Chapter 05

프로세스 동기화

Contents

학습목표

- 프로세스 간 통신의 개념을 이해하고 종류를 파악한다.
- 공유 자원 사용 시의 임계구역 문제를 알아본다.
- 임계구역 문제를 해결하기 위한 조건과 해결 방법을 알아본다.

1-1 프로세스 간 통신의 개념

프로세스 간 통신의 종류

- 프로세스 내부 데이터 통신
  - 하나의 프로세스 내에 2개 이상의 스레드가 존재하는 경우의 통신
  - 프로세스 내부의 스레드는 전역 변수나 파일을 이용하여 데이터를 주고받음

- 프로세스 간 데이터 통신
  - 같은 컴퓨터에 있는 여러 프로세스끼리 통신하는 경우
  - 공용 파일 또는 운영체제가 제공하는 파이프를 사용하여 통신

- 네트워크를 이용한 데이터 통신
  - 여러 컴퓨터가 네트워크로 연결되어 있을 때 통신
  - 소켓을 이용하여 데이터를 주고받음

1-1 프로세스 간 통신의 개념

프로세스 간 통신의 종류

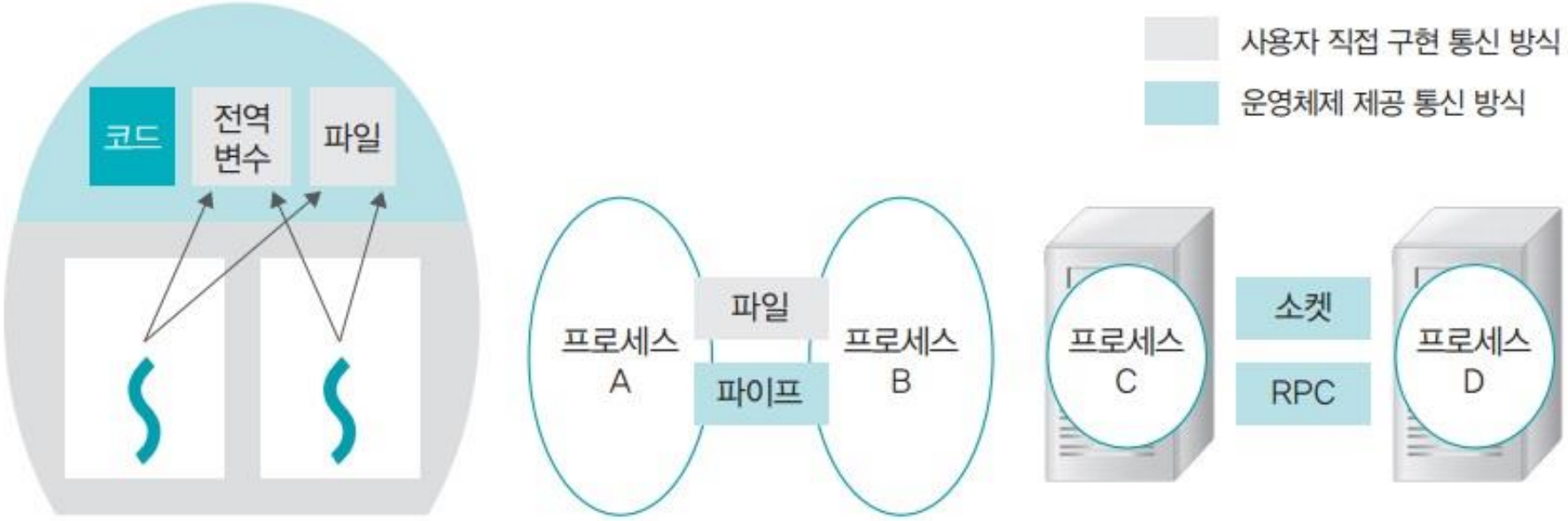

그림 5-1 **프로세스 간 통신의 종류**

1-1 프로세스 간 통신의 개념

통신 방식의 이해

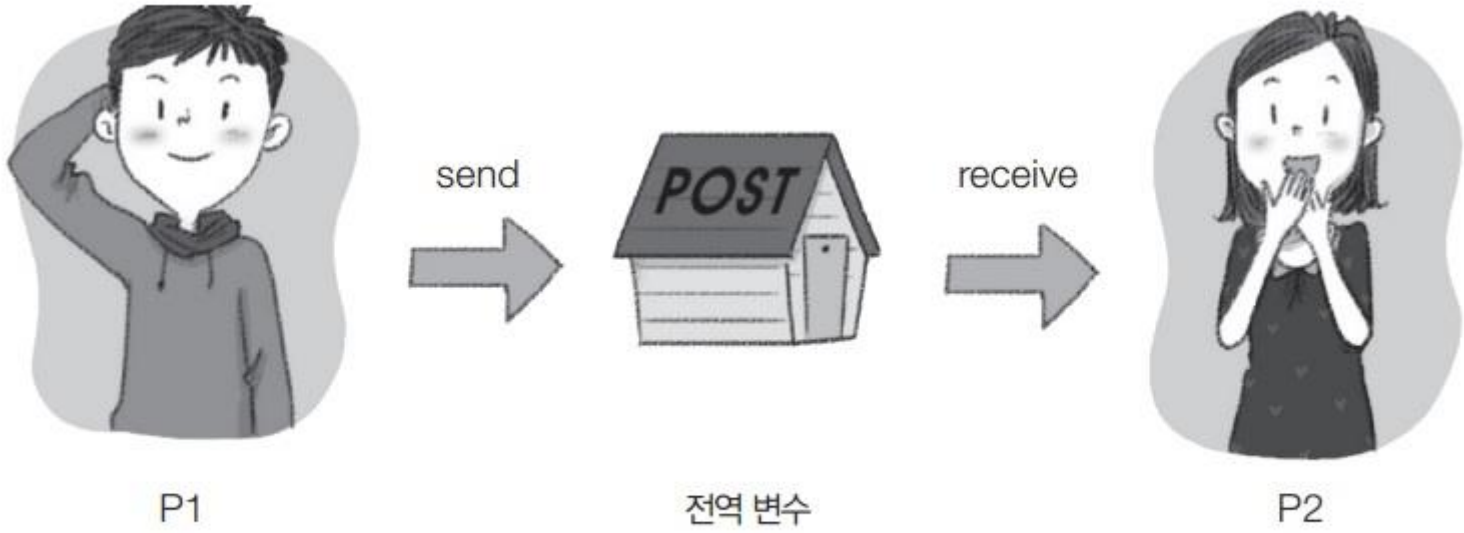

그림 5-2 통신 방식의 이해

1-2 프로세스 간 통신의 분류

통신 방향에 따른 분류

- 양방향 통신
  - 데이터를 동시에 양쪽 방향으로 전송할 수 있는 구조로, 일반적인 통신은 모두 양방향 통신
  - 프로세스 간 통신에서는 소켓 통신이 양방향 통신에 해당

- 반양방향 통신
  - 데이터를 양쪽 방향으로 전송할 수 있지만 동시 전송은 불가능하고 특정 시점에 한쪽 방향으로만 전송할 수 있는 구조
  - 반양방향 통신의 대표적인 예는 무전기

- 단방향 통신
  - 모스 신호처럼 한쪽 방향으로만 데이터를 전송할 수 있는 구조
  - 프로세스 간통신에서는 전역 변수와 파이프가 단방향 통신에 해당

1-2 프로세스 간 통신의 분류

전역 변수는 단방향 통신

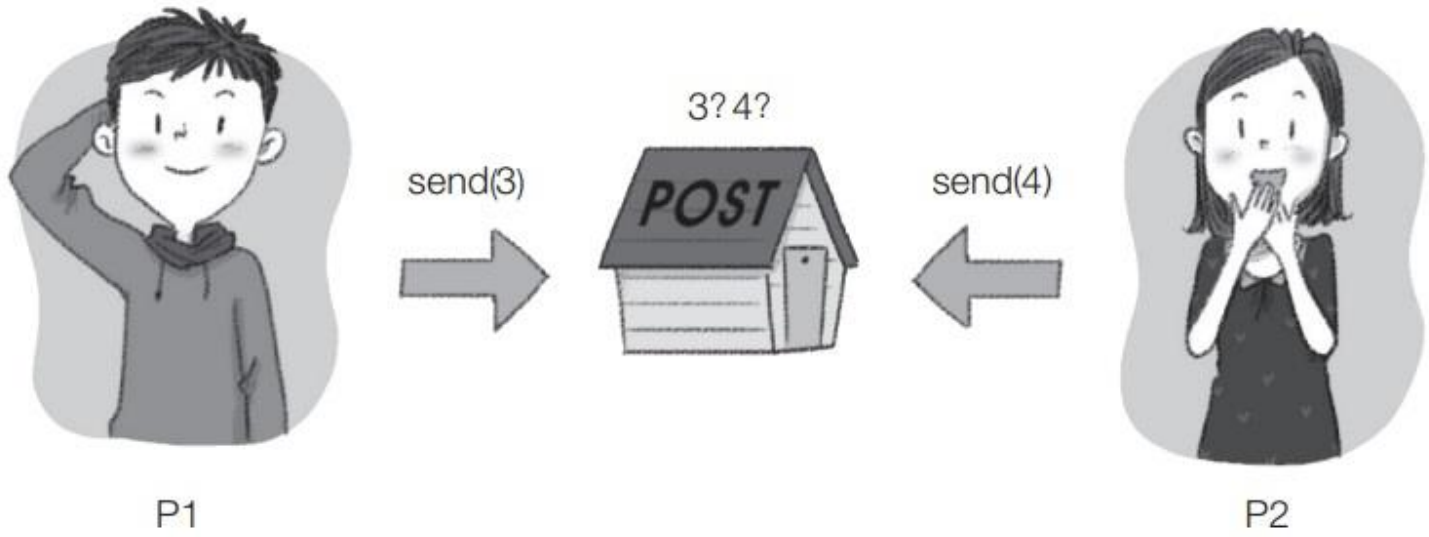

그림 5-3 전역 변수가 단방향 통신인 이유

1-2 프로세스 간 통신의 분류

통신 구현 방식에 따른 분류

- 대기가 있는 통신
  - 동기화를 지원하는 통신 방식
  - 데이터를 받는 쪽은 데이터가 도착할 때까지 자동으로 대기 상태에 머물러 있음

- 대기가 없는 통신
  - 동기화를 지원하지 않는 통신 방식
  - 데이터를 받는 쪽은 바쁜 대기를 사용하여 데이터가 도착했는지 여부를 직접 확인

1-2 프로세스 간 통신의 분류

정리

표 5-1 프로세스 간 통신의 분류

분류 방식	종류	예
통신 방향에 따른 분류	양방향 통신	일반적 통신, 소켓
	반양방향 통신	무전기
	단방향 통신	전역 변수, 파일, 파이프
통신 구현 방식에 따른 분류	대기가 있는 통신(동기화 통신)	파이프, 소켓
	대기가 없는 통신(비동기화 통신)	전역 변수, 파일

1-3 프로세스 간 통신의 종류

프로세스 간 통신 방식

- 데이터를 주거나 받는 쓰기 연산과 읽기 연산으로 이루어짐

```
send → write(GV, message)
receive → read(GV, message)
```

1-3 프로세스 간 통신의 종류

전역 변수를 이용한 통신

- 공동으로 관리하는 메모리를 사용하여 데이터를 주고받는 것
- 데이터를 보내는 쪽에서는 전역 변수나 파일에 값을 쓰고, 데이터를 받는 쪽에서는 전역 변수의 값을 읽음

```
int GV;

int main()
{  int pid;

   pid=fork();
       ⋮
```

그림 5-4 전역 변수를 이용한 통신

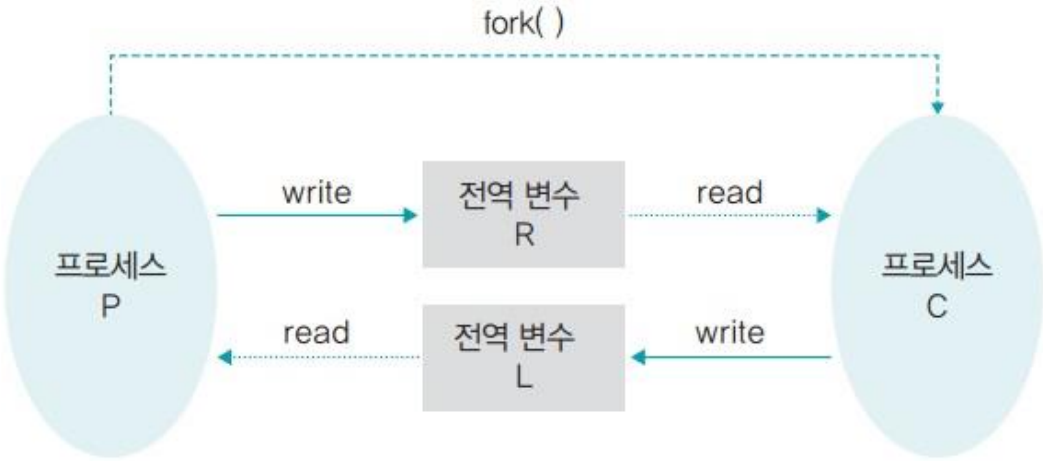

그림 5-5 전역 변수를 이용한 양방향 통신

1-3 프로세스 간 통신의 종류

파일을 이용한 통신

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{   int fd;
    char buf[5];

    fd=open("com.txt", O_RDWR);     /* init */

    write(fd, "Test", 5);

    read(fd, buf, 5);

    close(fd);

    exit(0);
}
```

하드디스크로 쓰기

하드디스크에서 읽기

그림 5-6 **파일 입출력 코드**

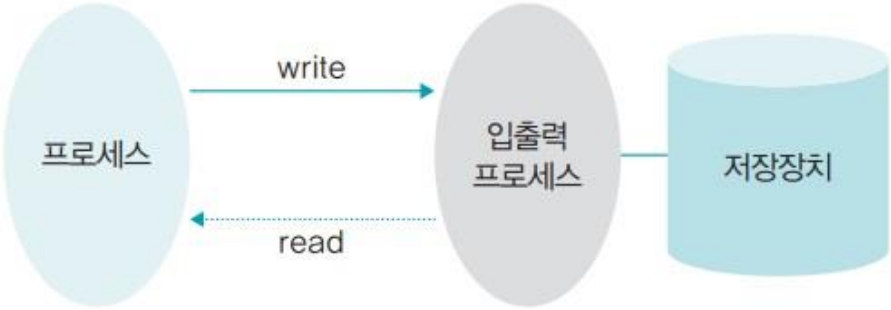

그림 5-7 **파일 입출력 연산**

1-3 프로세스 간 통신의 종류

파일을 이용한 통신

- 파일 열기
  - **open(“com.txt”, O_RDWR)** : com.txt 파일을 읽기와 쓰기를 할 수 있는 형태로 준비
  - 파일이 열리면 open 함수는 그 파일에 접근할 수 있는 권한인 파일 기술자 fd를 사용자에게 반환
- 읽기 또는 쓰기 연산
  - **write(fd, “Test”, 5)** : fd, 즉 com.txt 파일에 Test라는 문자열을 쓰라는 뜻
  - **read(fd, buf, 5)** : fd, 즉 com.txt 파일에서 5B를 읽어 변수 buf에 저장
- 파일 닫기
  - **close(fd)** : fd가 가리키는 파일, 즉 com.txt 파일을 닫음

1-3 프로세스 간 통신의 종류

파이프를 이용한 통신

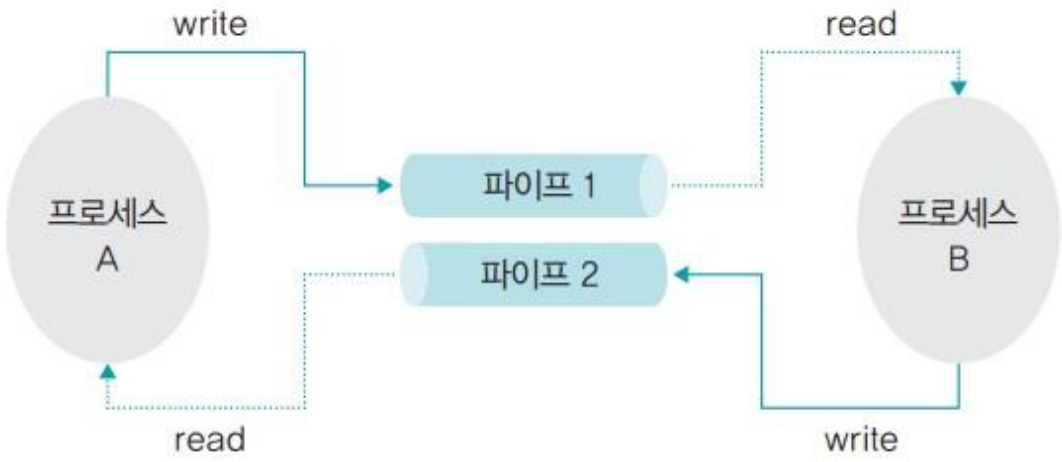

그림 5-8 **파이프를 이용한 통신**

- 운영체제가 제공하는 동기화 통신 방식으로, 파일 입출력과 같이 open() 함수로 기술자를 얻고 작업을 한 후 close() 함수로 마무리
- 파이프로 양방향 통신을 하려면 파이프 2개 사용
- 파이프에 쓰기 연산을 하면 데이 터가 전송되고 읽기 연산을 하면 데이터를 받음

1-3 프로세스 간 통신의 종류

이름 없는 파이프

- 일반적으로 파이프라고 하면 이름 없는 파이프를 가리킴

이름 있는 파이프

- FIFO라 불리는 특수 파일을 이용하며 서로 관련 없는 프로세스 간 통신에 사용

1-3 프로세스 간 통신의 종류

소켓을 이용한 통신

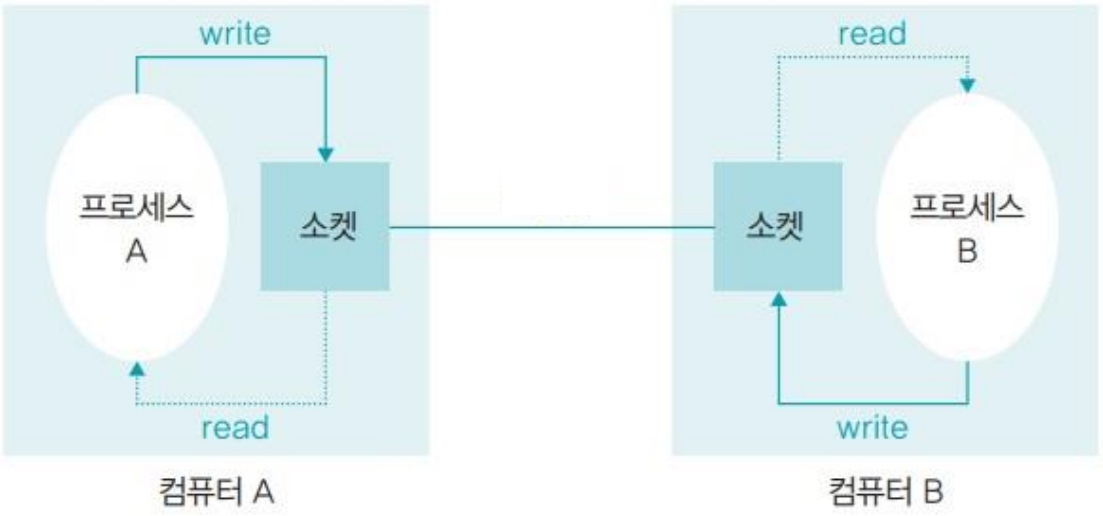

그림 5-9 소켓을 이용한 통신

- 여러 컴퓨터에 있는 프로세스끼리 통신하는 방법
- 통신하고자 하는 프로세스는 자신의 소켓과 상대의 소켓을 연결
- 시스템에 있는 프로세스가 소켓을 바인딩한 후 소켓에 쓰기 연산을 하면 데이터가 전송되고, 읽기 연산을 하면 데이터를 받게 됨

1-3 프로세스 간 통신의 종류

요약

표 5-2 프로세스 간 통신 요약

종류	운영체제 동기화 지원	open()/close() 사용
전역 변수	×(바쁜 대기)	X
파일	×(wait() 함수 이용)	O
파이프	O	O
소켓	O	O

2-1 공유 자원의 접근

공유 자원

- 여러 프로세스가 공동으로 이용하는 변수, 메모리, 파일 등을 말함
- 공동으로 이용되기 때문에 누가 언제 데이터를 읽거나 쓰느냐에 따라 그 결과가 달라질 수 있음

경쟁 조건

- 2개 이상의 프로세스가 공유 자원을 병행적으로 읽거나 쓰는 상황
- 경쟁 조건이 발생하면 공유 자원 접근 순서에 따라 실행 결과가 달라질 수 있음

2-1 공유 자원의 접근

공유 자원의 접근 예

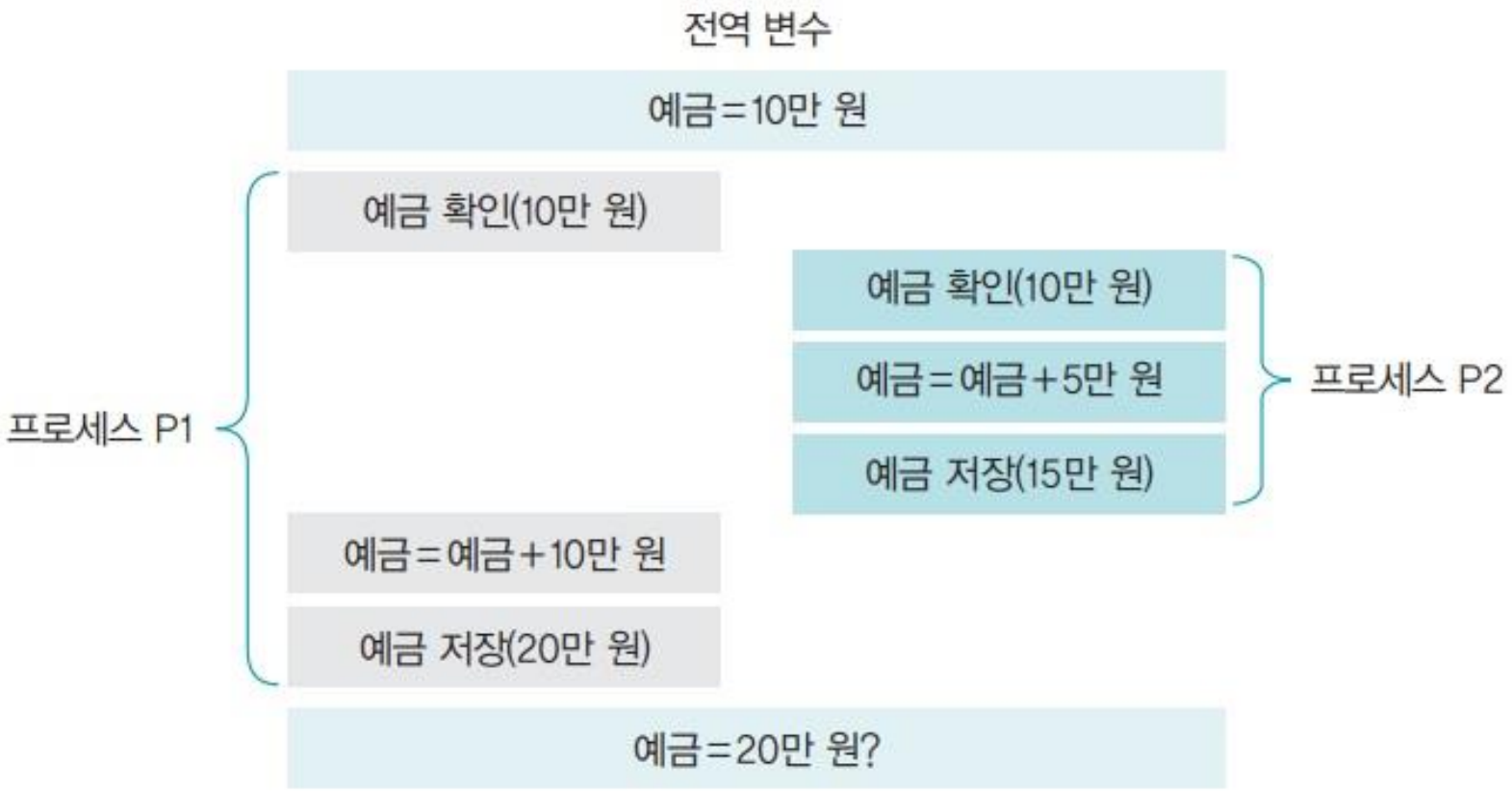

그림 5-10 공유 자원의 접근

2-2 임계 구역

임계 구역

- 공유 자원 접근 순서에 따라 실행 결과가 달라지는 프로그램의 영역
- 믹서는 공유가 불가능한 자원으로서 주방의 임계구역
- 임계구역에서는 프로세스들이 동시에 작업하면 안 됨
- 어떤 프로세스가 임계구 역에 들어가면 다른 프로세스는 임계구역 밖에서 기다려야 하며 임계구역의 프로세스가 나와야 들어갈 수 있음

그림 5-11 가스레인지와 믹서

2-3 생산자-소비자 문제

코드 및 실행 순서에 따른 결과

- 생산자는 수를 증가시켜가며 물건을 채우고 소비자는 생산자를 쫓아가며 물건을 소비
- 생산자 코드와 소비자 코드가 동시에 실행되면 문제가 발생

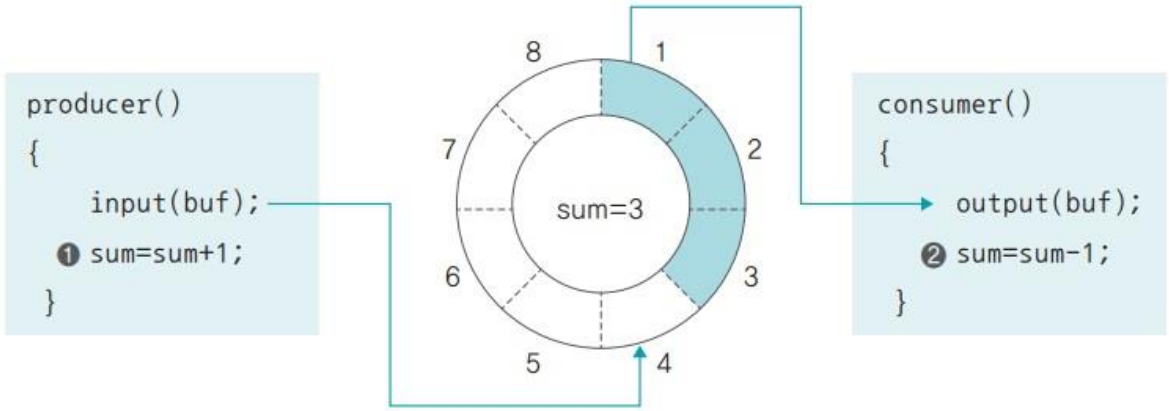

그림 5-12 생산자-소비자 문제

❶ 생산자 sum = 4
❷ 소비자 sum = 2
결과 sum = 2

(a) ❶, ❷ 순서로 실행

❷ 소비자 sum = 2
❶ 생산자 sum = 4
결과 sum = 4

(b) ❷, ❶ 순서로 실행

그림 5-13 실행 순서에 따른 결과 차이

2-3 생산자-소비자 문제

하드웨어 자원을 공유하면 발생하는 문제

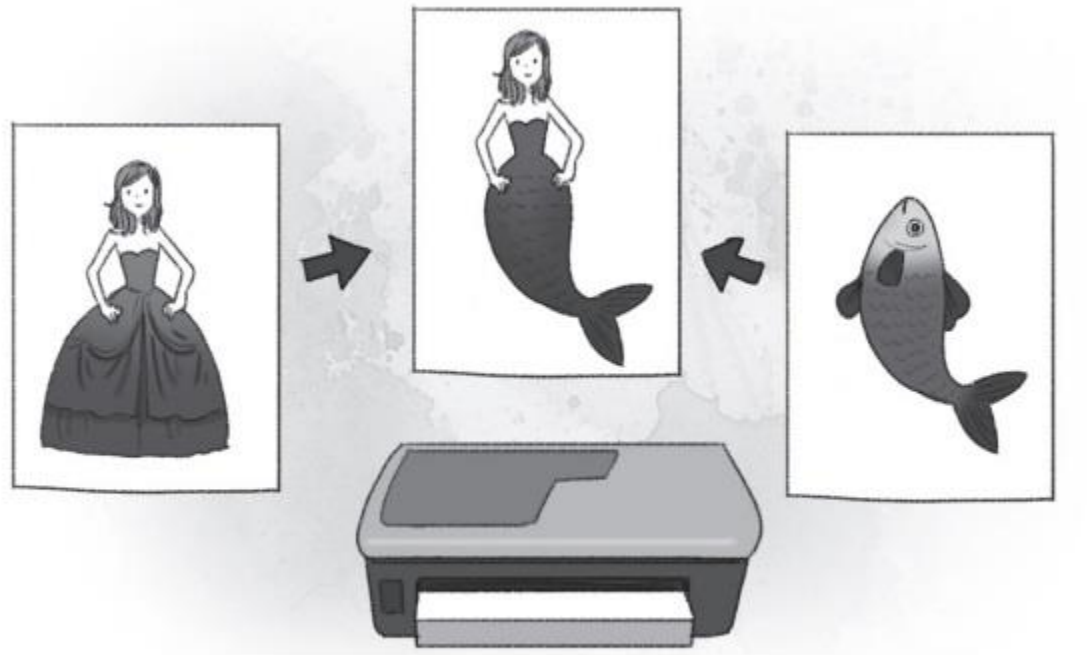

그림 5-14 하드웨어 자원을 공유하면 발생하는 문제

2-4 임계구역 해결 조건

상호 배제mutual exclusion

- 한 프로세스가 임계구역에 들어가면 다른 프로세스는 임계구역에 들어갈 수 없는 것

한정 대기bounded waiting

- 어떤 프로세스도 무한 대기하지 않아야 함

진행의 융통성progress flexibility

- 한 프로세스가 다른 프로세스의 진행을 방해해서는 안 된다는 것

3-1 기본 코드 소개

임계구역 해결 방법을 설명하기 위한 기본 코드

```
#include <stdio.h>

typedef enum {false, true} boolean;
extern boolean lock=false;
extern int balance;

main() {
     while(lock==true);
     lock=true;
     balance=balance+10;    /* 임계구역 */
     lock=false;
}
```

그림 5-15 기본 코드

3-2 임계구역 해결 조건을 고려한 코드 설계

전역 변수로 잠금을 구현한 코드

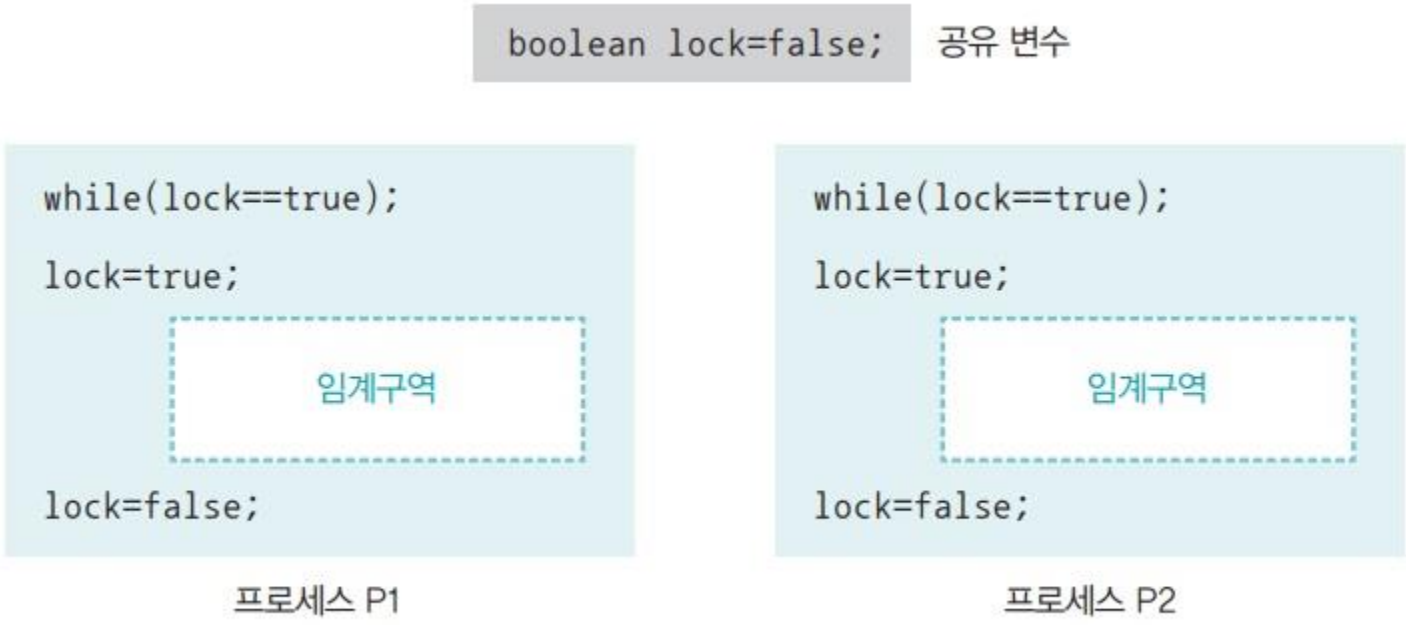

그림 5-16 **전역 변수로 잠금을 구현한 코드**

3-2 임계구역 해결 조건을 고려한 코드 설계

전역 변수로 잠금을 구현한 코드의 문제

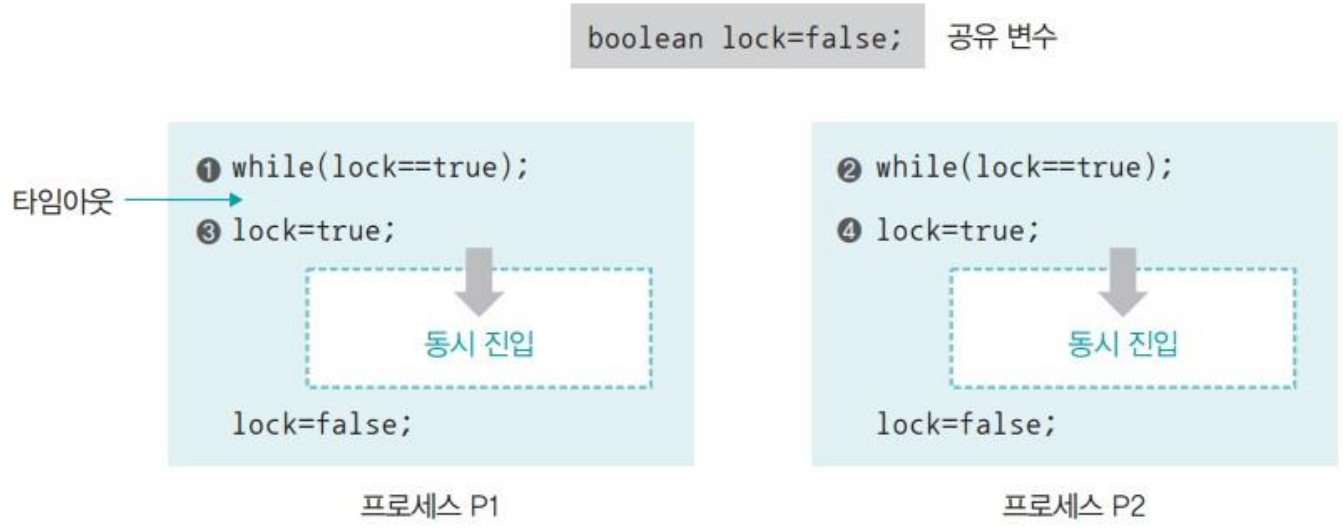

그림 5-17 동시 진입 상황(상호 배제 조건을 충족하지 않는 경우)

❶ 프로세스 P1은 while(lock==true); 문을 실행

❷ 프로세스 P2는 while(lock==true); 문을 실행

❸ 프로세스 P1은 lock=true; 문을 실행하여 임계구역에 잠금을 걸고 진입

❹ 프로세스 P2도 lock=true; 문을 실행하여 임계구역에 잠금을 걸고 진입(결국 둘 다 임계 구역에 진입)

3-2 임계구역 해결 조건을 고려한 코드 설계

상호 배제 조건을 충족하는 코드

```
boolean lock1=false;
boolean lock2=false;
```

공유 변수

```
lock1=true;
while(lock2==true);
        임계구역
lock1=false;
```

프로세스 P1

```
lock2=true;
while(lock1==true);
        임계구역
lock2=false;
```

프로세스 P2

그림 5-18 상호 배제 조건을 충족하는 코드

3-2 임계구역 해결 조건을 고려한 코드 설계

상호 배제 조건을 충족하는 코드의 문제

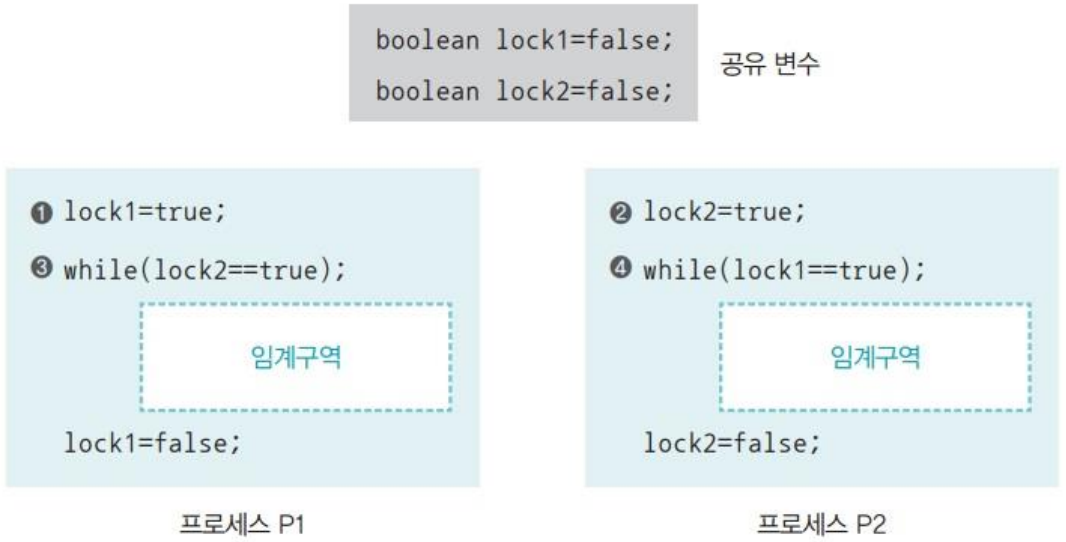

그림 5-19 무한 대기 상황(한정 대기 조건을 충족하지 않는 경우)

- 프로세스 P1은 lock1=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 프로세스 P2가 실행 상태로 바뀜
- 프로세스 P2도 lock2=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 프로세스 P1이 실행 상태로 바뀜
- 프로세스 P2가 lock2=true; 문을 실행했기 때문에 프로세스 P1은 while(lock2==true); 문에서 무한 루프에 빠짐
- 프로세스 P1이 lock1=true; 문을 실행했기 때문에 프로세스 P2도 while(lock1 ==true); 문에서 무한 루프에 빠짐

3-2 임계구역 해결 조건을 고려한 코드 설계

상호 배제와 한정 대기 조건을 충족하는 코드

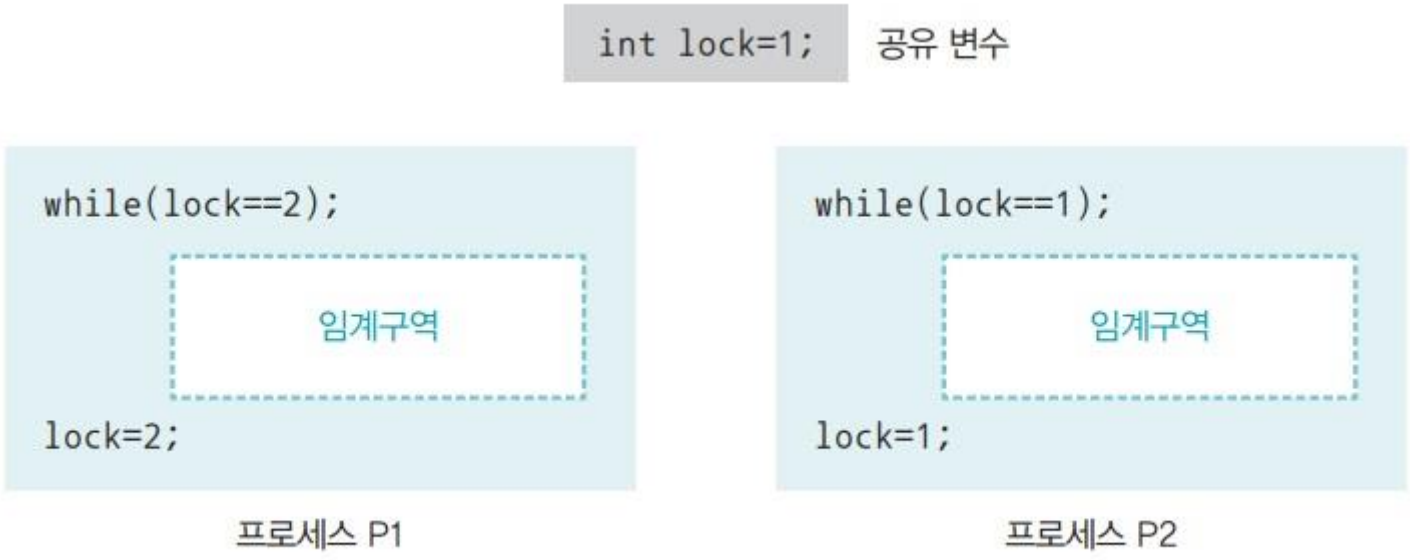

그림 5-20 상호 배제와 한정 대기 조건을 충족하는 코드(진행의 융통성 조건을 충족하지 않는 경우)

3-2 임계구역 해결 조건을 고려한 코드 설계

임계구역 문제의 하드웨어적인 해결 방법

- 검사와 지정test-and-set 코드로 하드웨어의 지원을 받아 while(lock==true); 문과 lock=true; 문을 한꺼번에 실행
- 검사와 지정 코드를 이용하면 명령어 실행 중간에 타임아웃이 걸려 임계구역을 보호하지 못하는 문제가 발생하지 않음

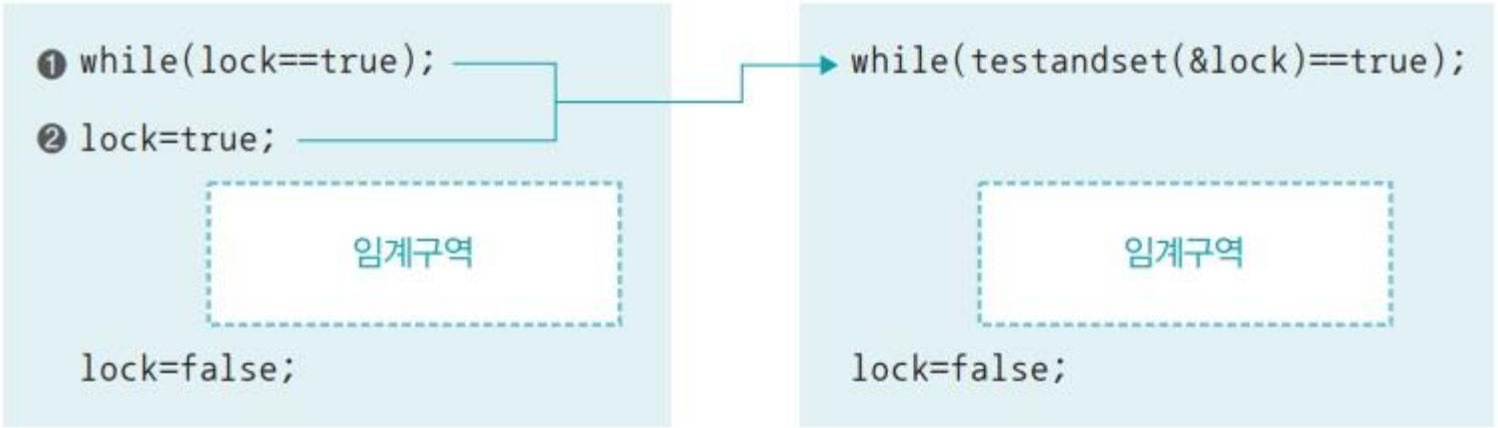

그림 5-21 검사와 지정을 이용한 코드

3-3 피터슨 알고리즘

피터슨 알고리즘

- 임계구역 해결의 세 가지 조건을 모두 만족
- 2개의 프로세스만 사용 가능하다는 한계가 있음

```
boolean lock1=false;
boolean lock2=false;
int turn=1;
```

공유 변수

```
lock1=true;
turn=2;
While(lock2==true && turn==2);
        임계구역
lock1=false;
```

프로세스 P1

```
lock2=true;
turn=1;
While(lock1==true && turn==1);
        임계구역
lock2=false;
```

프로세스 P2

그림 5-22 **피터슨 알고리즘**

3-4 데커 알고리즘

데커 알고리즘

```
boolean lock1=false;
boolean lock2=false;
int turn=1;
```

공유 변수

```
lock1=true;
while(lock2==true)
{  if(turn==2)  {
       lock1=false;
       while(turn==2);
       lock1=true;  }   /* end if */
}   /* end while */
       임계구역
turn=2;
lock1=false;
```

프로세스 P1

```
lock2=true;
while(lock1==true)
{  if(turn==1)  {
       lock2=false;
       while(turn==1);
       lock2=true;  }   /* end if */
}   /* end while */
       임계구역
turn=1;
lock2=false;
```

프로세스 P2

그림 5-23 **데커 알고리즘**

3-4 데커 알고리즘

데커 알고리즘의 동작

❶ 프로세스 P1은 우선 잠금을 검(lock1=true;)

❷ 프로세스 P2의 잠금이 걸렸는지 확인[while(lock2==true)]

❸ 만약 프로세스 P2도 잠금을 걸었다면 누가 먼저인지 확인[if(turn ==2)]

만약 프로세스 P1의 차례라면(turn =1) 임계구역으로 진입

만약 프로세스 P2의 차례라면(turn =2) ④로 이동

❹ 프로세스 P1은 잠금을 풀고(lock1 =false;) 프로세스 P2가 작업을 마칠 때까지 기다림 [while(turn = =2);]

프로세스 P2가 작업을 마치면 잠금을 걸고(lock1 =true;) 임계구역으로 진입

3-5 세마포어

세마포어

- 임계구역에 진입하기 전에 스위치를 사용 중으로 놓고 임계구역으로 들어감
- 이후에 도착하는 프로세스는 앞의 프로세스가 작업을 마칠 때까지 기다림
- 프로세스가 작업을 마치면 다음 프로세스에 임계구역을 사용하라는 동기화 신호를 보냄

그림 5-24 세마포어와 토글 스위치

```
Semaphore(n);
P();
        임계구역
V();
```

그림 5-25 세마포어 코드

3-5 세마포어

세마포어 내부 코드

- **Semaphore(n)** : 전역 변수 RS를 n으로 초기화, RS에는 현재 사용 가능한 자원의 수가 저장
- **P()** : 잠금을 수행하는 코드로 RS가 0보다 크면(사용 가능한 자원이 있으면) 1만큼 감소시키고 임계구역에 진입, 만약 RS가 0보다 작으면(사용 가능한 자원이 없으면) 0보다 커질 때까지 기다림
- **V()** : 잠금 해제와 동기화를 같이 수행하는 코드로, RS 값을 1 증가시키고 세마포어에서 기다리는 프로세스에게 임계구역에 진입해도 좋다는 wake_up 신호를 보냄

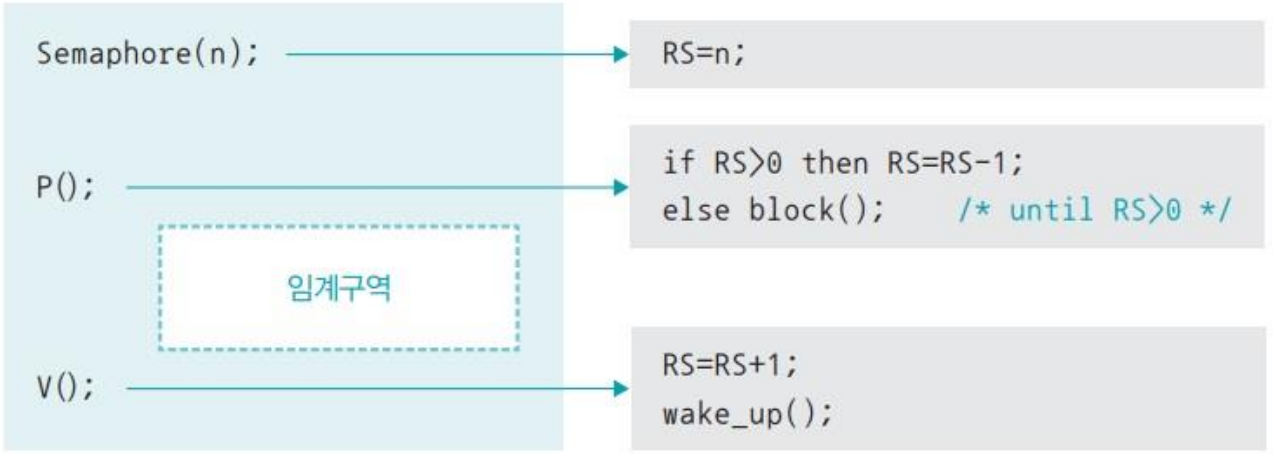

그림 5-26 세마포어 내부 코드

3-5 세마포어

예금 5만 원이 사라진 문제를 세마포어를 사용하여 해결한 코드

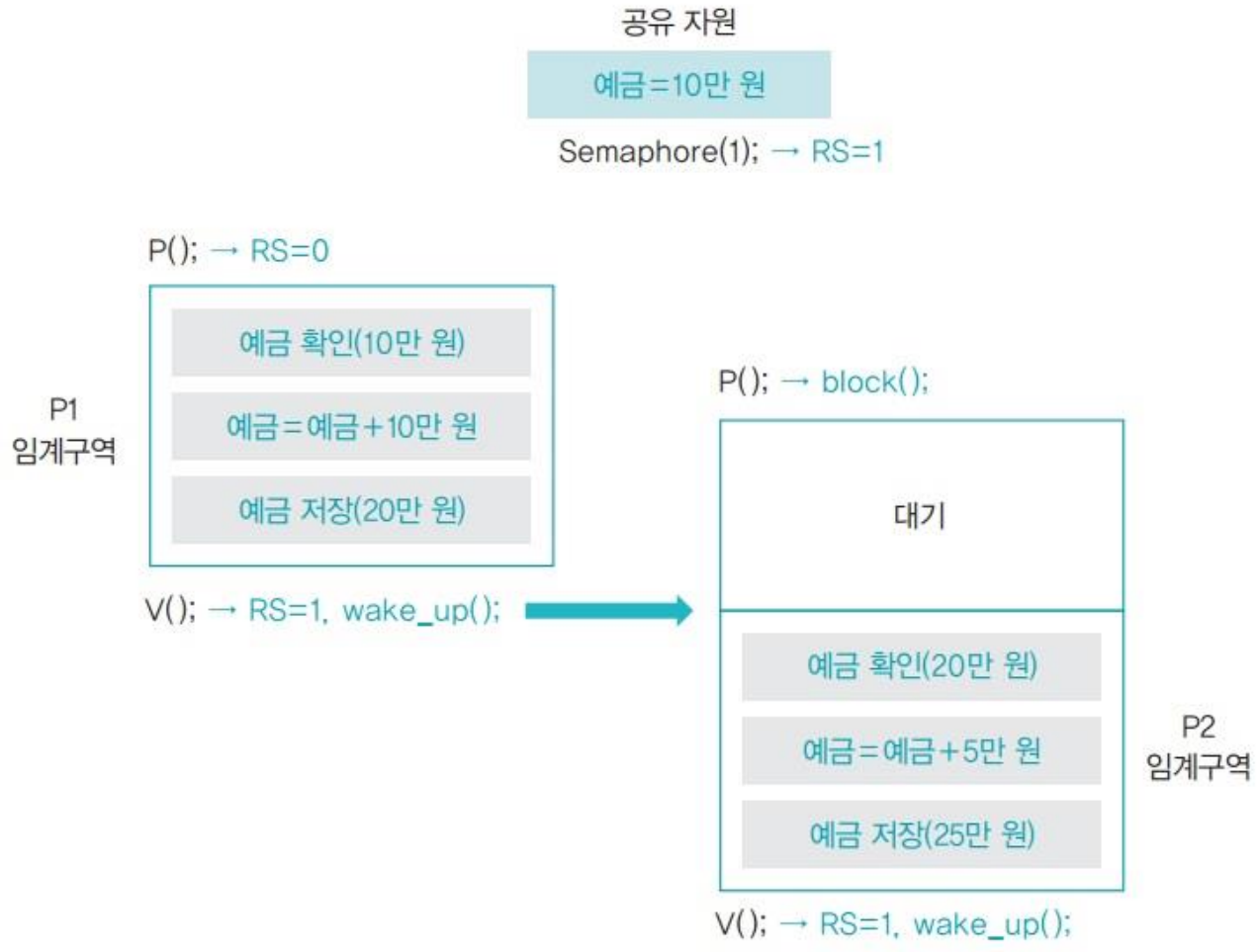

그림 5-27 세마포어 사용 예

3-5 세마포어

공유 자원이 여러 개일 때 세마포어 사용 예

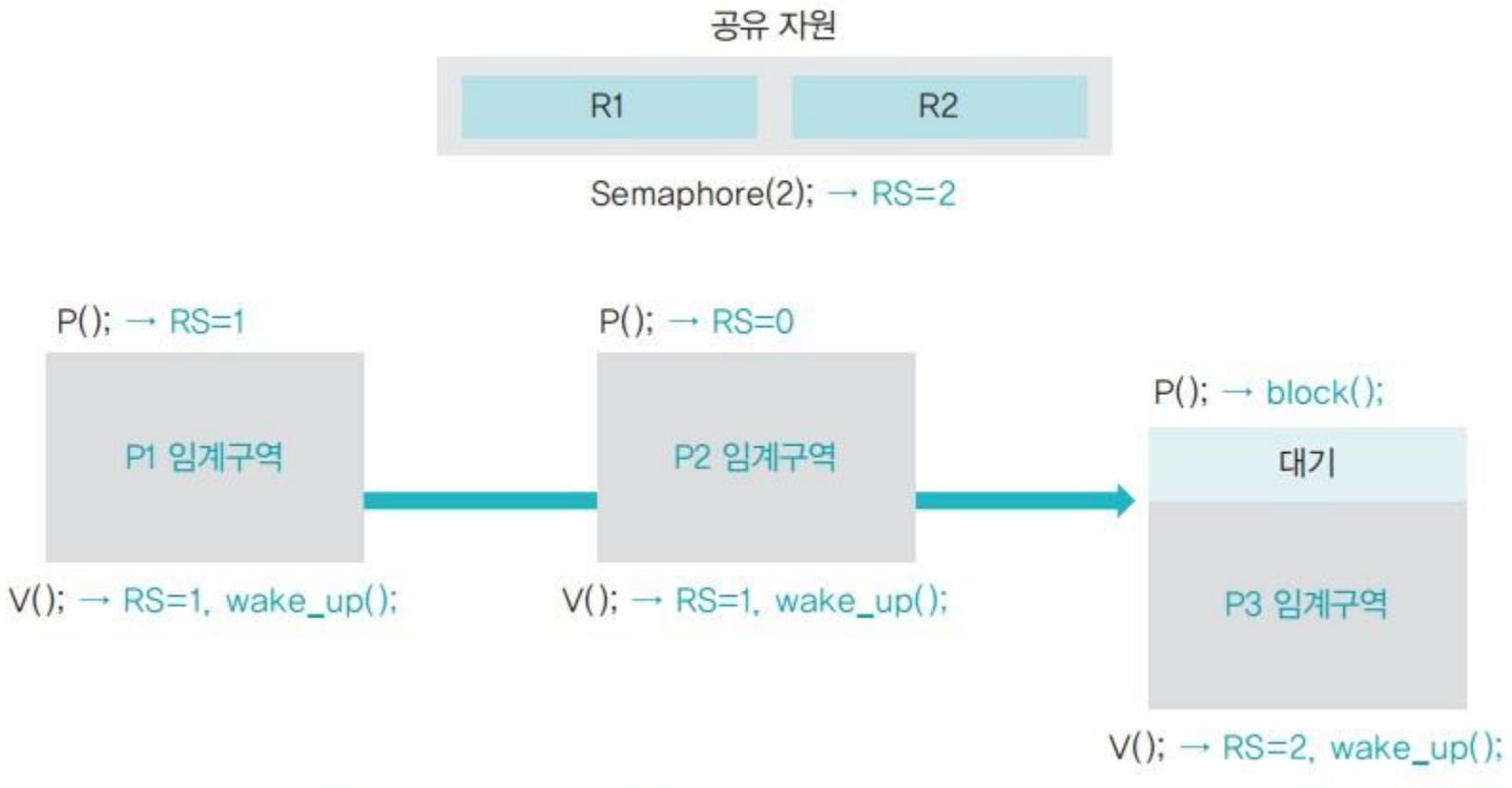

그림 5-28 공유 자원이 여러 개일 때의 세마포어 사용 예

3-6 모니터

세마포어의 잘못된 사용 예

그림 5-29 세마포어의 잘못된 사용 예

❶ 프로세스가 세마포어를 사용하지 않고 바로 임계구역에 들어간 경우로 임계구역을 보호할 수 없음

❷ P( )를 두 번 사용하여 wake_up 신호가 발생하지 않은 경우로 프로세스 간의 동기화가 이루어지지 않아 세마포어 큐에서 대기하고 있는 프로세스들이 무한 대기에 빠짐

❸ P( )와 V( )를 반대로 사용하여 상호 배제가 보장되지 않은 경우로 임계구역을 보호할 수 없음

3-6 모니터

모니터monitor

- 공유 자원을 내부적으로 숨기고 공유 자원에 접근하기 위한 인터페이스만 제공함으로써 자원을 보호하고 프로세스 간에 동기화를 시킴

모니터의 작동 원리

❶ 임계구역으로 지정된 변수나 자원에 접근하고자 하는 프로세스는 직접 P( )나 V( )를 사용하지 않고 모니터에 작업 요청

❷ 모니터는 요청받은 작업을 모니터 큐에 저장한 후 순서대로 처리하고 그 결과만 해당 프로세스에 알려줌

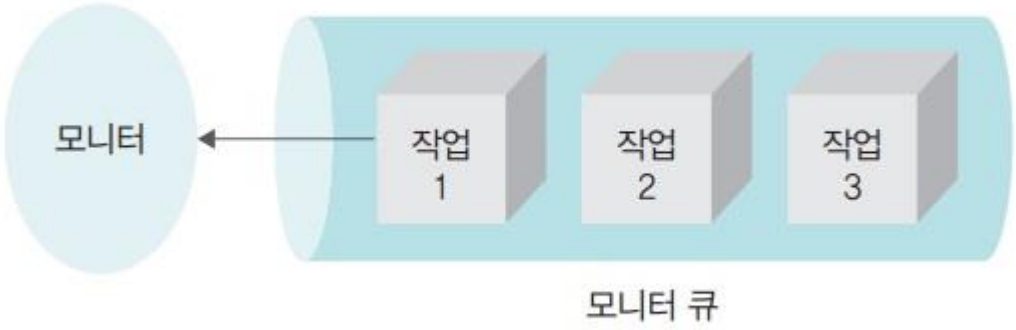

그림 5-30 모니터의 작동 원리

3-6 모니터

- **예금 5만 원이 사라진 문제를 모니터를 사용하여 해결한 코드**

그림 5-31 모니터 사용법

- **자바로 작성한 모니터 내부 코드**

```
monitor shared_balance {
   private:
      int balance=10;          /* shared data */
      boolean busy=false;
      condition mon;           /* condition variable */

   public:
      increase(int amount) {           /* public interface */
          if(busy==true) mon.wait();   /* waiting in queue */
          busy=true;
          balance=balance+amount;
          mon.signal();        /* wake up next waiting process */
      }
```

그림 5-32 자바로 작성한 모니터 내부 코드

심화학습 4-1 파일

순차 파일

- 아무리 큰 파일이라도 파일 내의 데이터는 개념적으로 한 줄로 저장됨

그림 5-33 파일의 화면 상태와 저장 상태

그림 5-34 순차적 접근의 예

4-1 파일

파일 기술자

- open( ) 함수로 파일을 열면 파일 기술자 fd를 얻음
- 파일 기술자는 파일 접근 권한 외에 현재 파일의 어느 위치를 읽고 있는지에 대한 정보도 보관
- 처음 파일이 열리면 파일 기술자는 맨 앞에 위치
- 파일에서 파일 기술자는 단 하나이고, 읽기를 하든 쓰기를 하든 파일 기술자는 계속 전진

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void main()
{   int fd;
    char buf[5];

    fd=open("com.txt", O_RDWR);
    read(fd, buf, 5);
    printf("%s", buf);
    close(fd);
    return 0;
}
```

그림 5-35 파일 입출력 코드

4-1 파일

파일 기술자

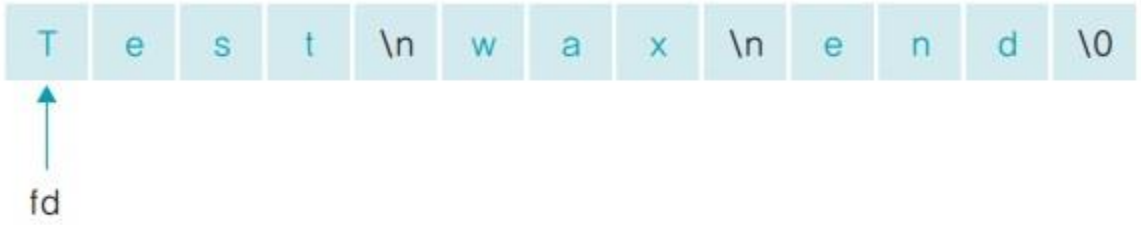

그림 5-36 파일을 처음 열었을 때 파일 기술자의 위치

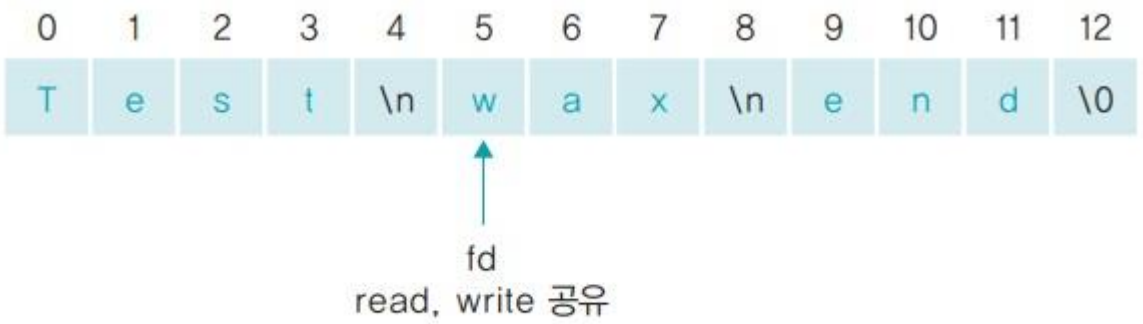

그림 5-37 read 연산을 한 후 파일 기술자의 위치

4-1 파일

파일을 이용한 통신

- read( )와 write( ) 함수가 파일 기술자를 공유하며 통신

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void main()
{    int pid, fd;
     char buf[5];

     fd=open("com.txt", O_RDWR);       /* init */
     pid=fork();

     if(pid<0 || fd<0) exit(-1);

     else if(pid==0) {                 /* child */
             write(fd, "Test", 5);
             close(fd);
             exit(0); }

     else { wait(0);                   /* parent */
            lseek(fd, 0, SEEK_SET);
            read(fd, buf, 5);
            printf("%s",buf);
            close(fd);
            exit(0); }
}
```

그림 5-38 파일을 이용한 통신 코드

4-2 파이프

파이프을 이용한 통신

- 파이프는 파일 기술자를 fd[2]와 같이 2개의 원소를 가진 배열로 정의
- 배열에서 원소 하나는 읽기용이고 하나는 쓰기용으로 사용

```
#include <stdio.h>
#include <unistd.h>

void main()
{  int pid, fd[2];
   char buf[5];

     if(pipe(fd)==-1) exit(-1); /* init */
     pid=fork();

     if(pid<0) exit(-1);

     else if(pid==0) {          /* child */
             close(fd[0]);      /* unused */
             write(fd[1], "Test", 5);
             close(fd[1]);
             exit(0); }

     else {                     /* parent */
             close(fd[1]);      /* unused */
             read(fd[0], buf, 5);
             close(fd[0]);
             printf("%s",buf);
             exit(0); }
}
```

```
#include <stdio.h>
#include <unistd.h>

void main()
{  int pid, fd[2];
   char buf[5];

     if(pipe(fd)==-1) exit(-1); /* init */
     pid=fork();

     if(pid<0) exit(-1);

     else if(pid==0) {          /* child */
             close(fd[0]);      /* unused */
             write(fd[1], "Test", 5);
             close(fd[1]);
             exit(0); }

     else {                     /* parent */
             close(fd[1]);      /* unused */
             read(fd[0], buf, 5);
             close(fd[0]);
             printf("%s",buf);
             exit(0); }
}
```

그림 5-39 파이프를 이용한 통신 코드

4-3 네트워킹

소켓을 이용한 네트워킹

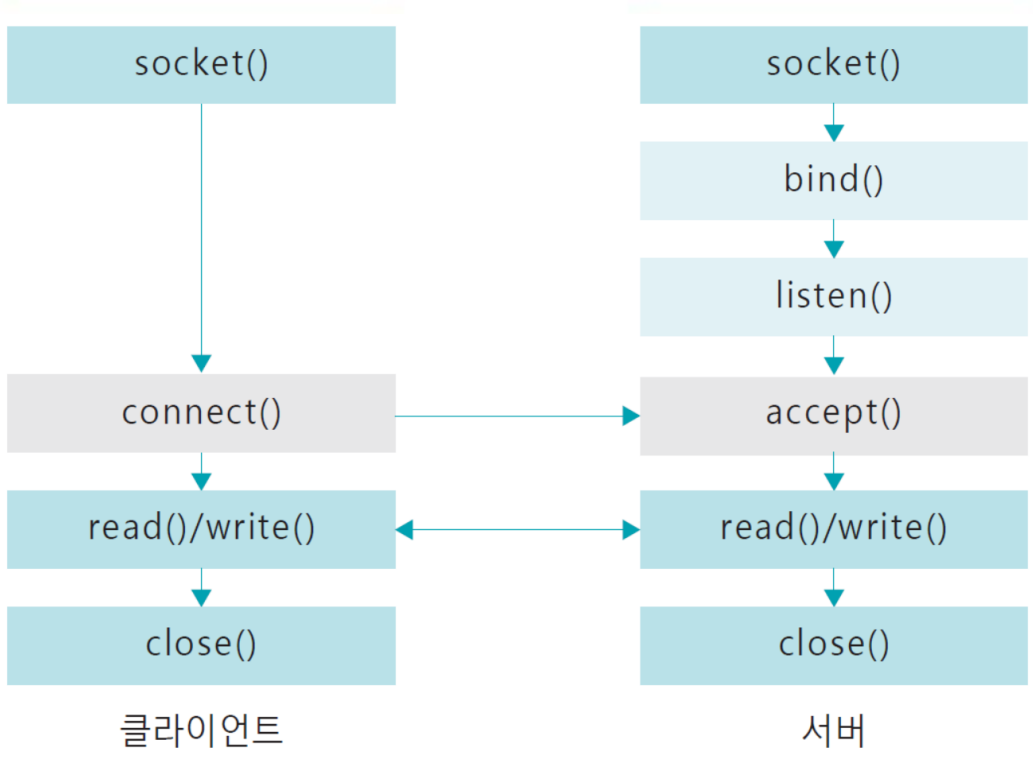

그림 5-42 클라이언트/서버의 통신 절차

4-3 네트워킹

클라이언트 쪽의 통신 절차

- 처음에 socket( ) 구문으로 소켓을 생성하고 변수 sp로 이 소켓에 접근
- 통신이 초기화되면 sp와 ad를 이용하여 서버와 connect( )를 시도
- 연결이 이루어지면 소켓으로부터 5B를 읽어 화면에 출력하고 사용한 소켓 기술자를 닫은 후 클라이언트 프로그램을 끝냄

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void main()
{   int sp;
    char buf[5];
    struct sockaddr_in ad;                          /* hold IP address */

    sp=socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    memset(&ad, 0, sizeof(ad));
    ad.sin_family=AF_INET;
    ad.sin_addr.s_addr=inet_addr("127.0.0.1");      /* loop back addr */
    ad.sin_port=htons(11234);                       /* PORT=11234 */

    connect(sp, (struct sockaddr *) &ad, sizeof(ad));

    read(sp, buf, 5);
    printf("%s",buf);

    close(sp);
    exit(0);
}
```

그림 5-43 클라이언트 코드

4-3 네트워킹

서버 쪽의 통신 절차

- 통신을 초기화한 후 소켓을 생성하고 bind( )를 이용하여 소켓을 등록
- 통신이 끝나면 소켓 기술자를 닫고 무한 루프를 돔

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void main()
{   int sp, sa;
    struct sockaddr_in ad;                  /* hold IP address */

    memset(&ad, 0, sizeof(ad));
    ad.sin_family=AF_INET;
    ad.sin_addr.s_addr=htonl(INADDR_ANY);
    ad.sin_port=htons(11234);               /* PORT=11234 */

    sp=socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    bind(sp, (struct sockaddr *) &ad, sizeof(ad));
    listen(sp, 10);
    while(1) {
        sa=accept(sp, 0, 0);
        write(sa, "Test", 5);
        close(sa);
    }
 }
```

그림 5-44 서버 코드

Thank You