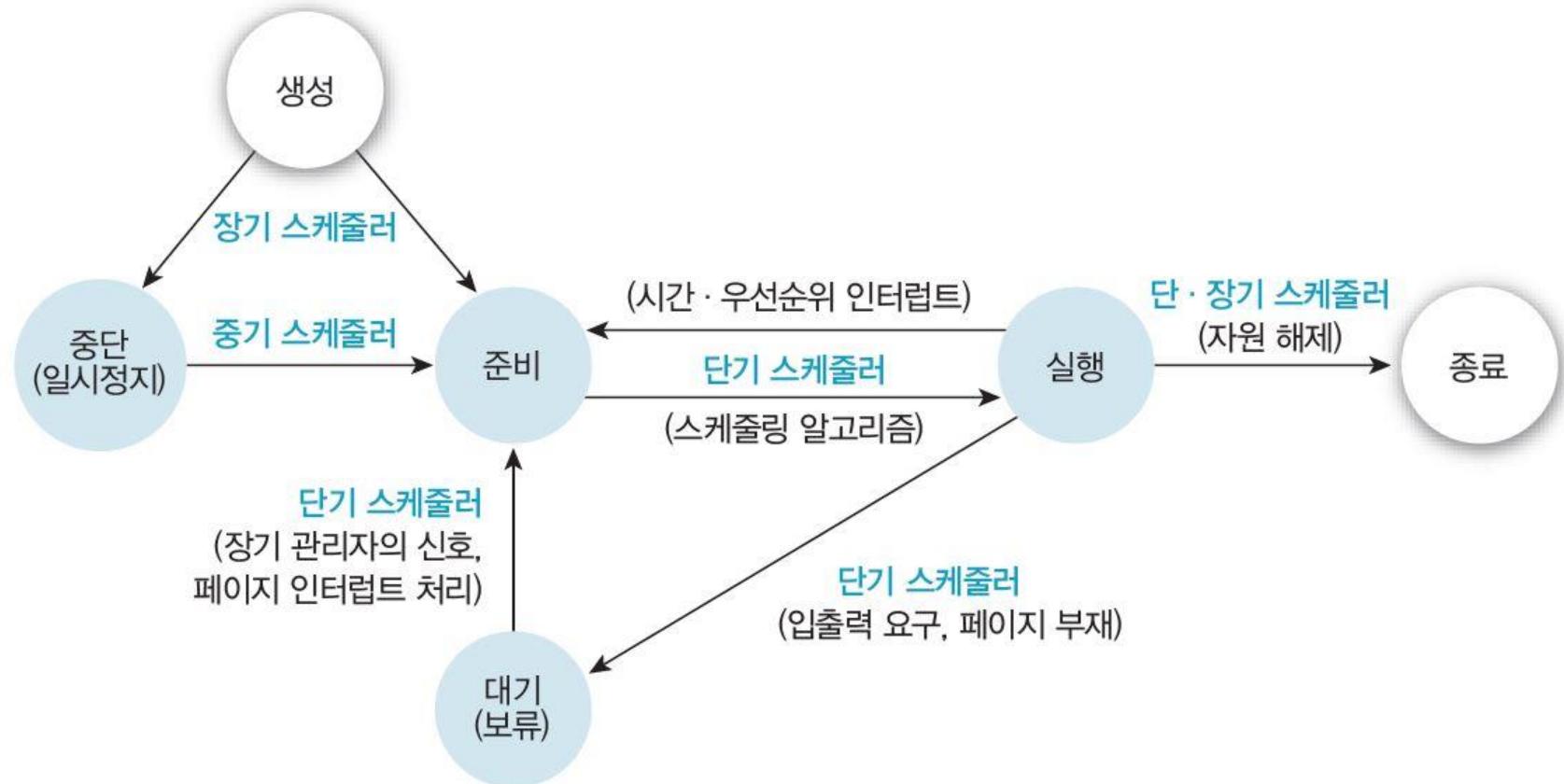


그림 6-12 프로세스 상태 변화와 스케줄러의 역할

7. 선점 스케줄링과 비선점 스케줄링

■ 선점 스케줄링

- 프로세스 하나가 장시간 동안 프로세서 독점 방지하여 모든 프로세스에 프로세서를 서비스 할 기회 늘림. 따라서 우선순위가 높은 프로세스들이 긴급 처리 요청할 때 유용
- 실시간 시스템에서 인터럽트를 받아들이지 않으면 결과는 예측 불가
- 대화식 시분할 시스템이나 실시간 시스템에서 빠른 응답시간 유지 위해 필수
- 오버헤드가 커질 수 있어 효과적인 이용을 위해서는 메모리에 프로세스가 많은 적재 필요
- 프로세서를 사용 가능할 때마다 실행할 수 있는 프로세스들이 준비 상태에 있어야 효과적
- 우선순위라는 개념을 반드시 고려, 우선순위는 의미 있게 부여하지 않으면 효과 없음

■ 비선점 스케줄링

- 실행 시간이 짧은 프로세스(작업)가 실행 시간이 긴 프로세스(작업)를 기다리는 대신 모든 프로세서 공정 관리
- 우선순위가 높은 프로세스 중간에 입력해도 대기 중인 프로세스는 영향을 받지 않으므로 응답시간 예측 용이

8. 스케줄링 알고리즘의 선점 기준

■ 반환시간, 대기시간, 반응시간의 관계

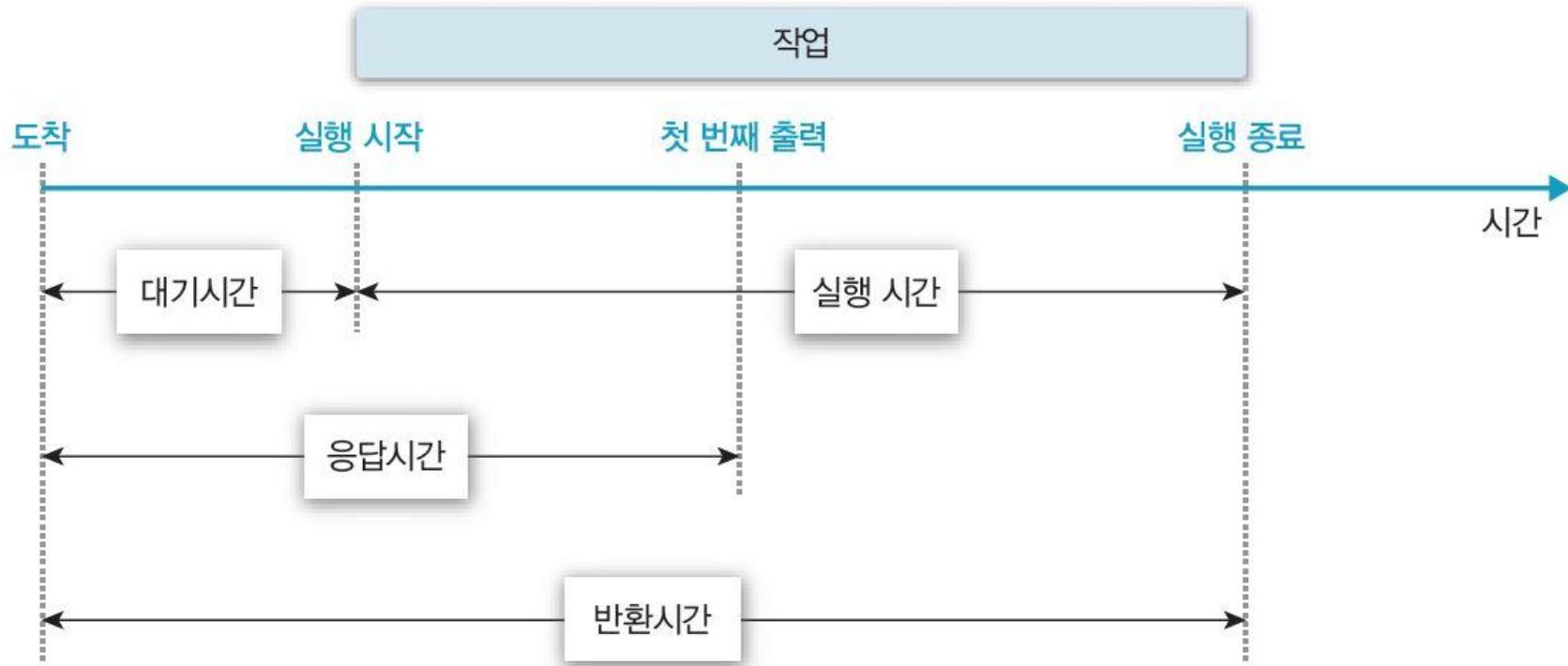


그림 6-13 반환시간, 대기시간, 응답시간의 관계

8. 스케줄링 알고리즘의 선점 기준

■ 프로세스의 반환시간과 대기시간

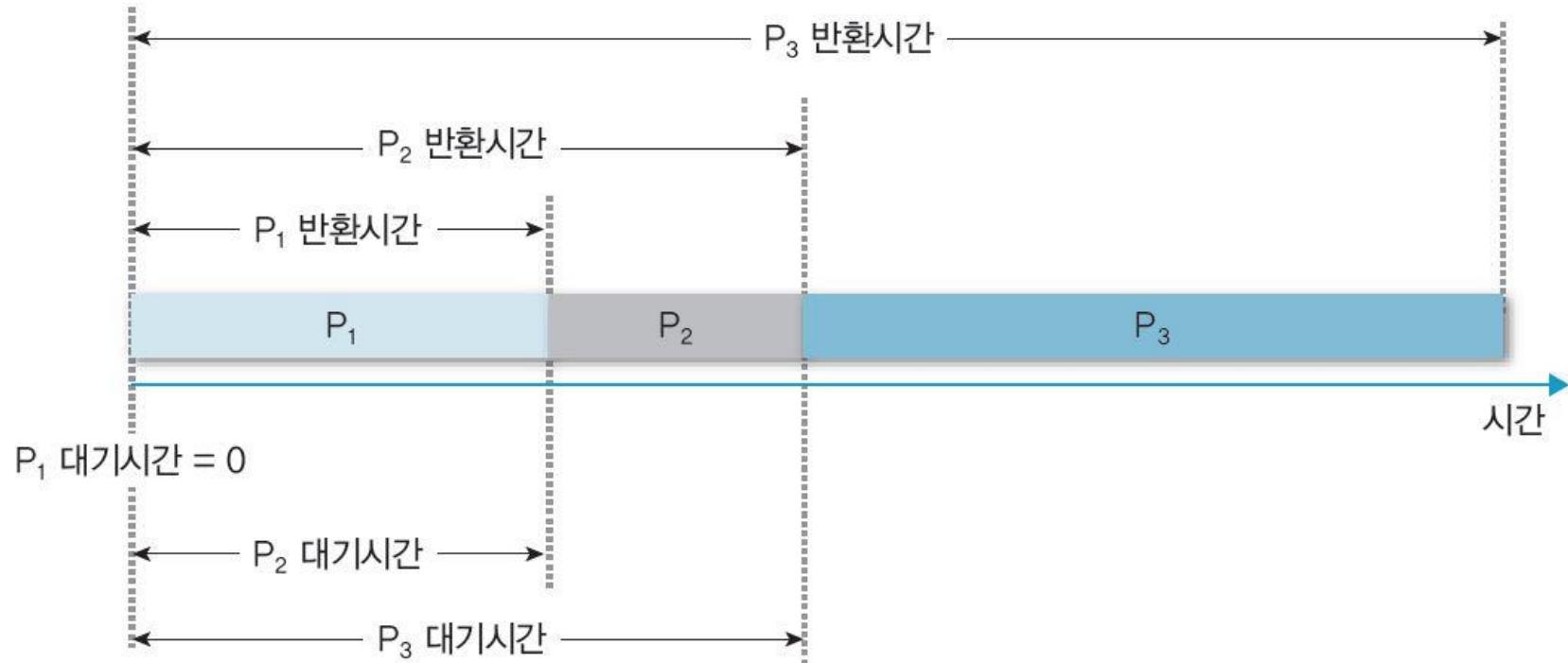


그림 6-14 프로세스 P₁, P₂, P₃의 반환시간, 대기시간 프로세스 3개가 동시에 도착하여 P1, P2, P3 순으로 실행

1. 선입선처리 스케줄링

■ 선입선처리 스케줄링의 예 1

프로세스	도착 시간	실행 시간
P ₁	0	10
P ₂	1	28
P ₃	2	6
P ₄	3	4
P ₅	4	14

(a) 준비 큐



(b) 간트 차트

표 6-1 그림 6-16 예의 반환시간과 대기시간

프로세스	반환시간	대기시간
P ₁	10	0
P ₂	(38 - 1)=37	(10 - 1)=9
P ₃	(44 - 2)=42	(38 - 2)=36
P ₄	(48 - 3)=45	(44 - 3)=41
P ₅	(62 - 4)=58	(48 - 4)=44
평균 반환시간: 38.4[=(10 + 37 + 42 + 45 + 58)/5]		평균 대기시간: 26[=(0 + 9 + 36 + 41 + 44)/5]

1. 선입선처리 스케줄링

■ 선입선처리 스케줄링의 장점과 단점

표 6-3 선입선처리 스케줄링의 장점과 단점

장점	<ul style="list-style-type: none">• 스케줄링의 이해와 구현이 단순하다.• 준비 큐에 있는 모든 프로세스가 결국 실행되므로 기아 없는 공정한 정책이다.• 프로세서가 지속적으로 유용한 프로세스를 수행하여 처리율이 높다.
단점	<ul style="list-style-type: none">• 비선점식이므로 대화식 프로세스(작업)에는 부적합하다.• 장기 실행 프로세스가 뒤의 프로세스(작업)를 모두 지연시켜 평균 대기시간이 길어져 최악의 대기시간이 된다.• 긴 프로세스(작업)가 실행되는 동안 짧은 프로세스(작업)가 긴 대기시간으로 호위 효과가 발생할 수 있다.

2. 최소작업 우선 스케줄링 SJF, Shortest Job First

■ 최소작업 우선 스케줄링의 개념

- 각 작업의 프로세서 실행 시간 이용하여 프로세서가 사용 가능할 때 실행 시간이 가장 짧은 작업(프로세스)에 할당하는 방법
- 원리

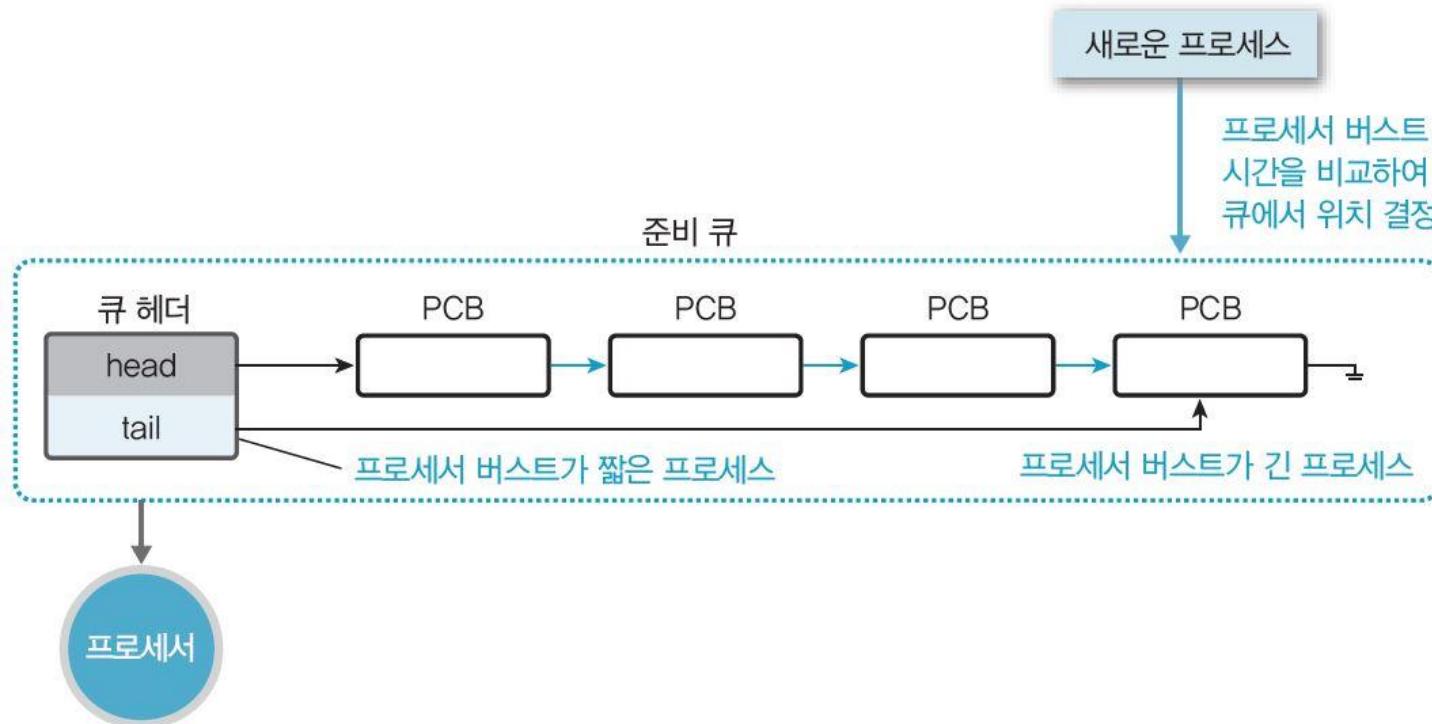
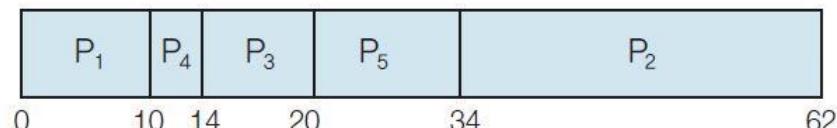


그림 6-19 최소작업 우선 스케줄링의 원리

2. 최소작업 우선 스케줄링 SJF, Shortest Job First

■ 최소작업 우선 스케줄링 예

프로세스	도착 시간	실행 시간
P ₁	0	10
P ₂	1	28
P ₃	2	6
P ₄	3	4
P ₅	4	14



(a) 준비 큐

(b) 간트 차트

그림 6-21 최소작업 우선 스케줄링 예

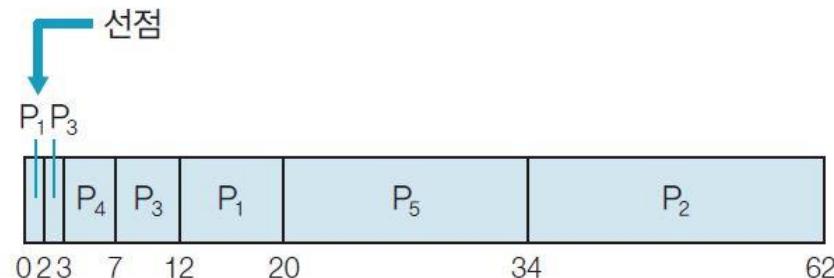
표 6-4 그림 6-21 예의 반환시간과 대기시간

프로세스	반환시간	대기시간
P ₁	10	0
P ₂	(62 - 1)=61	(34 - 1)=33
P ₃	(20 - 2)=18	(14 - 2)=12
P ₄	(14 - 3)=11	(10 - 3)=7
P ₅	(34 - 4)=30	(20 - 4)=16
평균 반환시간: 26[=(10 + 61 + 18 + 11 + 30)/5]	평균 대기시간: 13.6[=(0 + 33 + 12 + 7 + 16)/5]	

2. 최소작업 우선 스케줄링 SJF, Shortest Job First

■ 선점 최소작업 우선 스케줄링 예

프로세스	도착 시간	실행 시간
P ₁	0	10
P ₂	1	28
P ₃	2	6
P ₄	3	4
P ₅	4	14



(a) 준비 큐

(b) 간트 차트

그림 6-22 선점 최소작업 우선 스케줄링 예

표 6-5 그림 6-22 예의 반환시간과 대기시간

프로세스	반환시간	대기시간
P ₁	(20 - 2) = 18	(12 - 2) = 10
P ₂	(62 - 1) = 61	(34 - 1) = 33
P ₃	(12 - 3) = 9	(7 - 3) = 4
P ₄	(7 - 3) = 4	(3 - 3) = 0
P ₅	(34 - 4) = 30	(20 - 4) = 16
평균 반환시간: $24.4 [=(18 + 61 + 9 + 4 + 30)/5]$		평균 대기시간: $12.6 [=(10 + 33 + 4 + 0 + 16)/5]$

2. 최소작업 우선 스케줄링 SJF, Shortest Job First

■ 최소작업 우선 스케줄링 : 짧은 작업 우선 처리 예

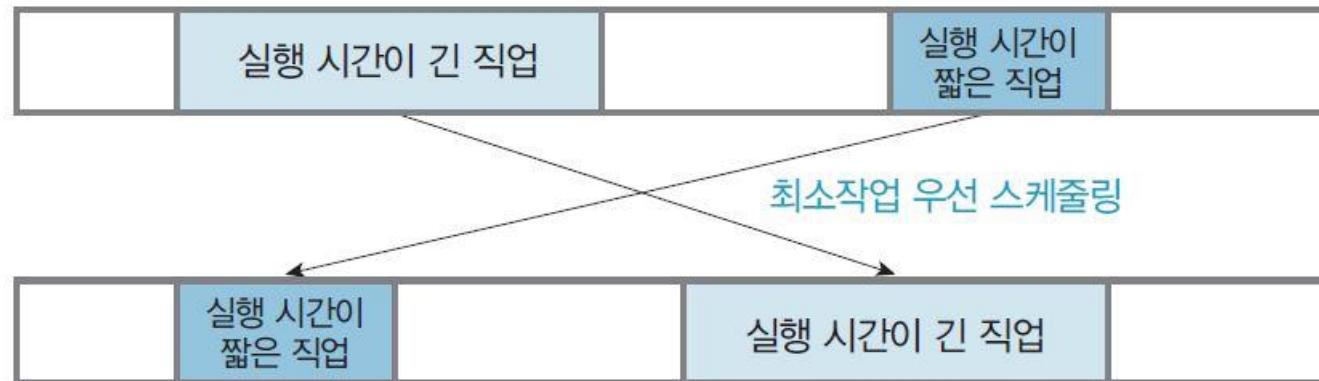


그림 6-23 최소작업 우선 스케줄링 : 짧은 작업 우선 처리

■ 표 6-6 최소작업 우선 스케줄링의 장점과 단점

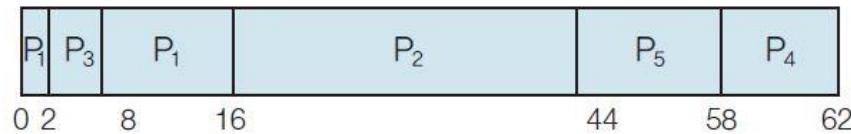
장점	항상 실행 시간이 짧은 작업을 신속하게 실행하므로 평균 대기시간이 가장 짧다.
단점	<ul style="list-style-type: none">초기의 긴 작업을 짧은 작업을 종료할 때까지 대기시켜 기아가 발생한다.기본적으로 짧은 작업이 항상 실행되도록 설정하므로 불공정한 작업을 실행한다.실행 시간을 예측하기가 어려워 실용적이지 못하다.

3. 우선 순위 스케줄링 priority scheduling

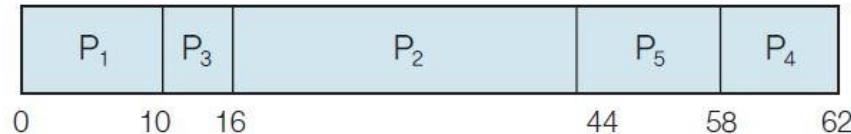
■ 우선 순위 스케줄링 예

프로세스	도착 시간	실행 시간	우선순위
P ₁	0	10	3
P ₂	1	28	2
P ₃	2	6	4
P ₄	3	4	1
P ₅	4	14	2

(a) 준비 큐



(b) 선점 우선순위 간트 차트



(c) 비선점 우선순위 간트 차트

그림 6-26 우선순위 스케줄링 예

4. 라운드 로빈 round-robin 스케줄링

■ 라운드 로빈 스케줄링의 개념

- 특별히 시분할 시스템을 위해 설계
- 작은 단위의 시간인 규정 시간량^{time quantum} 또는 시간 할당량^{time slice} 정의
- 보통 규정 시간량은 10×10 밀리초에서 100×10 밀리초 범위
- 준비 큐를 순환 큐^{circular queue}로 설계하여 스케줄러가 준비 큐를 돌아가면서 한 번에 한 프로세스에 정의된 규정 시간량만큼 프로세서 제공

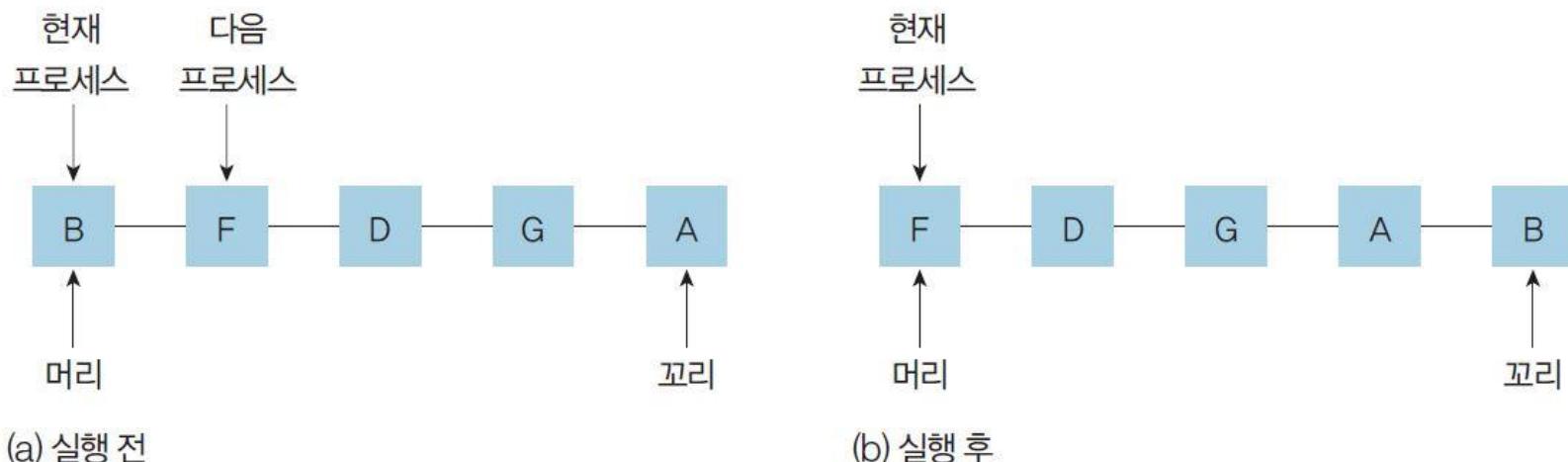


그림 6-28 프로세스 B를 실행한 후의 준비 큐

4. 라운드 로빈 round-robin 스케줄링

- 원리

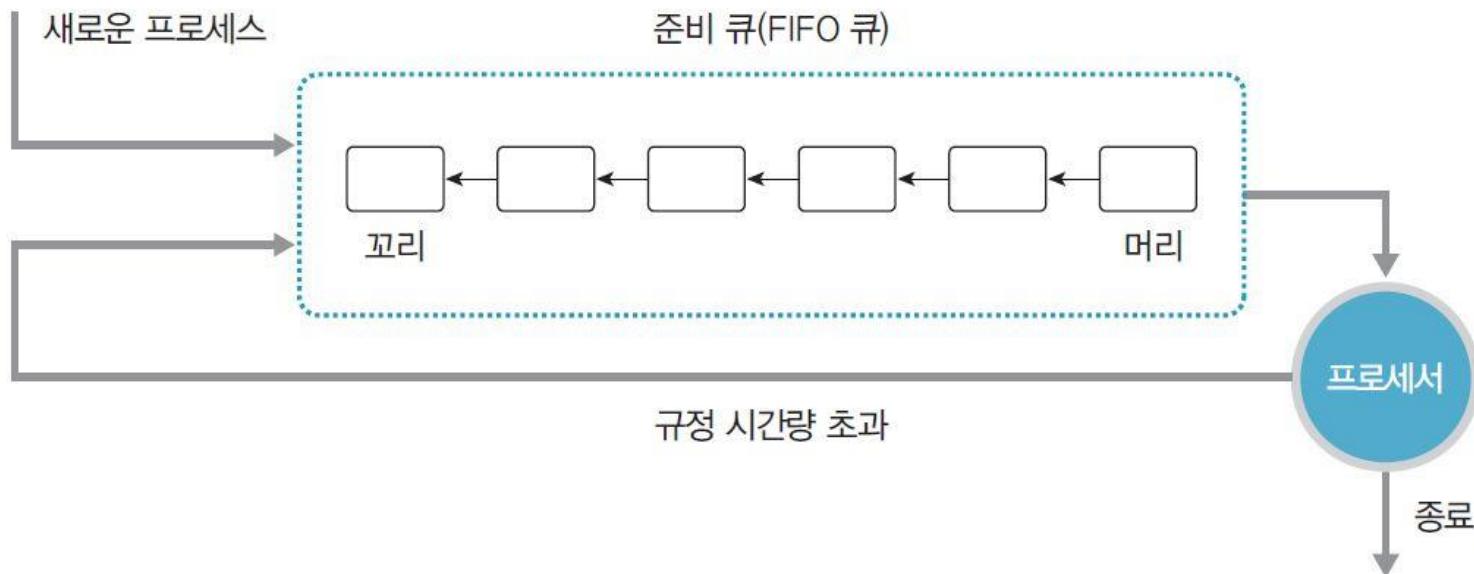


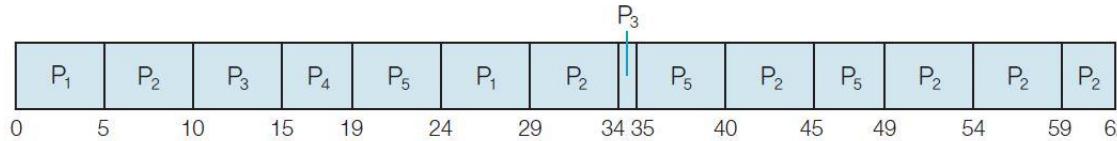
그림 6-29 라운드 로빈 스케줄링

4. 라운드 로빈 round-robin 스케줄링

■ 라운드 로빈 스케줄링 예

프로세스	도착 시간	실행 시간
P ₁	0	10
P ₂	1	28
P ₃	2	6
P ₄	3	4
P ₅	4	14

(a) 준비 큐



(b) 간트 차트

그림 6-30 라운드 로빈 스케줄링 예

표 6-10 그림 6-30 예의 반환시간과 대기시간

프로세스	반환시간	대기시간
P ₁	29	(24 - 5) = 19
P ₂	(62 - 1) = 61	(49 - 15 - 1) = 33
P ₃	(35 - 2) = 33	(34 - 5 - 2) = 27
P ₄	(19 - 3) = 16	(10 - 3) = 7
P ₅	(49 - 4) = 45	(45 - 10 - 4) = 31
평균 반환시간: 36.8[=(29 + 61 + 33 + 16 + 45)/5]		평균 대기시간: 23.4[=(19 + 33 + 27 + 7 + 31)/5]

4. 라운드 로빈(round-robin) 스케줄링

■ 문맥 교환 시간이 라운드 로빈 스케줄링에 미치는 영향

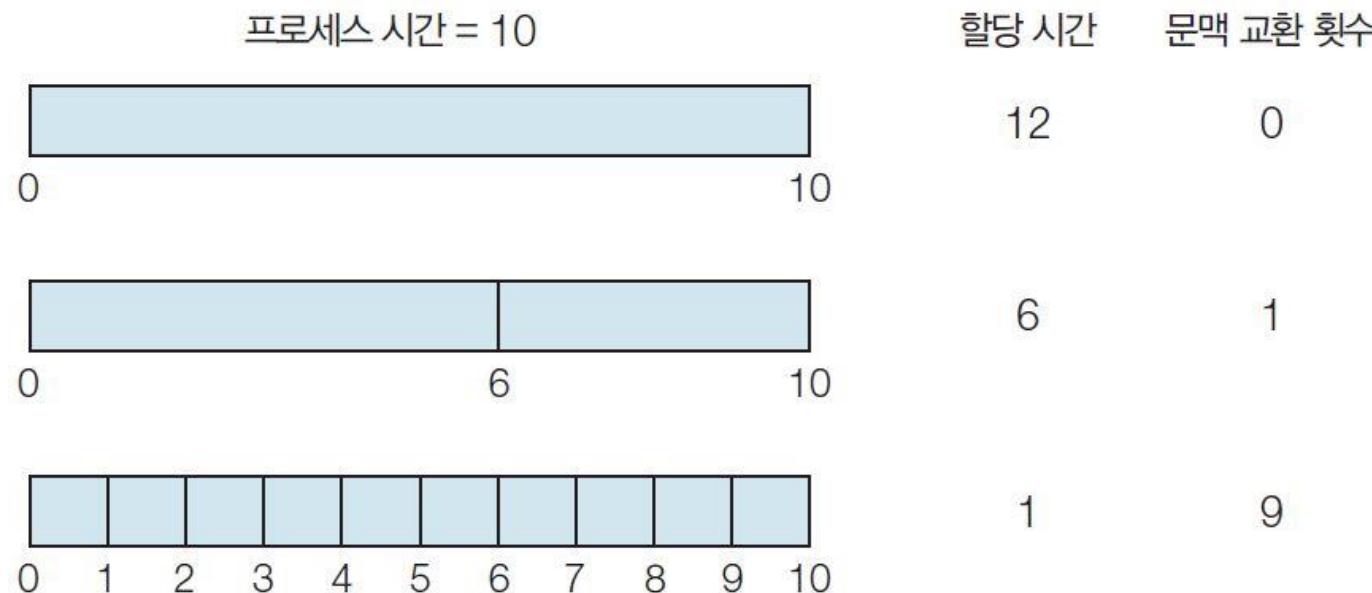


그림 6-31 문맥 교환 횟수를 증가시키는 작은 규정 시간량

규정 시간량이 작을수록 문맥 교환 횟수는 많아지므로 문맥 교환에 소요하는 시간보다 규정 시간량을 충분히 크게 해야 함

4. 라운드 로빈(round-robin) 스케줄링

■ 규정 시간량에 따른 반환 시간의 변화



프로세스	시간
P_1	6
P_2	3
P_3	1
P_4	7

규정 시간량이 작으면 문맥 교환을 많이 하므로 평균 반환시간이 더 좋지 않다

그림 6-32 규정 시간량에 따른 반환시간의 변화



그림 6-33 규정 시간량에 따른 프로세스의 반환시간

5. 다단계 큐 스케줄링 MLQ, MultiLevel Queue

■ 다단계 큐 스케줄링의 개념

- 각 작업을 서로 다른 묶음으로 분류할 수 있을 때 사용
- 준비 상태 큐를 종류별로 여러 단계로 분할, 그리고 작업을 메모리의 크기나 프로세스의 형태에 따라 특정 큐에 지정. 각 큐는 자신만의 독자적인 스케줄링 갖음
- 원리

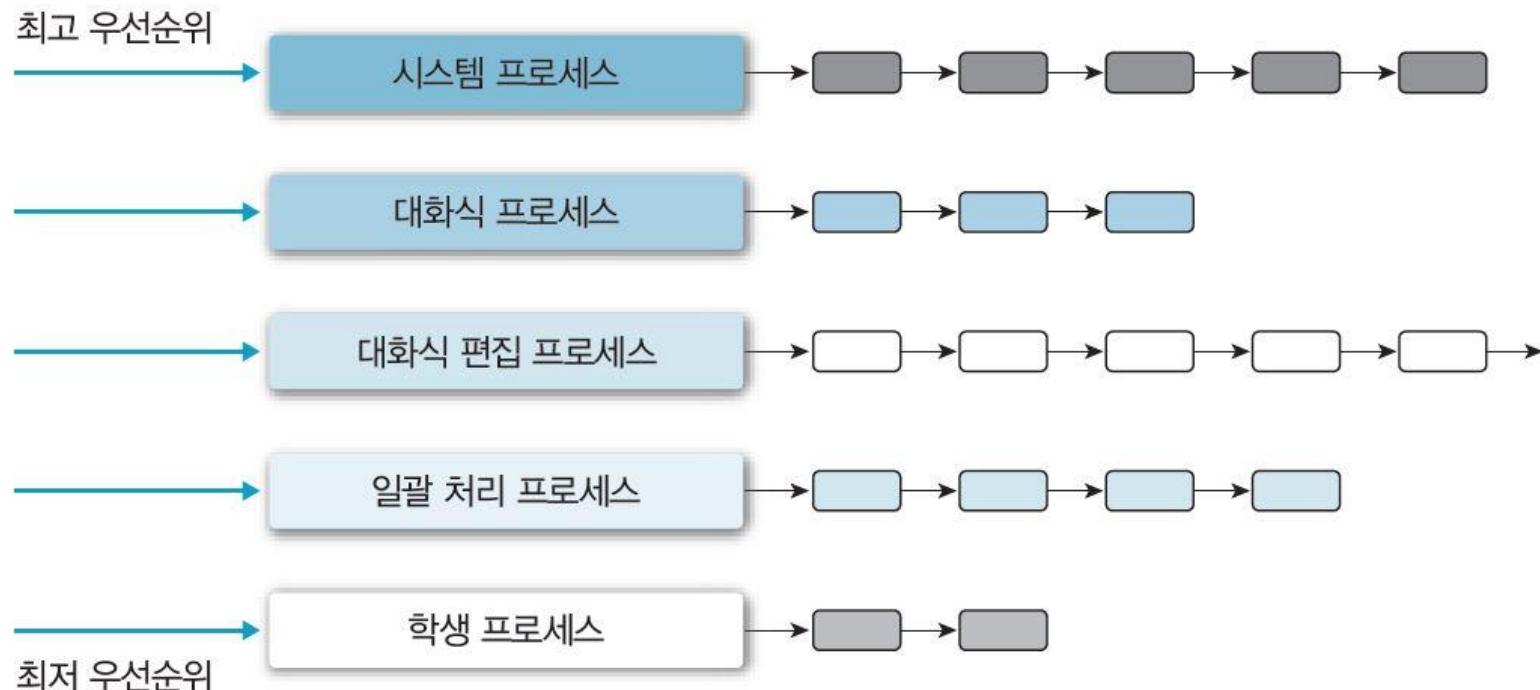


그림 6-34 다단계 큐 스케줄링

각 큐는 순서대로 절대적인 우선순위 갖음
큐 사이에 시간을 나눠 사용할 수도 있음

5. 다단계 큐 스케줄링 MLQ, MultiLevel Queue

■ 다단계 큐 스케줄링의 장점과 단점

표 6-12 다단계 큐 스케줄링의 장점과 단점

장점	응답이 빠르다.
단점	<ul style="list-style-type: none">여러 준비 큐와 스케줄링 알고리즘 때문에 추가 오버 헤드가 발생한다.우선순위가 낮은 큐의 프로세스는 무한정 대기하는 기아가 발생할 수 있다.

6. 다단계 피드백 큐^{MLFQ, MultiLevel Feedback Queue} 스케줄링

■ 다단계 피드백 큐 스케줄링의 개념

- 작업이 시스템에 들어가면 한 큐에서만 고정 실행
- 스케줄링 부담이 적다는 장점이 있으나 융통성이 떨어진다는 단점
- 작업이 큐 사이 이동 가능(프로세서 버스트의 특성에 따라 분리)
- 구조

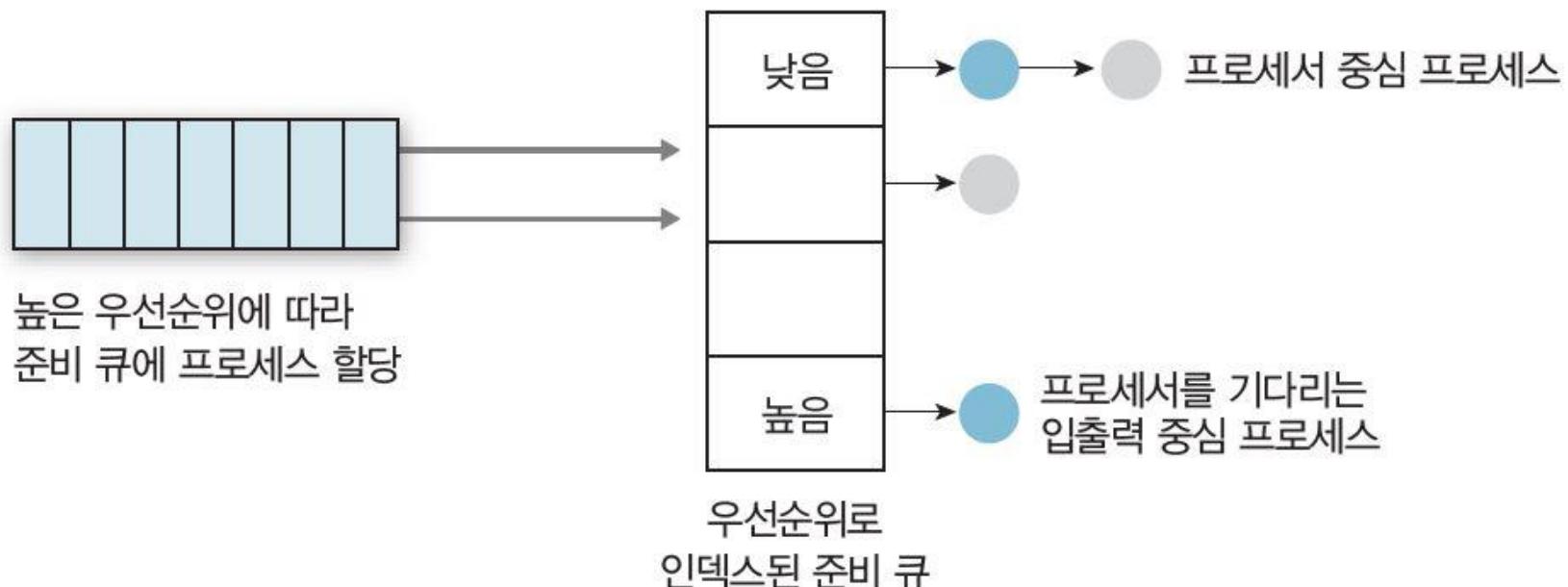


그림 6-35 다단계 피드백 큐 스케줄링 구조

6. 다단계 피드백 큐 MLFQ, MultiLevel Feedback Queue 스케줄링

■ 원리

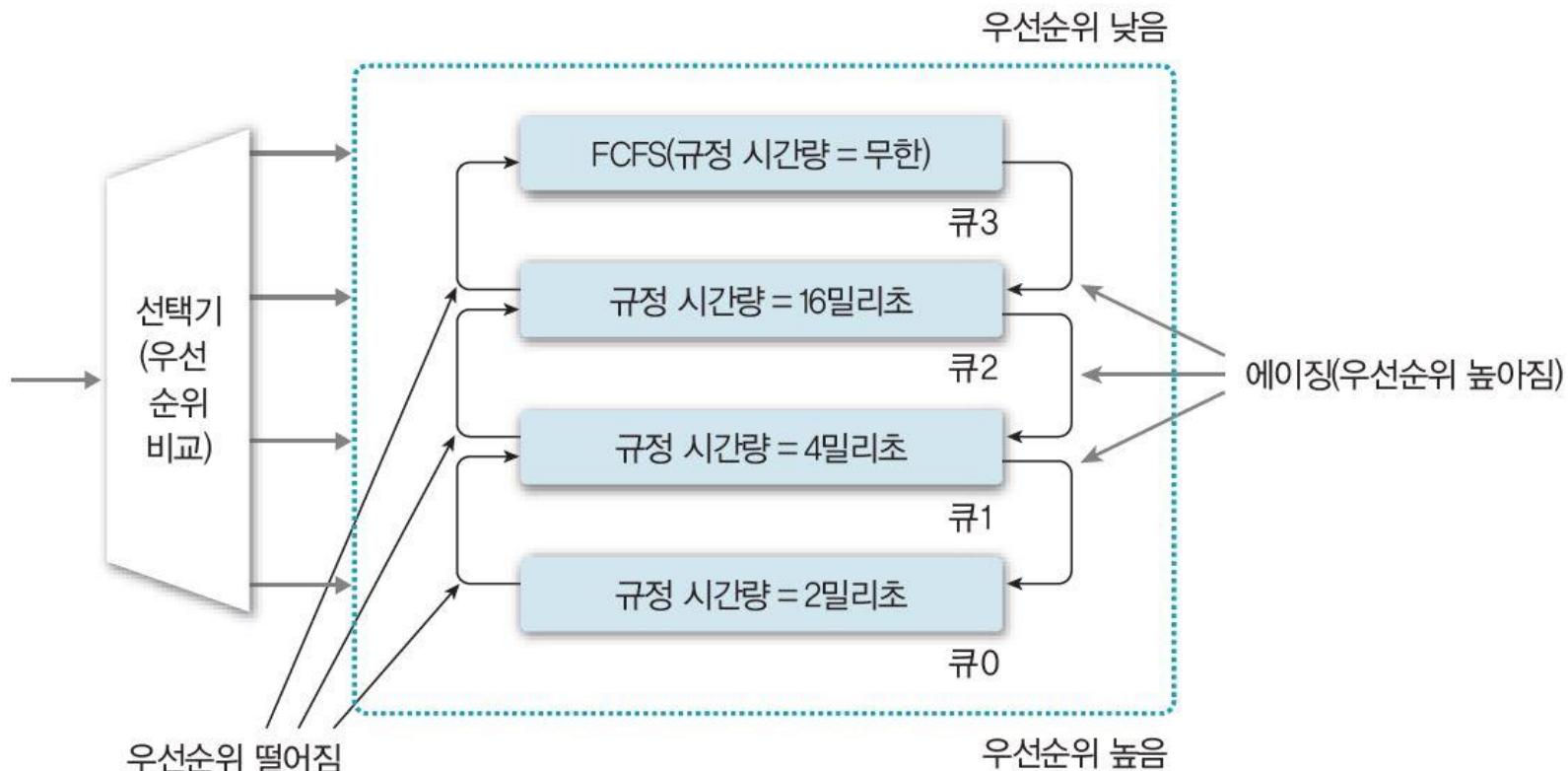


그림 6-36 다단계 피드백 큐 스케줄링

6. 다단계 피드백 큐 MLFQ, MultiLevel Feedback Queue 스케줄링

■ 다단계 피드백 큐 스케줄링의 정의 방법

- 큐 queue 수
- 각 큐에 대한 스케줄링
- 작업을 좀 더 높은 우선순위의 큐로 격상시키는 시기를 결정하는 방법
- 작업을 좀 더 낮은 우선순위의 큐로 격하시키는 시기를 결정하는 방법
- 프로세스들이 어느 큐에 들어갈 것인지 결정하는 방법
- •프로세스가 서비스를 받는 시기를 결정하는 방법

6. 다단계 피드백 큐 MLFQ, MultiLevel Feedback Queue 스케줄링

■ 다단계 피드백 큐 스케줄링의 장점과 단점

표 6-14 다단계 피드백 큐 스케줄링의 장점과 단점

장점	<ul style="list-style-type: none">매우 유연하여 스케줄러를 특정 시스템에 맞게 구성할 수 있다.자동으로 입출력 중심과 프로세서 중심 프로세스를 분류한다.적응성이 좋아 프로세스의 사전 정보가 없어도 최소작업 우선 스케줄링의 효과를 보여 준다.
단점	설계와 구현이 매우 복잡하다.

7. HRN(Highest Response-ratio Next) 스케줄링

■ HNR 스케줄링의 개념

- 최소작업 우선 스케줄링의 약점인 긴 작업과 짧은 작업 간의 지나친 불평 등을 보완
- 비 점 스케줄링이며 우선순위 스케줄링의 또 다른 예
- 선입선처리 스케줄링과 최소작업 우선 스케줄링의 약점을 해결위해 제안
- 우선순위

$$\text{우선순위} = \frac{\text{서비스를 받을 시간} + \text{대기한 시간}}{\text{서비스를 받을 시간}}$$

그림 6-38 HRN 스케줄링의 우선순위

■ 시스템 응답시간

시스템 응답시간 : 대기한 시간 + 서비스를 받을 시간

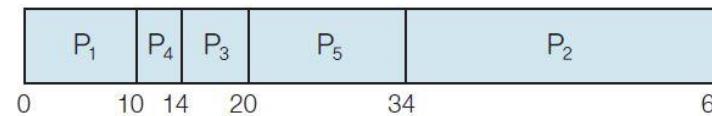
그림 6-39 HRN 스케줄링의 시스템 응답시간

7. HRN(Highest Response-ratio Next 스케줄링)

HNR 스케줄링의 예

프로세스	도착 시간	실행 시간
P ₁	0	10
P ₂	1	28
P ₃	2	6
P ₄	3	4
P ₅	4	14

(a) 준비 큐



(b) 간트 차트

그림 6-40 HRN 스케줄링 예

표 6-15 그림 6-40 예의 반환시간과 대기시간

프로세스	반환시간	대기시간
P ₁	10	0
P ₂	(62 - 1)=61	(34 - 1)=33
P ₃	(20 - 2)=18	(14 - 2)=12
P ₄	(14 - 3)=11	(10 - 3)=7
P ₅	(34 - 4)=30	(20 - 4)=16
평균 반환시간: 26[=(10 + 61 + 18 + 11 + 30)/5]	평균 대기시간: 14[=(0 + 33 + 12 + 7 + 16)/5]	

7. HRN(Highest Response-ratio Next 스케줄링)

■ HNR 스케줄링의 장점과 단점

표 6-16 HRN 스케줄링의 장점과 단점

장점	<ul style="list-style-type: none">• 자원을 효율적으로 활용한다.• 기아가 발생하지 않는다.
단점	오버헤드가 높을 수 있다(메모리와 프로세서 낭비).

Section 02 상호배제와 동기화(1.상호배제의 개념)

■ 상호배제 mutual exclusion의 개념 (Mutex, 뮤텍스)

- 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

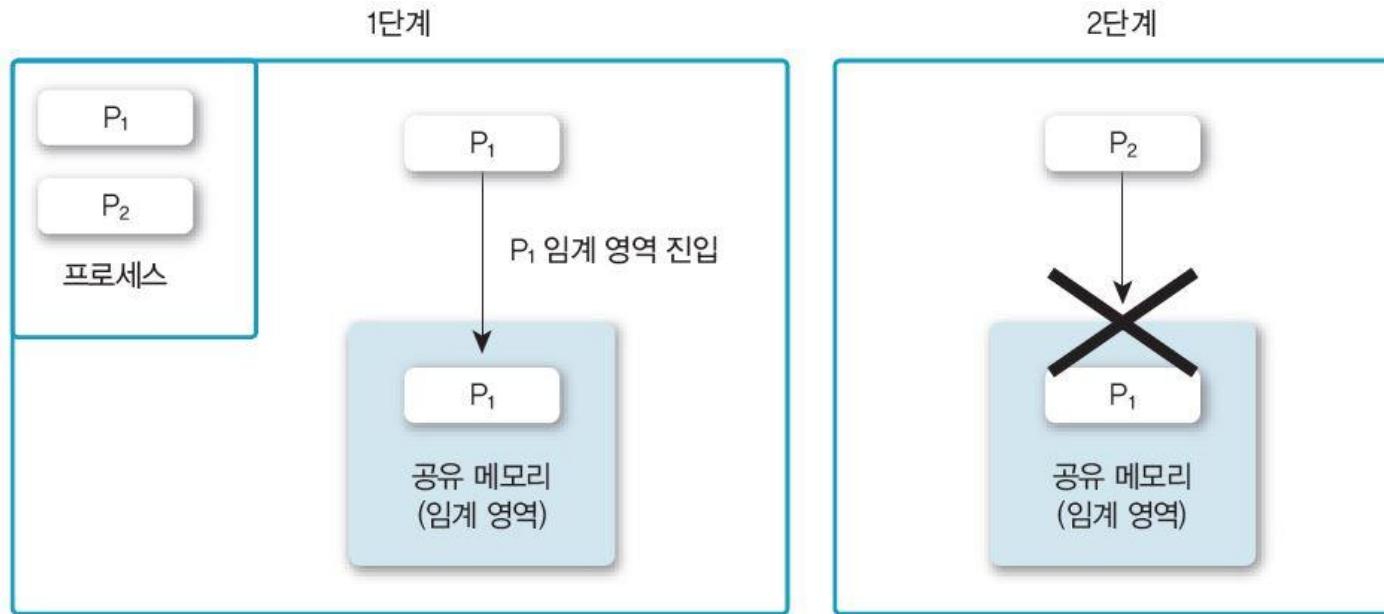


그림 4-13 상호배제의 개념

임계자원 *critical resource* : 두 프로세스가 동시에 사용할 수 없는 공유 자원

임계영역 *critical section* : 임계 자원에 접근하고 실행하는 프로그램 코드 부분

■ 상호배제의 조건

- ① 두 프로세스는 동시에 공유 자원에 진입 불가.
- ② 프로세스의 속도나 프로세서 수에 영향 받지 않음
- ③ 공유 자원을 사용하는 프로세스만 다른 프로세스 차단 가능
- ④ 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

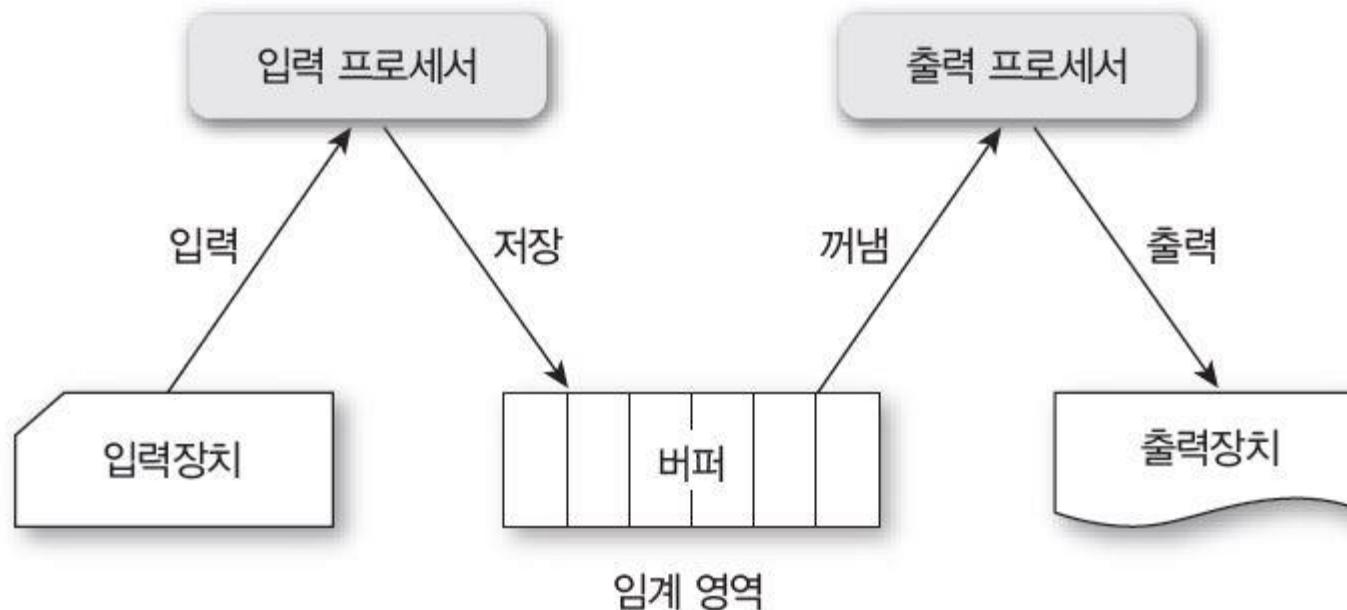


그림 4-14 임계 영역 예

2. 임계 영역

■ 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

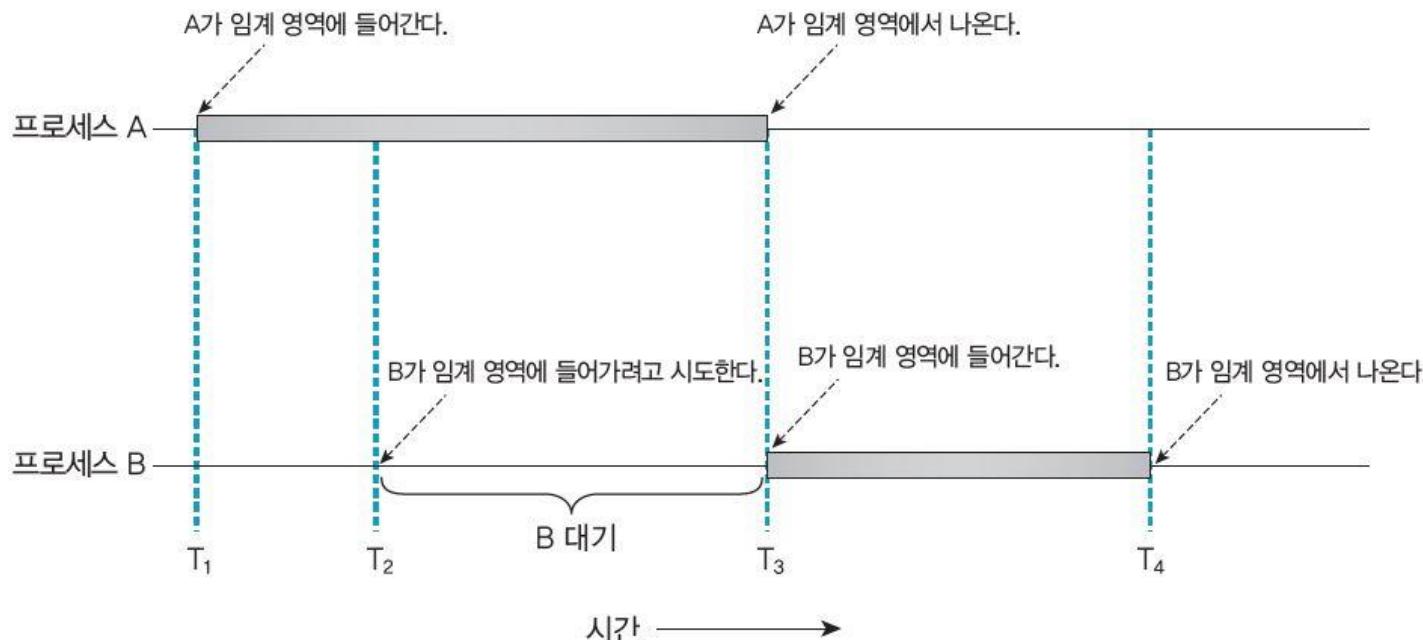


그림 4-15 임계 영역을 이용한 상호배제

2. 임계영역

■ 병행 프로세스에서 영역 구분

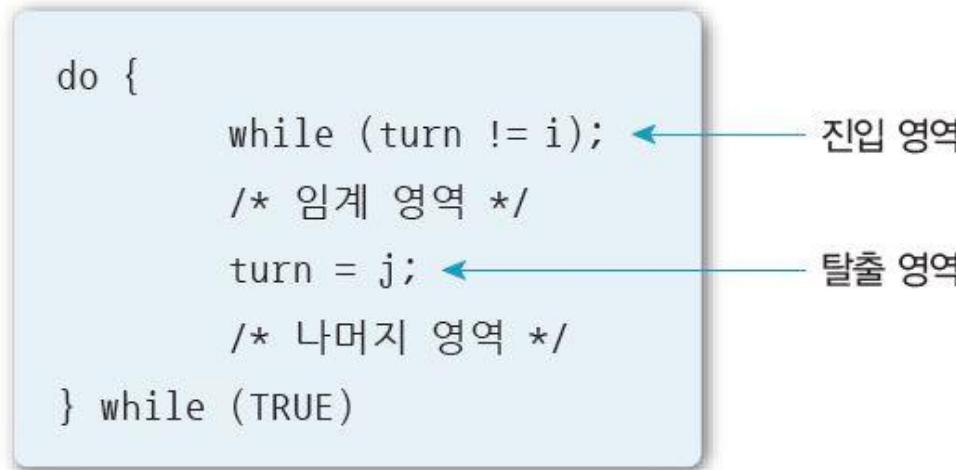


그림 4-16 병행 프로세스에서 영역 구분

■ 임계 영역의 조건

- ① 상호배제 : 어떤 프로세스가 임계 영역에서 작업 중, 다른 프로세스 임계 영역 진입 불가
- ② 진행 : 임계 영역에 프로세스가 없는 상태에서 어떤 프로세스가 들어갈지 결정
- ③ 한정 대기 : 다른 프로세스가 임계 영역을 무한정 기다리는 상황 방지 위해 임계 영역에 한 번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한

Critical Section Problem

■ Mutual Exclusion

Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

- 둘 이상의 프로세스가 동시에 cs에 존재해서는 안됨
- 상대 플래그 체크 이후에 자신의 플래그를 들게 되면 cs 동시 진입 가능성 있음

■ Progress (Deadlock-freedom)

Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

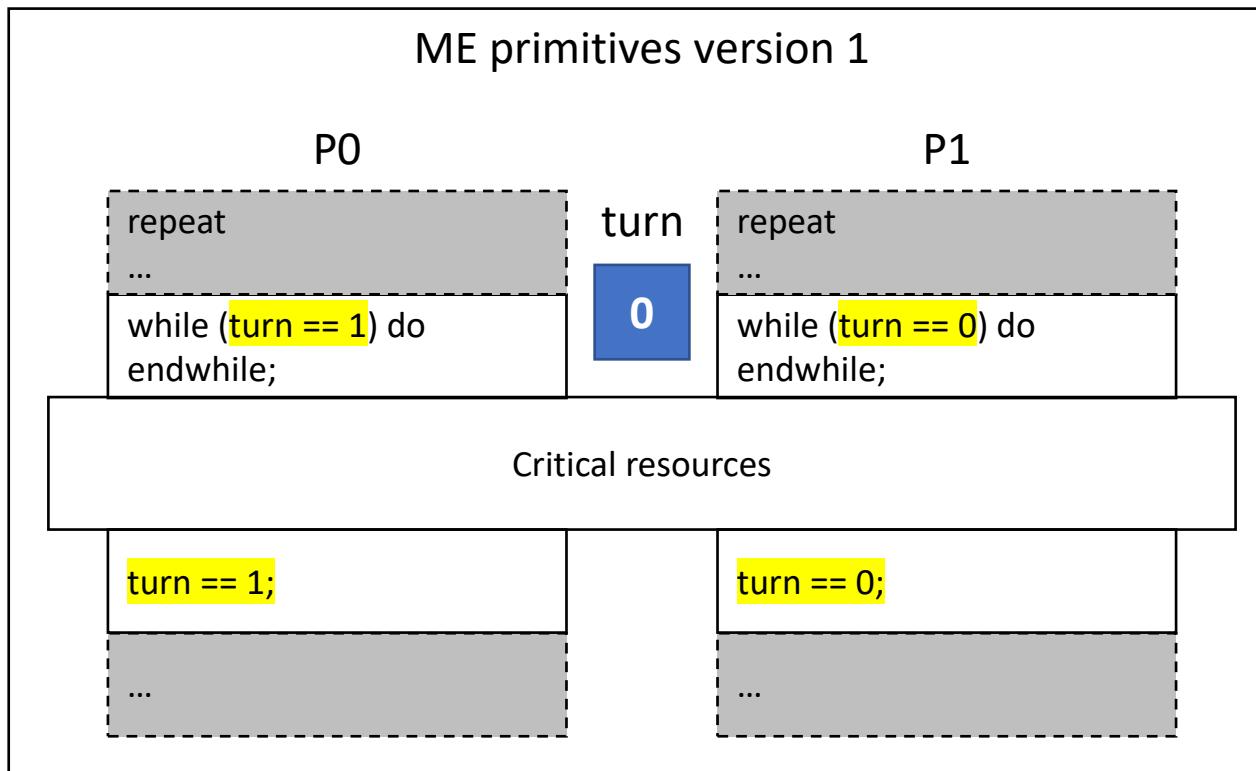
- 시스템 관점
- If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

■ Bounded Waiting (Starvation-freedom)

Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

- 프로세스 관점, Progress보다 더 강력한 조건
- If a process is trying to enter its critical section, then this process must eventually enter its critical section.
- “적극적인 양보”

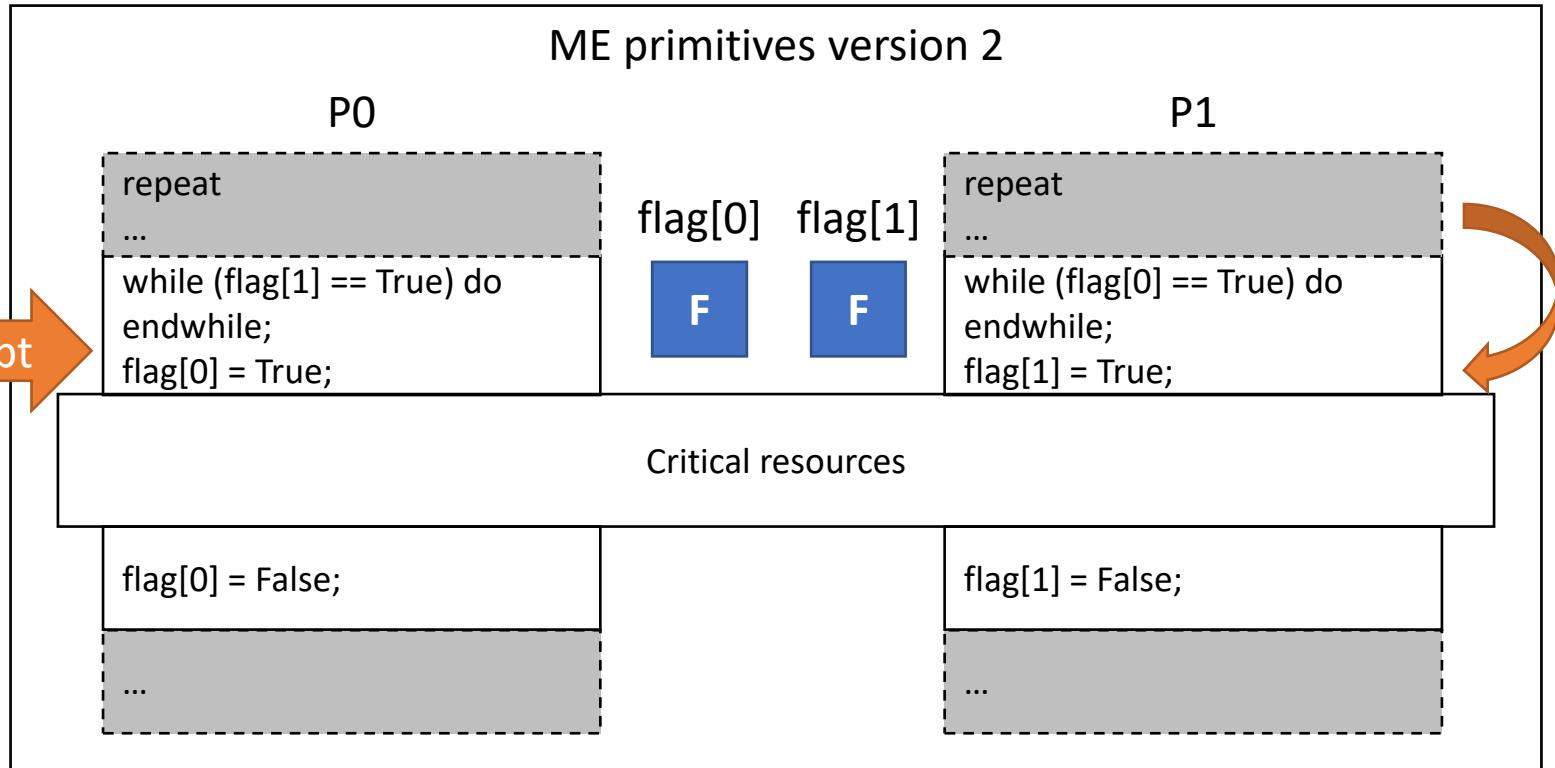
Three possible ME primitives



■ 임계 영역의 조건

- ① 상호배제 : 어떤 프로세스가 임계 영역에서 작업 중, 다른 프로세스 임계 영역 진입 불가
- ② 진행 : 임계 영역에 프로세스가 없는 상태에서 어떤 프로세스가 들어갈지 결정
- ③ 한정 대기 : 다른 프로세스가 임계 영역을 무한정 기다리는 상황 방지 위해 임계 영역에 한 번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한

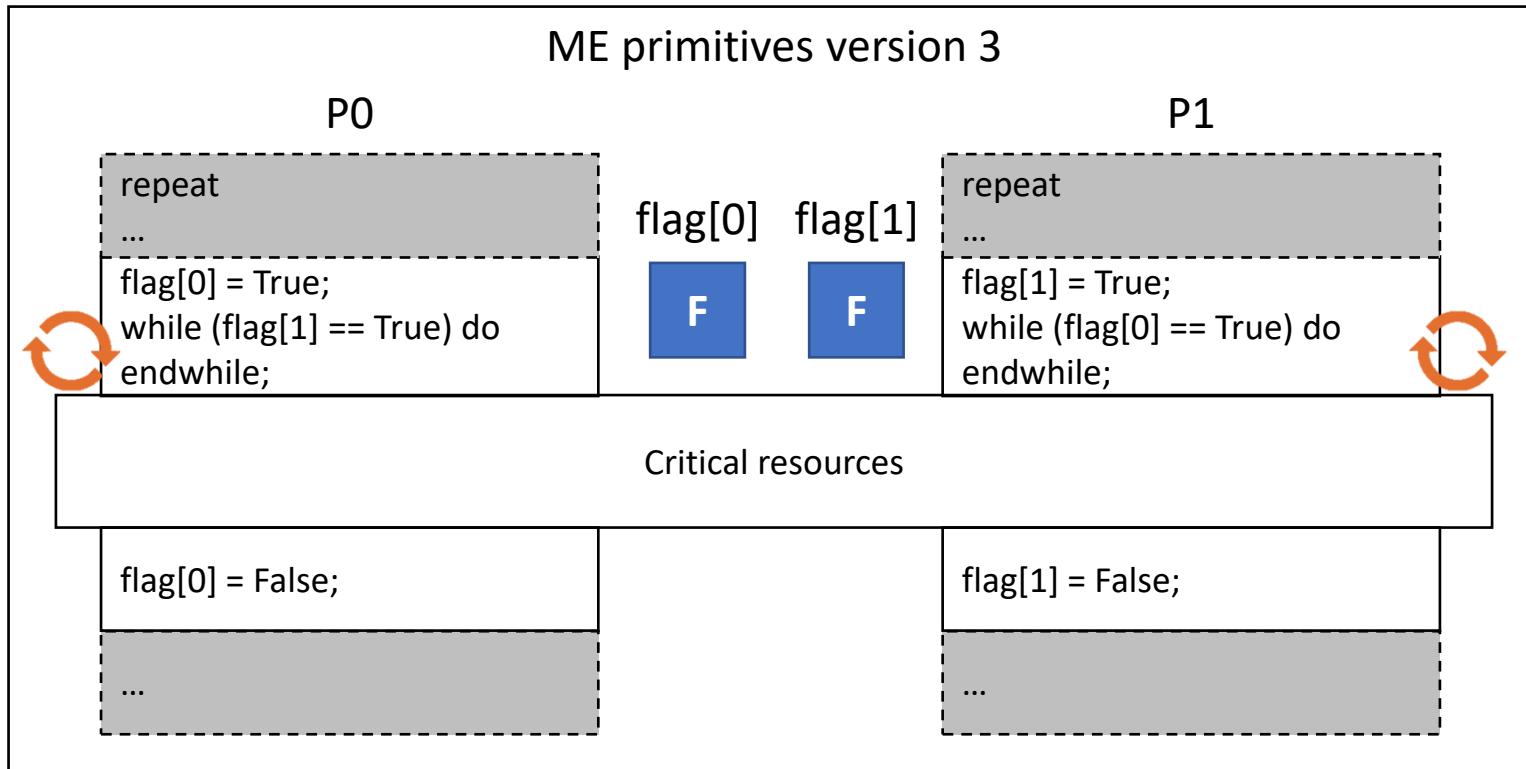
Three possible ME primitives



■ 임계 영역의 조건

- ① 상호배제 : 어떤 프로세스가 임계 영역에서 작업 중, 다른 프로세스 임계 영역 진입 불가
- ② 진행 : 임계 영역에 프로세스가 없는 상태에서 어떤 프로세스가 들어갈지 결정
- ③ 한정 대기 : 다른 프로세스가 임계 영역을 무한정 기다리는 상황 방지 위해 임계 영역에 한 번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한

Three possible ME primitives



■ 임계 영역의 조건

- ① 상호배제 : 어떤 프로세스가 임계 영역에서 작업 중, 다른 프로세스 임계 영역 진입 불가
- ② 진행 : 임계 영역에 프로세스가 없는 상태에서 어떤 프로세스가 들어갈지 결정
- ③ 한정 대기 : 다른 프로세스가 임계 영역을 무한정 기다리는 상황 방지 위해 임계 영역에 한 번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한

Section 03 상호배제 방법들

■ 상호배제 방법

표 4-1 상호배제 방법들

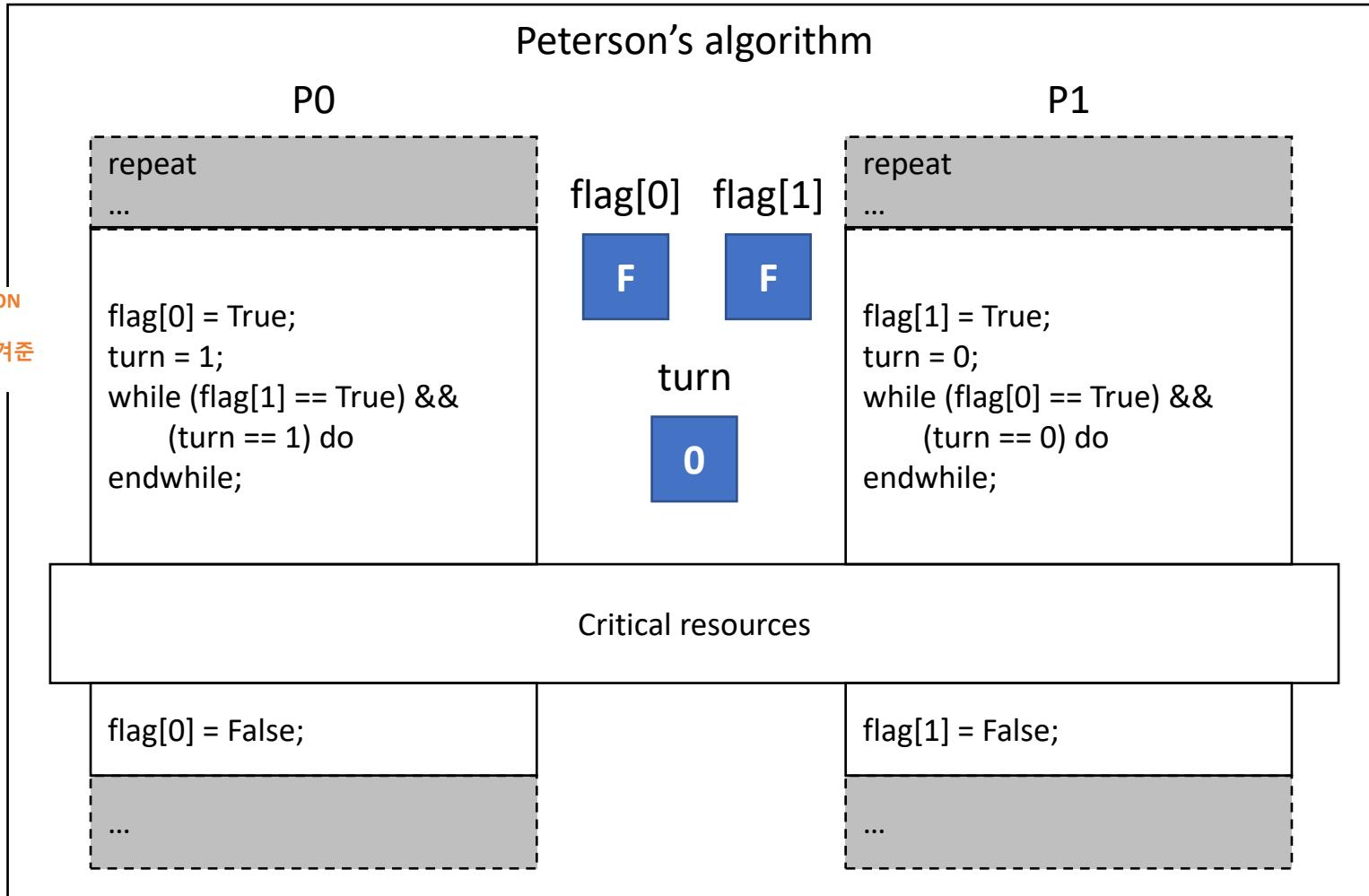
수준	방법	종류
고급	소프트웨어로 해결 소프트웨어가 제공 : 프로그래밍 언어와 운영체제 수준에서 제공	<ul style="list-style-type: none">데커의 알고리즘램포트의 베이커리(빵집) 알고리즘 <ul style="list-style-type: none">크누스의 알고리즘핸슨의 알고리즘다익스트라의 알고리즘
		<ul style="list-style-type: none">세마포모니터
저급	하드웨어로 해결 : 저급 수준의 원자 연산	TestAndSet ^{TAS} (테스)

1-1. 데커의 알고리즘

■ 데커의 알고리즘 개념

- 두 프로세스가 서로 통신하려고 공유 메모리를 사용하여 충돌 없이 단일 자원을 공유할 수 있도록 허용하는 것
- 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

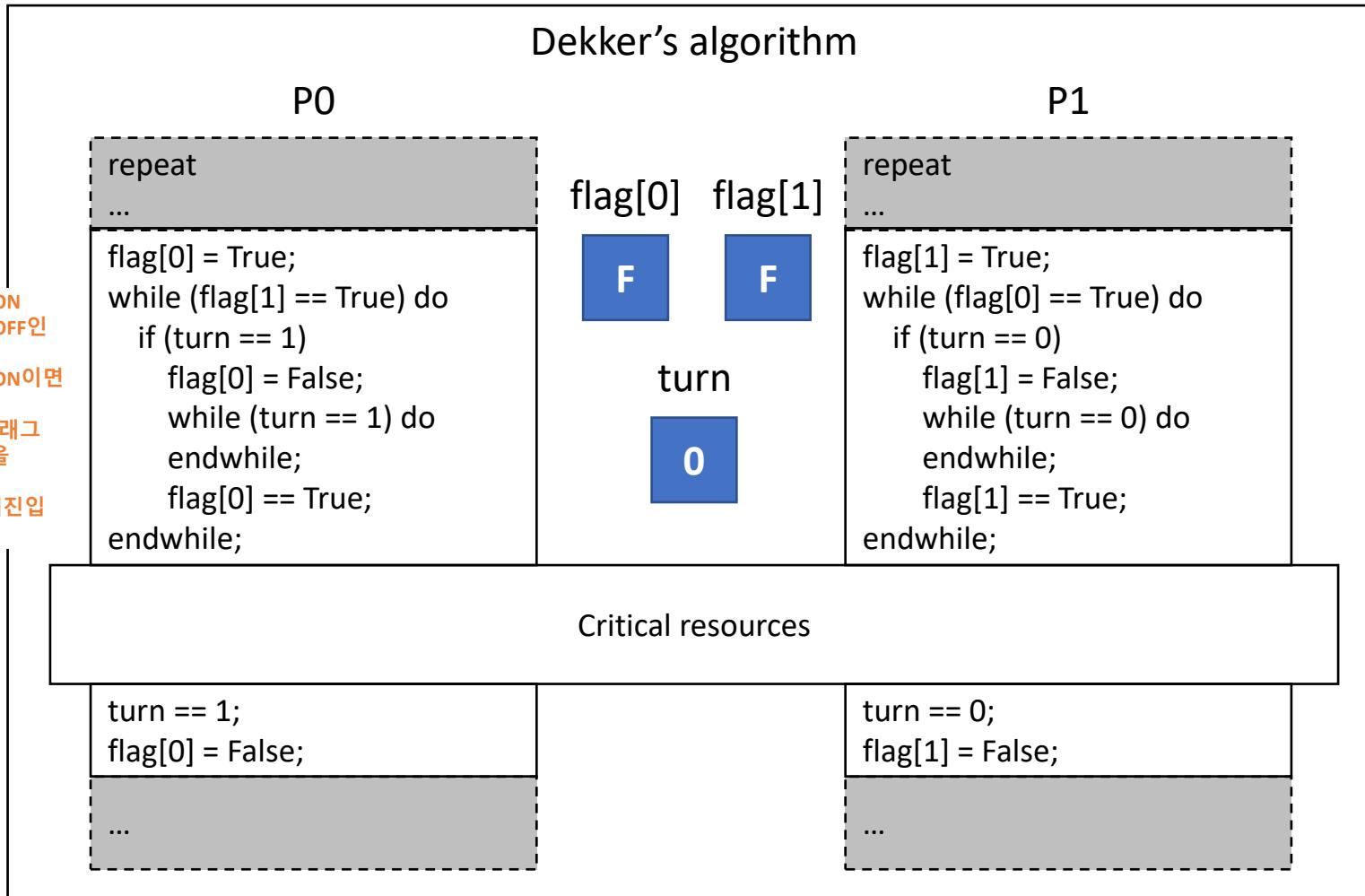
1-2. 피터슨의 알고리즘



■ 임계 영역의 조건

- ① 상호배제 : turn은 binary
- ② 진행 : 내 턴이 아니더라도 cs 진입 가능
- ③ 한정 대기 : 턴을 넘겨주기 때문에 무한히 대기하지 않음 (독점 불가)

1-1. 데커의 알고리즘



■ 임계 영역의 조건

- ① 상호배제 : turn은 binary
- ② 진행 : 상대 플래그가 OFF이면 cs 진입 가능
- ③ 한정 대기 : 상대 턴이라면 자신의 플래그를 OFF하기 때문

소프트웨어 해결 방법

- 속도가 느림
- 소프트웨어적으로 구현 가능, 그러나 구현이 복잡함
- 실행 중 선점될 수 있음
 - Interrupt 억제, 하지만 overhead 발생
- Busy waiting
- (데커, 피터슨 알고리즘 한정) 두 프로세스에 대해서만 사용 가능
 - 실제로 사용되고 있지 않으나 좋은 알고리즘적 설명을 제공

기타 유용한 상호배제 알고리즘

- 크누스 Knuth : 이전 알고리즘 관계 분석 후 일치하는 패턴을 찾아 패턴의 반복을 줄여서 프로세스에 프로세서 할당 무한정 연기할 가능성을 배제하는 해결책을 제시했으나, 프로세스들이 아주 오래 기다려야 함
- 램포트 Lamport : 사람들로 봄비는 빵집에서 번호표 뽑아 빵 사려고 기다리는 사람들에 비유해서 만든 알고리즘 준비 상태 큐에서 기다리는 프로세스마다 우선순위를 부여하여 그중 우선순위가 가장 높은 프로세스에 먼저 프로세서를 할당, '램포트의 베이 커리(빵집) 알고리즘'이라고 함
- 핸슨 Brinch Hansen : 실행 시간이 긴 프로세스에 불리한 부분을 보완하는 것, 대기시간과 실행 시간을 이용하는 모니터 방법, 분산 처리 프로세서 간의 병행성 제어 많이 발표

2. TestAndSet^{TAS}(테스) 명령어

■ TestAndSet 명령어의 개념

- 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(원자적연산 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

원자적 연산 atomic 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 원자적 연산 수행

예제 4-6 TestAndSet 명령어

```
// target을 검사하고, target 값을 true로 설정
boolean TestAndSet (boolean *target) {
    boolean temp = *target;      // 이전 값 기록
    *target = true;              // true로 설정
    return temp;                 // 값 반환
}
```

2. TestAndSet^{TAS}(테스) 명령어

■ 부울 변수 lock을 사용한 상호배제

예제 4-7

lock을 사용한 상호배제

```
do
{
    • lock: 전역 불리언 변수

    while (TestAndSet(&lock))      // lock을 검사하여 true이면 대기, false이면 임계 영역 진입
        ;
        // 아무것도 하지 않음

    // 임계 영역
    lock = false;                  // 다른 프로세스의 진입 허용 의미로 lock을 false로
    // 나머지 영역
} while (true);
```

TAS algorithm

P_i

Repeat

...

while (TestAndSet(&lock))
endwhile

Critical resources

lock = False;

...

■ 임계 영역의 조건

- ① 상호배제 : 동시에 CS에 접근 불가
- ② 진행 : 어떤 다른 프로세스가 lock을 false로 세팅한다면 진행 가능
- ③ 한정 대기 : 프로세스 i의 대기 시간이 무한할 수 있음

2. TestAndSet^{TAS}(테스) 명령어

■ TestAndSet 명령어를 사용한 상호배제

예제 4-8

TestAndSet 명령어를 이용한 상호배제

```
① do                                // 프로세스 Pi의 진입 영역
{
    ② waiting[i] = true;
    key = true;
    ③ while (waiting[i] && key)
        ④ key = TestAndSet(&lock);
    ⑤ waiting[i] = false;
        // 임계 영역
        // 탈출 영역
    ⑥ j = (i + 1) % n;
    ⑦ { while ((j != i) && !waiting[j])      // 대기 중인 프로세스를 찾음
          j = (j + 1) % n;
    ⑧ { if (j == i)                          // 대기 중인 프로세스가 없으면
          lock = false;
    ⑨ { else
          waiting[j] = false;                // 대기 프로세스가 있으면 다음 순서로 임계 영역에 진입
          // Pj가 임계 영역에 진입할 수 있도록
        // 나머지 영역
    } while (true);
```

• key == false
일 때, 임계영역
접근 가능

waiting[i]

1	2	3	...	n
		false		

→

3. 세마포 semaphore

■ 세마포 개념과 동작

- 다익스트라가 테스 명령어의 문제 해결을 위해 제안
- 상호배제 및 다양한 연산의 순서도 제공
- 세마포 값은 true나 false로, P와 V연산과 관련. 네덜란드어로 P는 검사 Proberen, V 증가 Verhogen 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

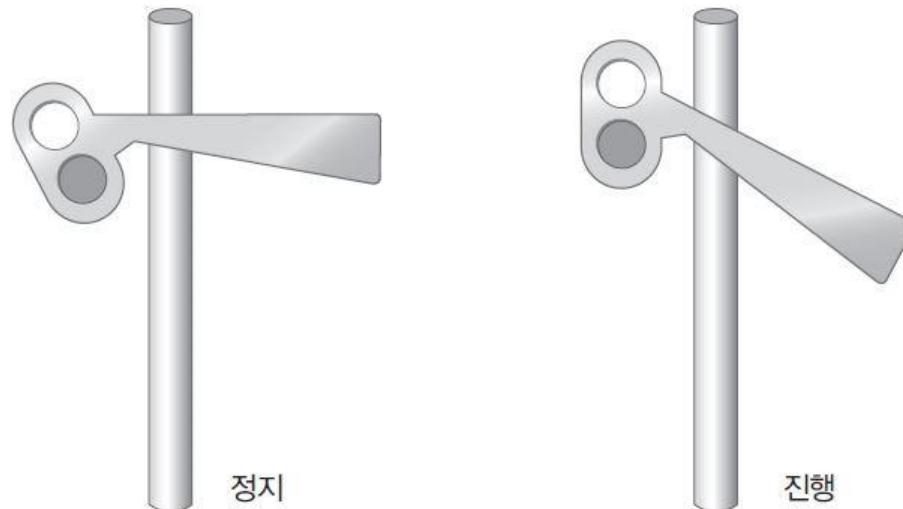


그림 4-23 세마포 예 : 열차 차단기

3. 세마포 semaphore

■ 세마포의 종류

▪ 이진 세마포

- 세마포 S를 상호배제에 사용, 1 또는 0으로 초기화, P와 V의 연산 교대 실행

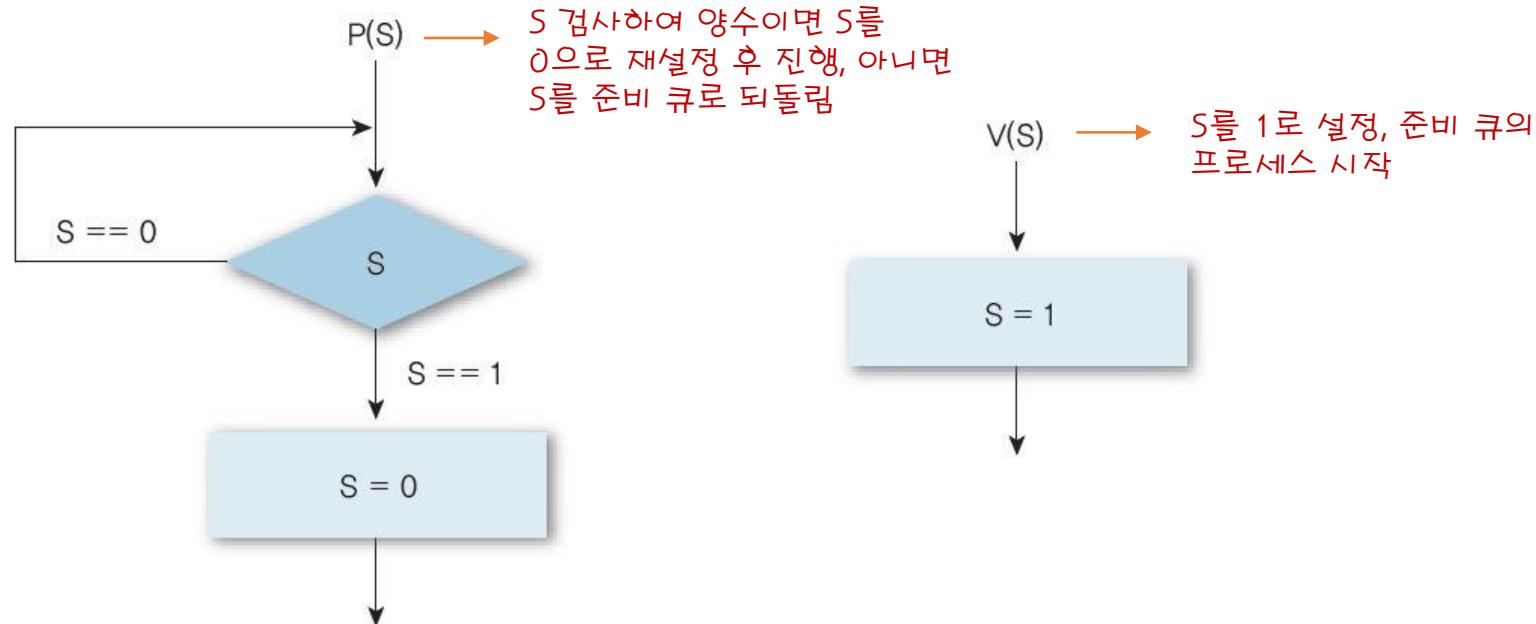


그림 4-26 이진 세마포의 알고리즘

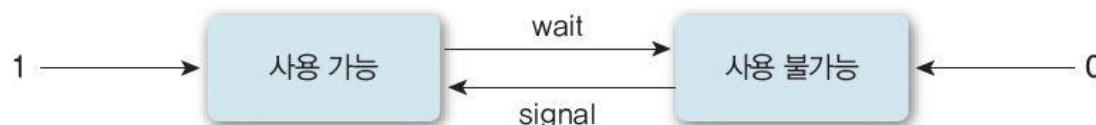


그림 4-27 이진 세마포의 상태도

3. 세마포 semaphore

■ 계수 세마포

- 유한한 자원에 접근 제어 가능, 여러 번 획득, 해제할 수 있도록 count 자원의 사용 허가 값으로 사용
- 사용 가능한 자원 수로 초기화하므로, count를 초기의 세마포 수로 초기화

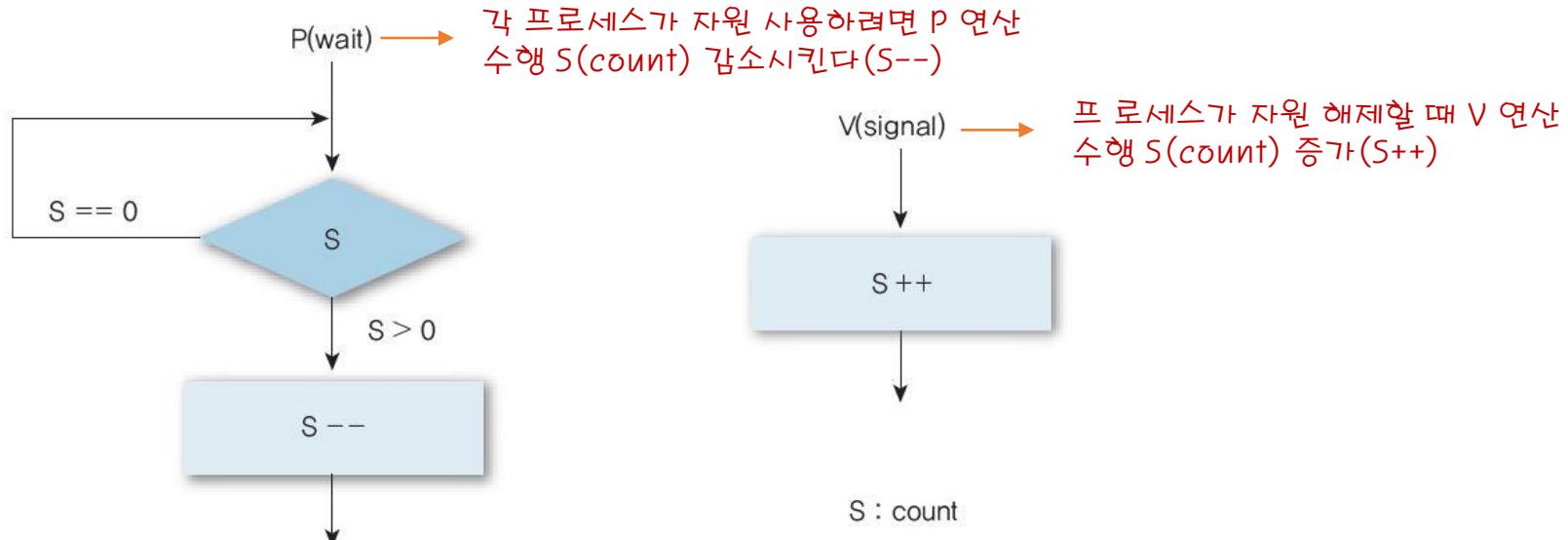


그림 4-28 계수 세마포의 알고리즘

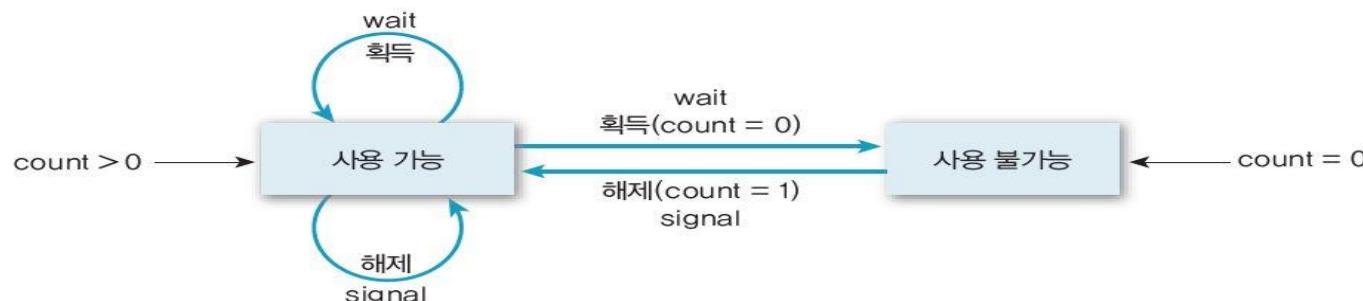


그림 4-29 계수 세마포의 상태도

3. 세마포 semaphore

■ 세마포의 구현

- 세마포의 구조

예제 4-12 > 세마포의 구조

```
struct semaphore {  
    int count;  
    queueType queue;  
};  
semaphore S;
```

4. 모니터monitor

■ 모니터의 개념과 구조

- 세마포의 오용으로 여러 가지 오류가 쉽게 발생하면 프로그램 작성 곤란.

이런 단점 극복 위해 등장

- 핸슨 Hansen 제안, 호 Hoare 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

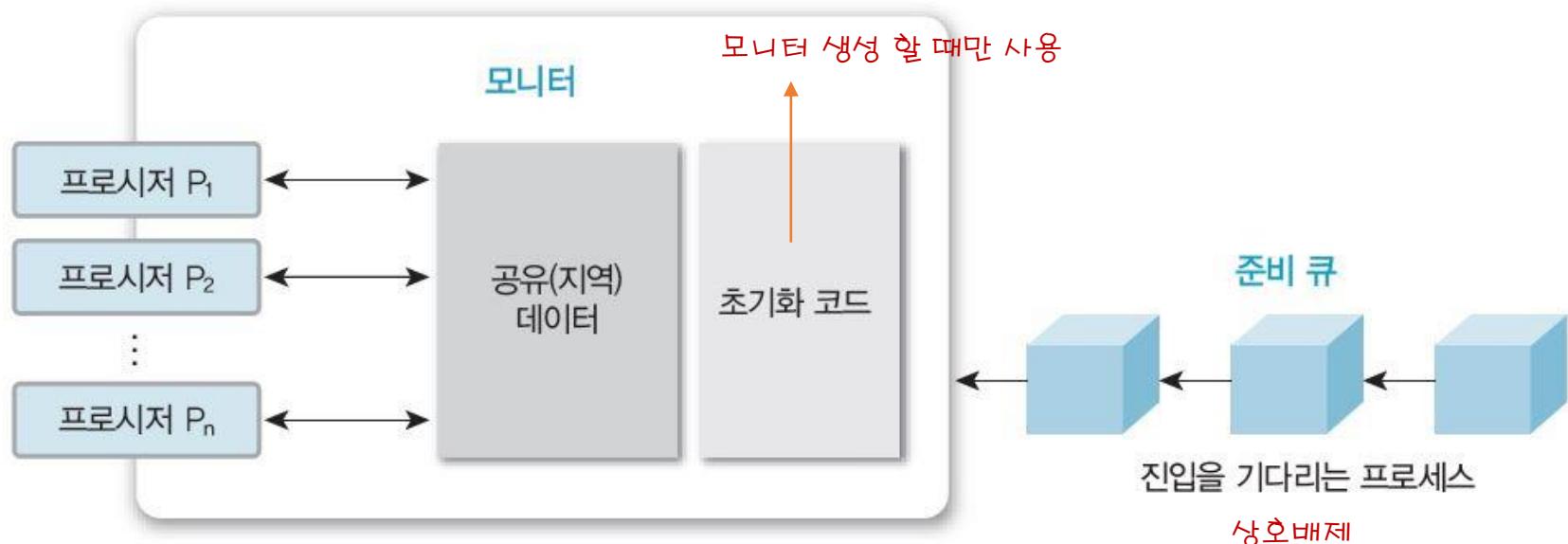


그림 4-30 모니터의 구조

1. 교착 상태의 개념

■ 프로세스의 자원 사용 순서

① 자원 요청 : 프로세스가 필요한 자원 요청

 해당 자원 다른 프로세스가 사용 중이면 요청을 수락 때까지 대기.

② 자원 사용 : 프로세스가 요청한 자원 획득하여 사용

③ 자원 해제 : 프로세스가 자원 사용 마친 후 해당 자원 되돌려(해제) 줌

3. 교착 상태의 발생 조건

■ 교착 상태 발생의 네 가지 조건

① 상호배제

- 자원을 최소 하나 이상 비공유. 즉, 한 번에 프로세스 하나만 해당 자원 사용할 수 있어야 함
- 사용 중인 자원을 다른 프로세스가 사용하려면 요청한 자원 해제될 때 까지 대기

② 점유와 대기

- 자원을 최소한 하나 정도 보유, 다른 프로세스에 할당된 자원 얻으려고 대기하는 프로세스 있어야 함

③ 비선점

- 자원 선점 불가. 즉, 자원은 강제로 빼앗을 수 없고, 자원 점유하고 있는 프로세스 끝나야 해제

④ 순환(환형) 대기

- 예

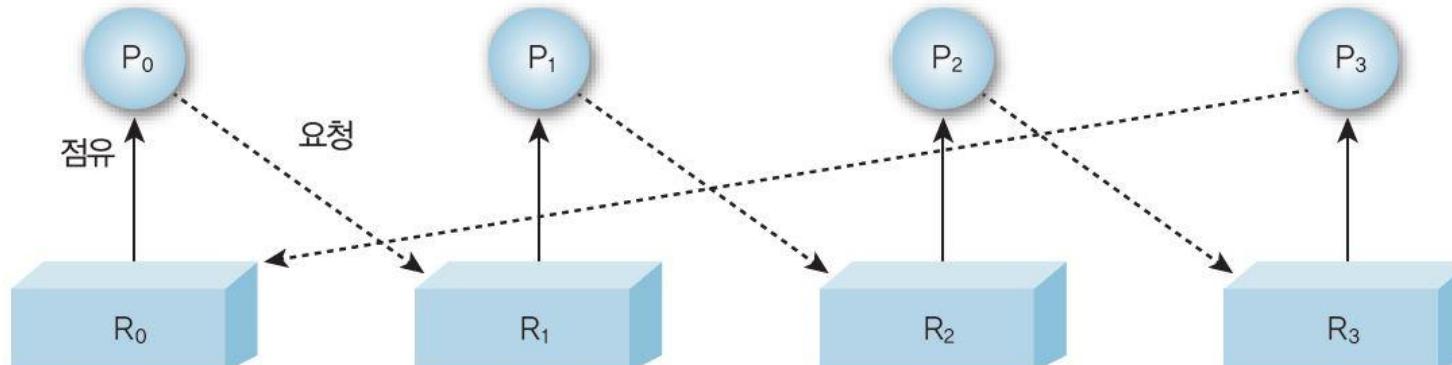


그림 5-5 순환 대기의 교착 상태

①~③만 만족해도 교착 상태가 발생 가능, 발생하지 않을 수도 있음
④는 ①~③ 조건을 만족할 때 발생 할 수 있는 결과, 점유와 대기 조건을 포함하므로 네 가지 조건이 모두 독립적인 것은 아님

3. 교착 상태의 발생 조건

■ 교착 상태의 예

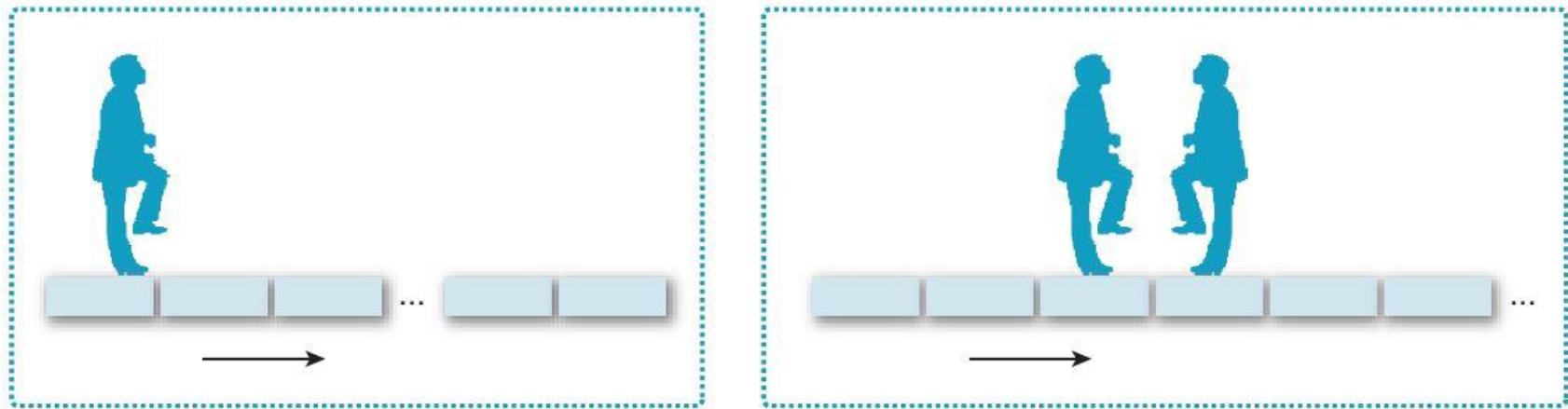


그림 5-6 강 건너기 예로 살펴본 교착 상태

- 상호배제 : 돌 하나를 한 사람만 딛을 수 있음
- 점유와 대기 : 각 사람은 돌 하나를 딛고 다음 돌을 요구
- 비선점 : 사람이 딛고 있는 돌을 강 제로 제거할 수 없음
- 순환 대기 : 원쪽에서 오는 사람은 오른쪽에서 오는 사람 기다리고, 오른쪽에서 오는 사람도 원쪽에서 오는 사람 기다림

■ 교착 상태 해결 방법

- ① 돌 중 한 사람이 되돌아간다(복귀).
- ② 징검다리 반대편을 먼저 확인하고 출발한다.
- ③ 강의 한편에 우선순위를 부여

4. 교착 상태의 표현

■ 교착 상태의 표현

- 시스템 자원 할당 그래프인 방향 그래프 표현

- 자원 할당 그래프

- $G = (V, E)$ 로 구성, 정점 집합 V 는 프로세스 집합 $P = \{P_1, P_2, \dots, P_n\}$ 과 자원 집합 $R = \{R_1, R_2, \dots, R_n\}$ 으로 나뉘. 간선 집합 E 는 원소를 (P_i, R_j) 나 (R_j, P_i) 와 같은 순서쌍으로 나타냄
- 예

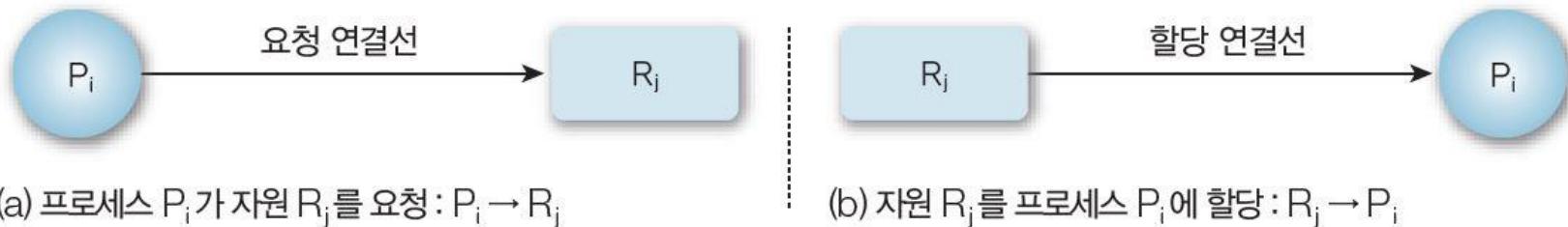


그림 5-7 자원 할당 그래프

4. 교착 상태의 표현

- 사이클이 있어 교착 상태인 자원 할당 그래프

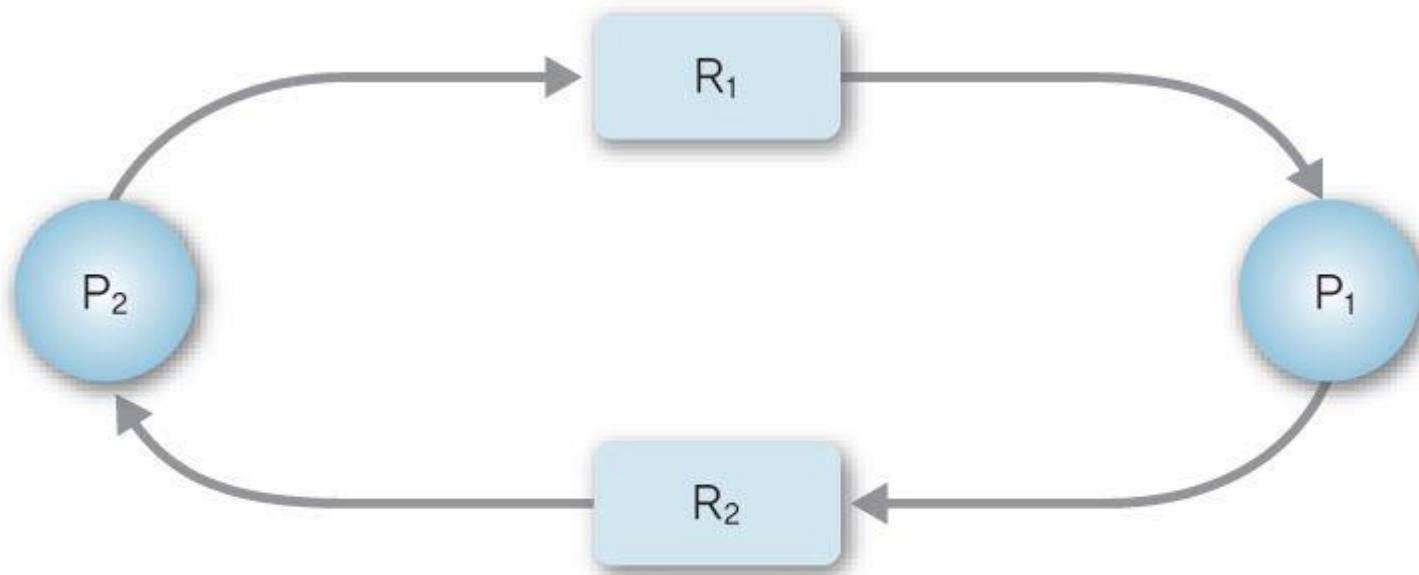
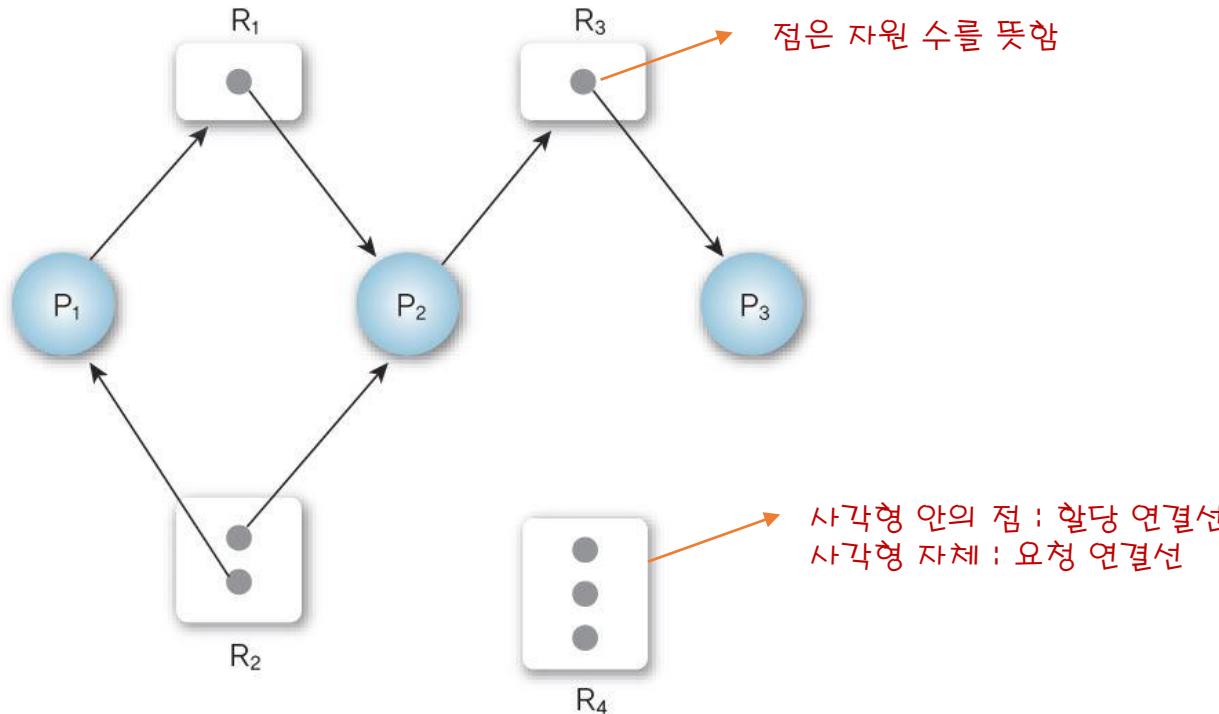


그림 5-8 사이클이 있어 교착 상태인 자원 할당 그래프 : 프로세스 P₁, P₂가 교착 상태

4. 교착 상태의 표현

■ 자원 할당 그래프의 예



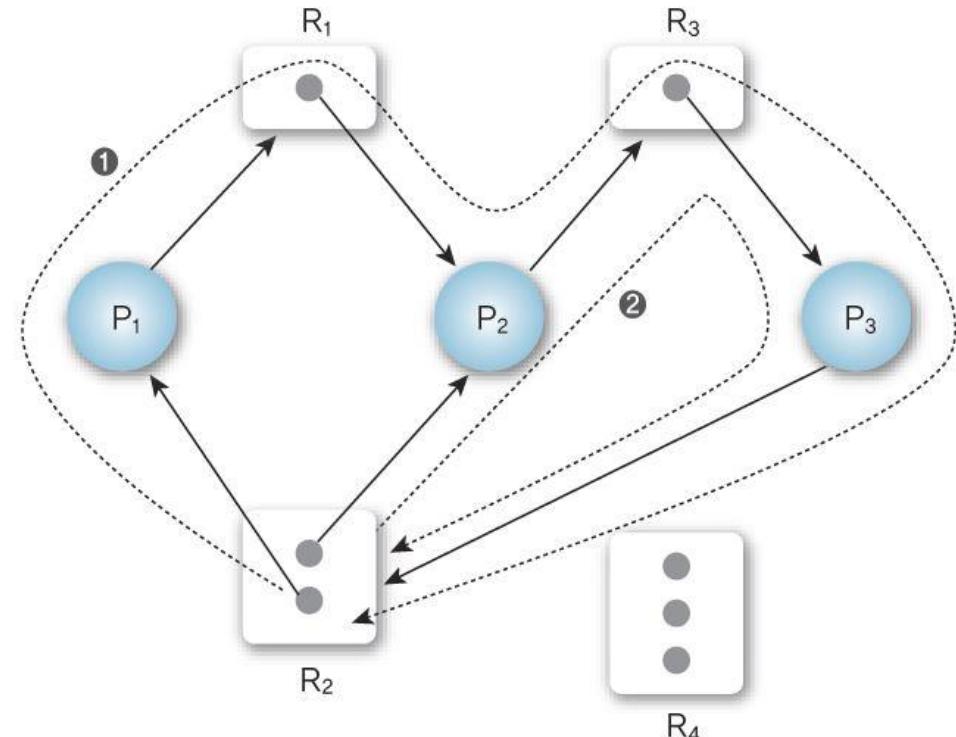
- ① 집합 P, R, E
- $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_1 \rightarrow R_2, P_2 \rightarrow R_3, P_2 \rightarrow R_4, P_3 \rightarrow R_3, P_3 \rightarrow R_4, R_1 \rightarrow P_1, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_2 \rightarrow P_3, R_3 \rightarrow P_2, R_3 \rightarrow P_3, R_4 \rightarrow P_1, R_4 \rightarrow P_2\}$

- ② 프로세스의 상태
- 프로세스 P_1 은 자원 R_2 의 자원을 하나 점유하고, 자원 R_1 을 기다린다.
 - 프로세스 P_2 는 자원 R_1 과 R_2 의 자원을 각각 하나씩 점유하고, 자원 R_3 을 기다린다.
 - 프로세스 P_3 은 자원 R_3 의 자원 하나를 점유 중이다.

그림 5-9 자원 할당 그래프 예와 프로세스 상태

4. 교착 상태의 표현

■ 교착 상태의 할당 그래프와 사이클



(a) 교착 상태의 할당 그래프

① $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
② $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

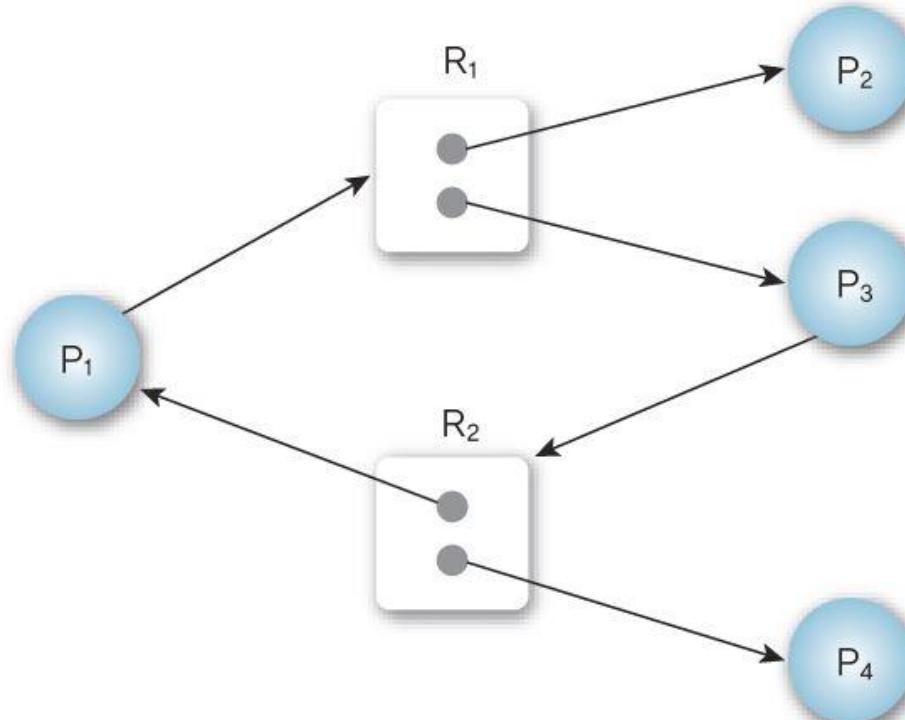
(b) (a)에 있는 사이클

그림 5–10 교착 상태의 할당 그래프와 사이클

그래프에 있는 사이클은 교착 상태 발생의 필요 조건이지 충분조건 아님

4. 교착 상태의 표현

- 사이클이 있으나 교착 상태가 아닌 할당 그래프



(a) 교착 상태의 할당 그래프

⋮

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

(b) (a)에 있는 사이클

그림 5-11 사이클이 있으나 교착 상태가 아닌 할당 그래프

자원 할당 그래프에 사이클이 없다면 교착 상태 아님
그러나 사이클이 있다면 시스템은 교착 상태일 수도 있고 아닐 수도 있음

Section 02 교착 상태의 해결 방법

■ 교착 상태 해결 방법 세 가지

① 예방 prevention

② 회피 avoidance

③ 탐지 detection, 회복

1. 교착 상태 예방

■ 자원의 상호배제 조건 방지

- 상호배제는 자원의 비공유 전제 되어야 함
- 일반적으로 상호배제 조건 만족하지 않으면 교착 상태 예방 불가능

■ 점유와 대기 조건 방지

- 점유와 대기^{hold and wait} 조건 발생 않으려면, 프로세스가 작업 수행 전에 필요한 자원 모두 요청하고 획득해야 함
- 대기상태에서는 프로세스가 자원 점유 불가능하므로 대기조건 성립 안됨(최대자원 할당)
- 점유와 대기 조건 방지 방법
 - 자원 할당 시 시스템 호출된 프로세스 하나를 실행하는 데 필요한 모든 자원 먼저 할당, 실행 후 다른 시스템 호출에 자원 할당
 - 프로세스가 자원을 전혀 갖고 있지 않을 때만 자원 요청할 수 있도록 허용하는 것. 프로세스가 자원을 더 요청하려면 자신에게 할당된 자원을 모두 해제해야 함
- 점유와 대기 조건 방지 방법의 단점
 - 자원 효율성 너무 낮음.
 - 기아 상태 발생 가능(대화식 시스템에서 사용 불가)

1. 교착 상태 예방

■ 비선점 조건 방지 non-preemption

- 이미 할당된 자원에 선점권이 없어야 한다 전제 조건 필요
- 어떤 자원을 가진 프로세스가 다른 자원 요청할 때 요청한 자원을 즉시 할당 받을 수 없어 대기해야 한다면, 프로세스는 현재 가진 자원 모두 해제. 그리고 프로세스가 작업 시작할 때는 요청한 새로운 자원과 해제한 자원 확보해야 함. 이런 방법은 비선점 조건을 효과적으로 무효화시키지만 이미 실행한 작업의 상태를 잊을 수도 있음. 따라서 작업 상태를 쉽게 저장, 복구 할 수 있을 때나 빈번하게 발생하지 않을 때만 좋은 방법
- 전용 입출력장치 등을 빼앗아 다른 프로세스에 할당 후 복구하는 과정 간단하지 않음. 대안으로 프로세스가 어떤 자원을 요청할 때 요청한 자원이 사용 가능한지 검사, 사용할 수 있다면 자원 할당. 사용할 수 없다면 대기 프로세스가 요청한 자원을 점유하고 있는지 검사. 요청한 자원을 대기 프로세스가 점유하고 있다면, 자원을 해제하고 요청 프로세스에 할당. 요청한 자원을 사용할 수 없거나 실행 중인 프로세스가 점유하고 있다면 요청 프로세스는 대기. 프로세스가 대기하는 동안 다른 프로세스가 점유한 자원을 요청하면 자원을 해제할 수 있음.
- 또 다른 방법은 두 프로세스에 우선순위 부여하고 높은 우선순위의 프로세스가 그보다 낮은 우선 순위의 프로세스가 점유한 자원 선점하여 해결. 이 방법은 프로세서 레지스터나 기억장치 레지스터와 같이 쉽게 저장되고 이후에 다시 복원하기 쉬운 자원에 사용. 프린터, 카드 판독기, 테이프 드라이버 같은 자원에는 적용 않음

1. 교착 상태 예방

■ 순환(환형) 대기 조건 방지

- 모든 자원에 일련의 순서 부여, 각 프로세스가 오름차순으로만 자원을 요청할 수 있게 함
- 이는 계층적 요청 방법으로 순환 대기 circular wait의 가능성 제거하여 교착 상태 예방.
- 예상된 순서와 다르게 자원을 요청하는 작업은 실제로 자원을 사용하기 전부터 오랫동안 자원 할당받은 상태로 있어야하므로, 상당한 자원 낭비 초래
- 자원 집합을 $R = \{R_1, R_2, \dots, R_n\}$ 이라고 하자. 각 자원에 고유 숫자 부여 어느 자원의 순서가 빠른지 알 수 있게 함
- 이것은 1:1 함수 $F: R \rightarrow N$ 으로 정의(여기서 N 은 자연수 집합 의미).
- 자원 R 의 집합이 CD 드라이브, 디스크 드라이브, 프린터 포함한다면 함수 F 는 다음과 같이 정의

$$F(\text{CD 드라이브}) = 2,$$

$$F(\text{디스크 드라이브}) = 4,$$

$$F(\text{프린터}) = 7$$

1. 교착 상태 예방

■ 교착 상태 예방시 고려할 규칙

- 각 프로세스는 오름차순으로만 자원 요청 가능
- 즉, 프로세스는 임의의 자원 R_i 요청할 수 있지만 그 다음부터는 $F(R_j) > F(R_i)$ 일 때만 자원 R_j 요청 가능. 데이터 형태 자원이 여러 개 필요하다면, 우선 요청할 형태 자원 하나 정해야 함. 앞서 정의한 함수 이용하면, CD 드라이브와 프린터를 동시에 사용해야 하는 프로세스는 CD 드라이브 먼저 요청 후 프린터 요청
- 또 다른 해결 방법으로 프로세스가 자원 R_j 요청 때마다 $F(R_i) \geq F(R_j)$ 가 되도록 R_j 의 모든 자원 해제하는 것. 그러면 순환 대기 조건 막을 수 있음.
- 순환 대기 성립한다는 가정하에서 사실의 성립(모순에 의한 증명).
 - 순환 대기의 프로세스 집합을 $\{P_0, P_1, \dots, P_n\}$ 이라고 하자. 이때 P_i 는 프로세스 P_{i+1} 이 점유 한 자원 R_i 대기(첨자는 모듈러 n . P_n 은 P_0 이 점유한 자원 R_n 을 대기. 모듈러 관계이므로 한 바퀴 뒤로 돌면 P_n 은 R_n 을 기다리고, P_0 이 R_n 을 갖게 됨). 프로세스 P_{i+1} 은 자원 R_i 를 요청하는 동안 자원 R_i 를 점유하므로 모든 i 에 대하여 $F(R_i) < F(R_{i+1})$ 이 성립 되도록 해야 함. 이는 $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$ 이 성립함 의미. 즉, $F(R_i) < F(R_{i+1})$ 은 불가능. 그러므로 여기서는 순환 대기 불가능. 함수 F 는 시스템에 있는 자원의 정상적인 이용 순서에 따라서 정의해야 함.
- 계층적 요청은 순환 대기 조건 가능성 제거하여 교착 상태 예방, 반드시 자원의 번호 순서로 요청해야 함. 또 번호 부여할 때 실제로 자원 사용하는 순서를 반영해야 한다

2. 교착 상태 회피

■ 교착 상태 회피의 개념

- 목적 : 덜 엄격한 조건 요구하여 자원 좀 더 효율적 사용
- 교착 상태의 모든 발생 가능성을 미리 제거하는 것이 아닌 교착 상태 발생할 가능성 인정하고 (세 가지 필요조건 허용), 교착 상태가 발생하려고 할 때 적절히 회피하는 것
- 예방보다는 회피가 더 병행성 허용
- 교착 상태의 회피 방법
 - 프로세스의 시작 중단
 - 프로세스의 요구가 교착 상태 발생시킬 수 있다면 프로세스 시작 중단
 - 자원 할당 거부(알고리즘 Banker's algorithm)
 - 프로세스가 요청한 자원 할당했을 때 교착 상태 발생할 수 있다면 요청한 자원 할당 않음

2. 교착 상태 회피

■ 시스템의 상태

- 안정 상태와 불안정 상태로 구분
- 교착 상태는 불안정 상태에서 발생
- 모든 사용자가 일정 기간 안에 작업을 끝낼 수 있도록 운영체제가 할 수 있으면 현재 시스템의 상태가 안정, 그렇지 않으면 불안정
- 교착 상태는 불안정 상태. 그러나 모든 불안정 상태가 교착 상태인 것은 아님, 단지 불안정 상태는 교착 상태가 되기 쉬움.
- 상태가 안정하다면 운영체제는 불안정 상태와 교착 상태를 예방 가능
- 불안정 상태의 운영체제는 교착 상태 발생시키는 프로세스의 자원 요청 방지 불가

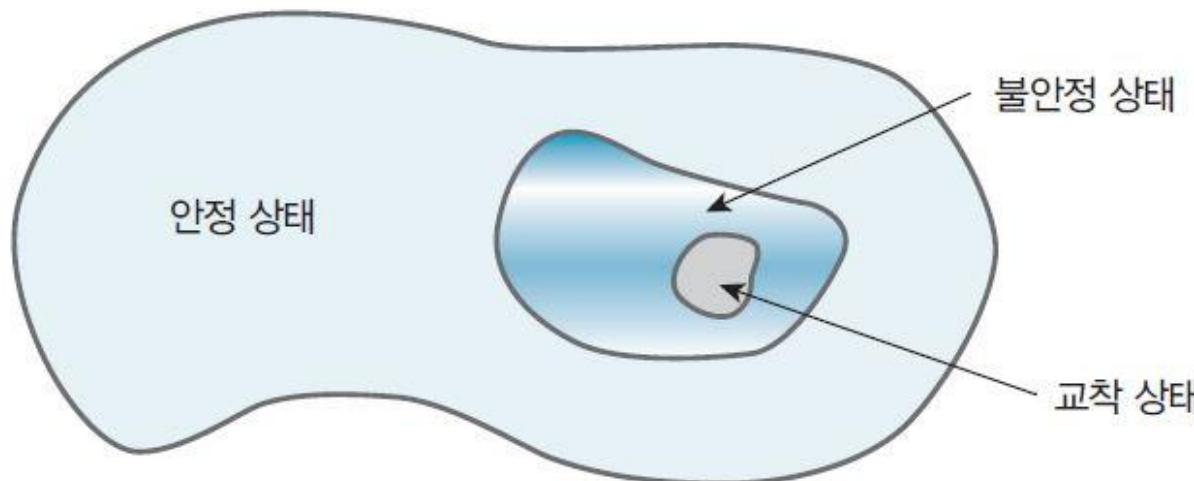


그림 5-12 안정 상태와 불안정 상태, 교착 상태의 공간

2. 교착 상태 회피

■ 안정 상태와 불안정 상태의 자원 예

프로세스	현재 사용량(t_0 시간)	최대 사용량
P_0	2	7
P_1	1	8
P_2	2	4
P_3	2	10
여분 자원 수	3	

(a) 안정 상태의 자원 예

그림 5-13 안정 상태와 불안정 상태의 자원 예

프로세스	현재 사용량(t_0 시간)	최대 사용량
P_0	5	7
P_1	1	8
P_2	2	4
P_3	1	7
여분 자원 수	1	

(b) 불안정 상태의 자원 예

불안정 상태는 교착 상태가 발생할 수 있는 가능성이 있다는 의미이지 반드시 교착 상태가 발생한다는 의미는 아님

2. 교착 상태 회피

■ 안정 상태에서 불안정 상태로 변환 예

안정 $\langle P_2, P_0, P_1, P_3 \rangle$		
구분	현재 사용량	최대 사용량
P_0	3	7
P_1	1	8
P_2	2	4
P_3	2	10
사용 가능 자원 수	2	

구분	현재 사용량	최대 사용량
P_0	2	7
P_1	1	8
P_2	2	4
P_3	2	10
사용 가능 자원 수	3	

(a) P_0 에 자원 할당

(b) P_1 에 자원 할당

불안정		
구분	현재 사용량	최대 사용량
P_0	2	7
P_1	2	8
P_2	2	4
P_3	2	10
사용 가능 자원 수	2	

P_2 는 실행 가능하나 P_0, P_1, P_3 은 불안정

그림 5-14 안정 상태에서 불안정 상태로 변화 예

2. 교착 상태 회피

■ 자원 할당 거부(은행가 알고리즘)

- 다익스트라의 은행가 알고리즘 이용
 - 자원의 할당 허용 여부 결정 전에 미리 결정된 모든 자원의 최대 가능한 할당량^{maximum}을 시뮬레이션하여 안전 여부 검사. 그런 다음 대기 중인 다른 모든 활동의 교착 상태 가능성 조사하여 '안전 상태' 여부 검사 확인.
 - 프로세스가 자원 요청 때마다 운영체제로 실행, THE 운영체제의 프로세스 설계 과정에서 개발
 - 자원 요청 승낙이 불안전한 상태에서 시스템을 배치할 수 있다고 판단하면 이 요청을 연기, 거부하여 교착 상태 예방. 따라서 각 프로세스에 자원을 어떻게 할당(자원 할당 순서 조정)할 것인지 정보 필요하므로 각 프로세스가 요청하는 자원 종류의 최대 수 알아야 함. 이 정보 이용 교착 상태 회피 알고리즘 정의 가능. 이는 은행에서 모든 고객이 만족하도록 현금 할당하는 과정과 동일.

2. 교착 상태 회피

■ 은행가 알고리즘 구현을 위한 자료구조

- Available : 각 형태별로 사용 가능한 자원 수(사용 가능량) 표시하는 길이가 m인 벡터
Available[j]=k이면, 자원을 k개 사용할 수 있다는 의미
- Max : 각 프로세스 자원의 최대 요청량(최대 요구량)을 표시하는 $n \times m$ 행렬. Max[i, j] =k이면, 프로세스 P는 자원이 R인 자원을 최대 k개까지 요청할 수 있다는 의미
- Allocation : 현재 각 프로세스에 할당되어 있는 각 형태의 자원 수(현재 할당량) 정의하는 $n \times m$ 행렬.
Allocation[i, j] =k이면, 프로세스 P는 자원이 R인 자원을 최대 k개 할당받고 있다는 의미
- Need : 각 프로세스에 남아 있는 자원 요청(추가 요구량) 표시하는 $n \times m$ 행렬. Need[i, j] =k이면, 프로세스 P는 자신의 작업을 종료하려고 자원 R를 k개 더 요청한다는 의미

N : 시스템의 프로세스 수, m : 자원 수

Need[i, j] = Max[i, j] - Allocation[i, j]라는 식 성립

2. 교착 상태 회피

■ 프로세스 P_i 가 자원 요청 시 일어나는 동작

- 1단계 : $Request_i \leq Need_i$ 이면 2단계로 이동하고, 그렇지 않으면 프로세스가 최대 요청치를 초과하기 때문에 오류 상태가 된다.
- 2단계 : $Request_i \leq Available$ 이면 3단계로 이동하고, 그렇지 않으면 자원이 부족하기 때문에 P_i 는 대기한다.
- 3단계 : 시스템은 상태를 다음과 같이 수정하여 요청된 자원을 프로세스 P_i 에 할당한다.

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

그림 5-15 은행가 알고리즘

자원 할당이 안정 상태라면 처리가 되고 있는 프로세스 P_i 는 자원 할당받음
불안정 상태라면 P_i 는 $Request_i$ 를 대기하고 이전 자원 할당의 상태로 복귀

2. 교착 상태 회피

■ 안전 알고리즘의 시스템 상태 검사

- 1단계 : Work와 Finish를 각각 길이가 m과 n인 벡터라고 하자. Work = Available, $\text{Finish}[i] = \text{false}$, $i = 1, 2, \dots, n$ 이 되도록 초기화한다.
- 2단계 : 다음 조건을 만족하는 i 값을 찾는다. i 값이 없으면 4단계로 이동한다.

$$\text{Finish}[i] == \text{false}$$

$$\text{Need}_i \leq \text{Work}$$

- 3단계 : 다음을 수행하고 2단계로 이동한다.

$$\text{Work} = \text{Work} + \text{Allocation}_i$$

$$\text{Finish}[i] = \text{true}$$

- 4단계 : 모든 i에 대하여 $\text{Finish}[i] == \text{true}$ 이면 시스템은 안정 상태이다.

그림 5-16 안전 알고리즘

2. 교착 상태 회피

■ 시간 t_0 일때 시스템의 상태

안정상태

프로세스	Allocation	Max	Need	Available
	ABCD	ABCD	ABCD	ABCD
P_0	2011	3214	1203	1222
P_1	0121	0252	0131	
P_2	4003	5105	1102	
P_3	0210	1530	1320	
P_4	1030	3033	2003	
할당량	7375			

그림 5-17 시간 t_0 일 때 시스템의 상태

2. 교착 상태 회피

■ 은행가 알고리즘의 단점

- 할당할 수 있는 자원의 일정량 요청. 자원은 수시로 유지 보수 필요, 고장이나 예방 보수 하기 때문에 일정하게 남아 있는 자원 수 파악 곤란
- 사용자 수가 일정해야 하지만 다중 프로그래밍 시스템에서는 사용자 수 항상 변함
- 교착 상태 회피 알고리즘을 실행하면 시스템 과부하 증가
- 프로세스는 자원 보유한 상태로 끝낼 수 없음. 시스템에서는 이보다 더 강력한 보장 필요
- 사용자가 최대 필요량을 미리 알려 주도록 요청하지만, 자원 할당 방법이 점점 동적으로 변하면서 사용자의 최대 필요량 파악 곤란. 게다가 최근 시스템은 편리한 인터페이스를 사용자에게 제공하려고 필요한 자원 몰라도 되는 방법 보편화
- 항상 불안정 상태 방지해야 하므로 자원 이용도 낮음

3. 교착 상태 회복

■ 교착 상태 회복에 필요한 다음 알고리즘

- 시스템 상태 검사하는 교착 상태 탐지 알고리즘
- 교착 상태에서 회복시키는 알고리즘

■ 교착 상태 회복의 특징

- 교착 상태 파악 위해 교착 상태 탐지 알고리즘을 언제 수행해야 하는지 결정하기 어려움
- 교착 상태 탐지 알고리즘 자주 실행하면 시스템의 성능 떨어지지만, 교착 상태에 빠진 프로세스 빨리 발견하여 자원의 유휴 상태 방지 가능. 하지만 자주 실행하지 않으면 반대 상황 발생
- 탐지와 회복 방법은 필요한 정보를 유지하고 탐지 알고리즘을 실행시키는 비용뿐 아니라 교착 상태 회복에 필요한 부담까지 요청

3. 교착 상태 회복

■ 교착 상태 회복 방법

■ 프로세스 중단

- 교착 상태 프로세스 모두 중단 : 교착 상태의 순환 대기를 확실히 해결하지만 자원 사용과 시간면에서 비용 많이 드는다. 오래 연산했을 가능성이 있는 프로세스의 부분 결과 폐기하여 다시 연산해야 함
- 한 프로세스씩 중단 : 한 프로세스 중단할 때마다 교착 상태 탐지 알고리즘 호출하여 프로세스가 교착 상태에 있는지 확인. 교착 상태 탐지 알고리즘을 호출하는 부담이 상당히 크다는 것이 단점
- 프로세스 중단이 쉽지 않을 수도 있음. 프로세스가 파일 업데이트하다가 중단된다면 해당 파일은 부정 확한 상태. 마찬가지로 프로세스가 데이터를 프린터에 인쇄하고 있을 때 중단하면 다음 인쇄를 진행하기 전에 프린터의 상태를 정상 상태로 되돌려야 함
- 최소 비용으로 프로세스들을 중단하는 우선 순위 설정
 - 프로세스가 수행된 시간과 앞으로 종료하는 데 필요한 시간
 - 프로세스가 사용한 자원 형태와 수(예 : 자원을 선점할 수 있는지 여부)
 - 프로세스를 종료하는 데 필요한 자원 수
 - 프로세스를 종료하는 데 필요한 프로세스 수
 - 프로세스가 대화식인지, 일괄식인지 여부

3. 교착 상태 회복

■ 자원 선점

- 선점 자원 선택 : 프로세스를 종료할 때 비용을 최소화하려면 적절한 선점 순서 결정. 비용 요인에는 교착 상태 프로세스가 점유한 자원 수, 교착 상태 프로세스가 지금까지 실행하는데 소요한 시간 등 매개변수가 포함된다.
- 복귀 : 필요한 자원을 잃은 프로세스는 정상적으로 실행 불가. 따라서 프로세스를 안정 상태로 복귀시키고 다시 시작해야 함. 일반적으로 안전 상태 결정 곤란, 완전히 복귀시키고(프로세스 중단) 재시작하는 것이 가장 단순한 방법. 프로세스를 교착 상태에서 벗어날 정도로만 복귀시키는 것이 더 효과적. 그러나 이 방법은 시스템이 실행하는 모든 프로세스의 상태 정보를 유지해야 하는 부담.
- 기아 : 동일한 프로세스가 자원들을 항상 선점하지 않도록 보장하려고 할 때 비용이 기반인 시스템에서는 동일한 프로세스를 희생자로 선택. 그러면 이 프로세스는 작업 완료하지 못하는 기아 상태가 되어 시스템 조치 요청. 따라서 프로세스가 짧은 시간 동안만 희생자로 지정되도록 보장해야 함. 가장 일반적인 해결 방법은 비용 요소에 복귀 횟수를 포함시키는 것