

운영체제

Lecture 06: Process/thread Synchronization



전남대학교 인공지능학부
박태준 (taejune.park@jnu.ac.kr)

지난 시간 복습

- ▶ 다중 프로그래밍 → 여러 프로세스가 하나의 CPU를 공유
 - CPU의 유휴(대기) 시간이 줄이기 → CPU 활용률 향상
- ▶ 스케줄링의 요소들 → 결국은 시분할!
 - 대기시간에 대한 이야기가 많이 나왔음....!
 - "어떤 프로세스(쓰레드)에게 얼마만큼의 작업을 할당 시켜줄까?" → 타임 슬라이스
 - 기아 현상이 발생하지 않게....!
- ▶ 각종 스케줄링 알고리즘들....!
 - 음...많다!
- ▶ Context switching overhead

Goal

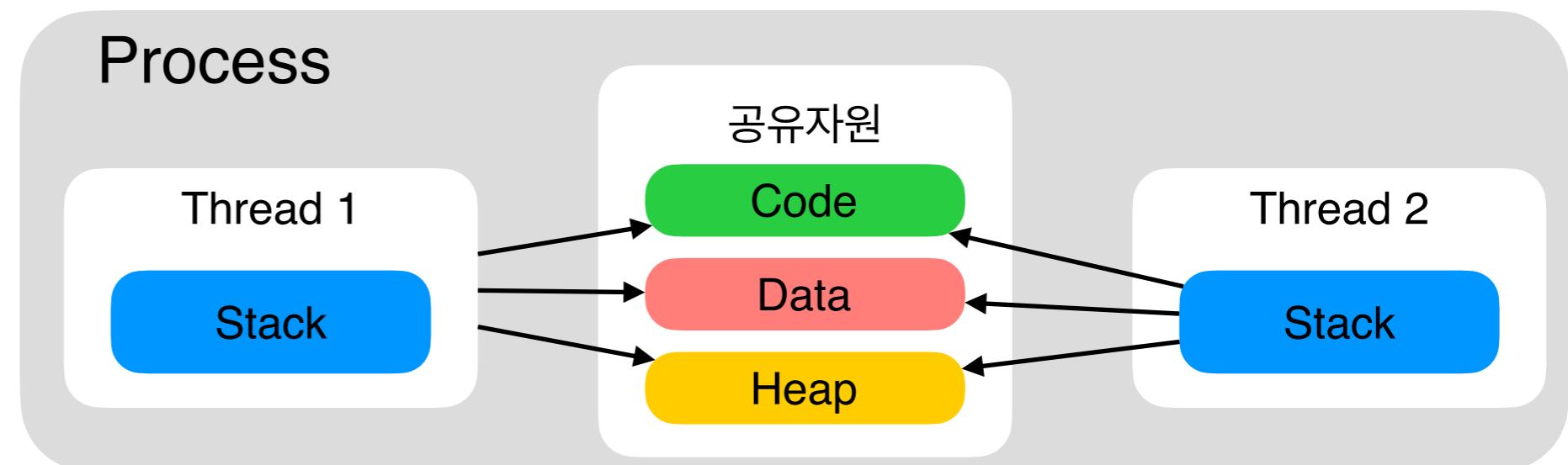
- ▶ $0 + 10000 - 10000 \neq 0?$
 - 왜 이런 현상이 발생할까? → Race condition
 - "동기화" 필요. 이것에 대해 알아보자!
- ▶ Inter-Process Communication, IPC (프로세스간 통신)
 - 종류와 개념
 - 쓰레드는 뭐... 전역 변수가 있으니...!
- ▶ 동기화가 필요한 이유에 대한 이해
- ▶ 상호배제와 임계영역 (Mutual exclusion and critical section)
 - 어떻게 구현할까?
- ▶ 멀티쓰레드 동기화
 - Mutex, spinlock, Semaphore
 - 생산자-소비자 문제

Inter-Process Communication, IPC

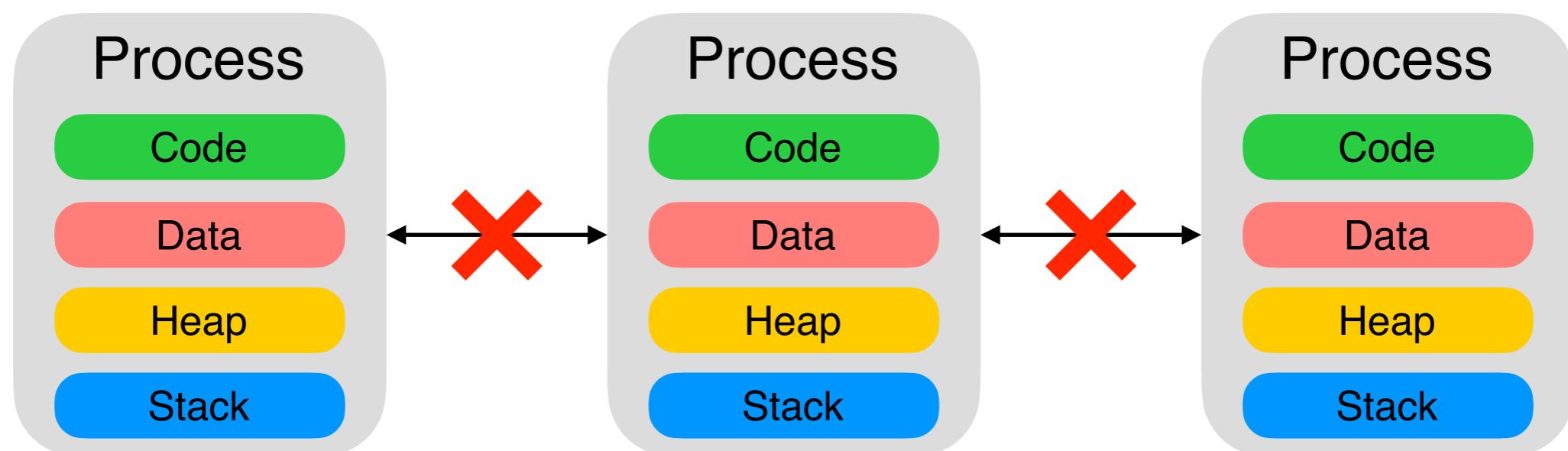
프로세스들 끼리 통신 하는 법

프로세스간 통신?

- ▶ 일단 쓰레드는...
 - 프로세스 내부 통신

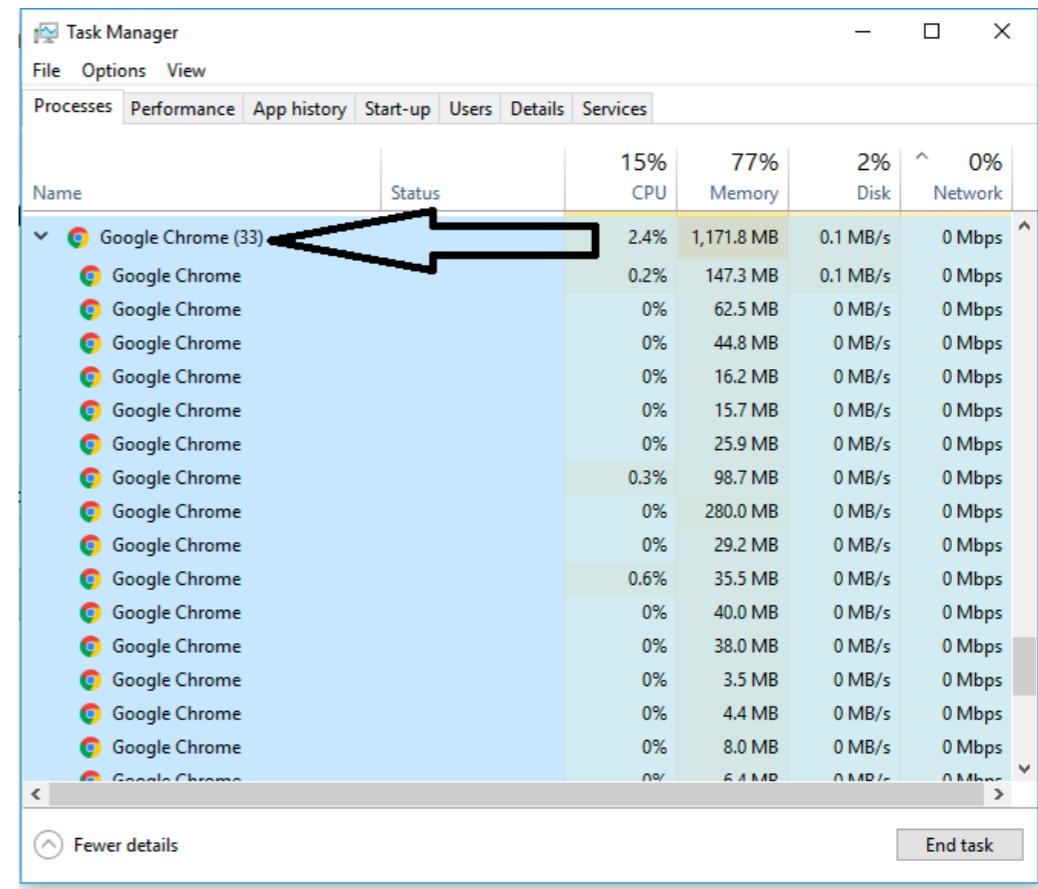


- ▶ 반면에 프로세스는.... → 가상메모리로 인하여 통신을 할 수 없다!



하지만 프로세스간의 통신은 꼭 필요...!

- ▶ 사실 컴퓨터에서 프로그램 하나만 돌려서 쓸 일이 잘 없습니다...!
 - 프로그램 하나가 여러 프로세스로 구성 되기도 하구요!
 - 또다종
- ▶ 간단한 예제들
 - Clipboard! "Copy and paste" and "drag and drop"
 - 복붙도 IPC중 하나
(e.g., Windows의 경우 WM_COPYDATA와 SendMessage() API를 이용하여 복사 데이터 전송)
 - 파일 공유
 - 사진앱 → 공유 → 카카오톡 전송
 - 카카오톡
 - Google chrome

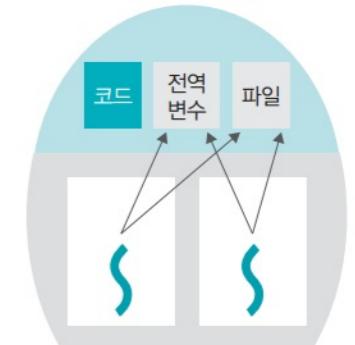


Name	Status	15% CPU	77% Memory	2% Disk	0% Network
Google Chrome (33)		2.4%	1,171.8 MB	0.1 MB/s	0 Mbps
Google Chrome		0.2%	147.3 MB	0.1 MB/s	0 Mbps
Google Chrome		0%	62.5 MB	0 MB/s	0 Mbps
Google Chrome		0%	44.8 MB	0 MB/s	0 Mbps
Google Chrome		0%	16.2 MB	0 MB/s	0 Mbps
Google Chrome		0%	15.7 MB	0 MB/s	0 Mbps
Google Chrome		0%	25.9 MB	0 MB/s	0 Mbps
Google Chrome		0.3%	98.7 MB	0 MB/s	0 Mbps
Google Chrome		0%	280.0 MB	0 MB/s	0 Mbps
Google Chrome		0%	29.2 MB	0 MB/s	0 Mbps
Google Chrome		0.6%	35.5 MB	0 MB/s	0 Mbps
Google Chrome		0%	40.0 MB	0 MB/s	0 Mbps
Google Chrome		0%	38.0 MB	0 MB/s	0 Mbps
Google Chrome		0%	3.5 MB	0 MB/s	0 Mbps
Google Chrome		0%	4.4 MB	0 MB/s	0 Mbps
Google Chrome		0%	8.0 MB	0 MB/s	0 Mbps
Google Chrome		0%	6.4 MB	0 MB/s	0 Mbps

프로세스간 통신의 종류

▶ 프로세스 내부 데이터 통신 (== 쓰레드)

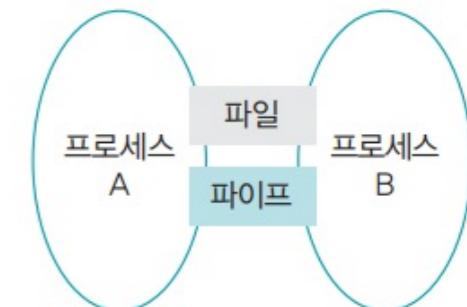
- 하나의 프로세스 내에 2개 이상의 스레드가 존재하는 경우의 통신
- 프로세스 내부의 스레드는 전역 변수나 파일을 이용하여 데이터를 주고받음



(a) 프로세스 내부 데이터 통신

▶ 프로세스 간 데이터 통신 ← 주로 이걸 말함

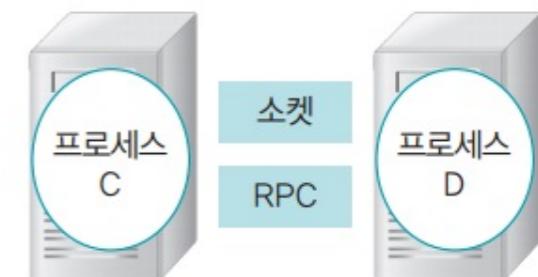
- 같은 컴퓨터에 있는 여러 프로세스끼리 통신하는 경우
- 공용 파일 또는 운영체제가 제공하는 파이프를 사용하여 통신



(b) 프로세스 간 데이터 통신

▶ 네트워크를 이용한 데이터 통신

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



(c) 네트워크를 이용한 데이터 통신

아주아주 원시적인 IPC 방법

- ▶ 파일을 경유해서 메시지를 전달한다!

```
#include <stdio.h>

int main() {
    FILE *f = fopen("temp.txt", "w");

    fprintf(f, "Can you hear me? \n");
    fclose(f);
    return 0;
}
```

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

int main() {
    sleep(10);
    FILE *f = fopen("temp.txt", "r");
    char buf[128];

    fread(buf, 1, 128, f);
    printf(buf)
    fclose(f);
    return 0;
}
```

Process 1.
Sender

File
temp.txt

Process 2.
Receiver

프로세스간 통신 분류

▶ 전송 방식에 따른 분류

• 단방향 통신 (Simplex)

- 라디오 신호처럼 한쪽 방향으로만 데이터를 전송할 수 있는 구조
- 프로세스 간통신에서는 전역 변수와 파일이 단방향 통신에 해당

• 반양방향 통신 (반이중 통신, Half duplex)

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

• 양방향 통신 (전이중 통신, Full duplex)

- 데이터를 동시에 양쪽 방향으로 전송할 수 있는 구조.
- 프로세스 간 통신에서는 소켓 통신이 양방향 통신에 해당

▶ 수신 대기에 따른 분류

• 대기가 있는 통신 (~ 인터럽트): 동기화를 지원하는 통신 방식

- 데이터를 받는 쪽은 데이터가 도착할 때까지 자동으로 대기 상태에 머물러 있음

• 대기가 없는 통신 (~ 풀링): 동기화를 지원하지 않는 통신 방식

- 데이터를 받는 쪽은 바쁜 대기를 사용하여 데이터가 도착했는지 여부를 직접 확인

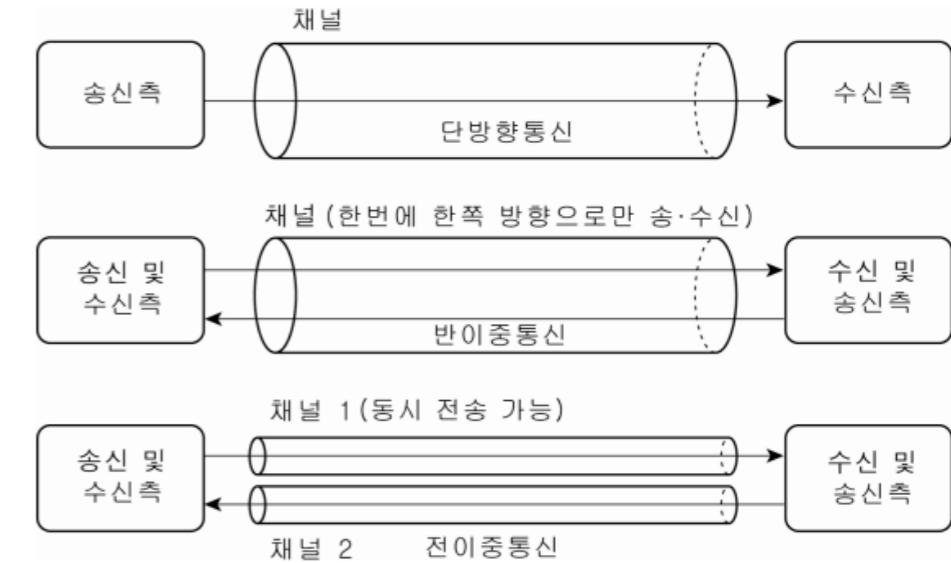


표 5-1 프로세스 간 통신의 분류

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

프로세스 간 통신 방법 (1)

- ▶ 파일을 이용한 통신
 - 데이터를 쓰고 읽는 함수를 통해서 이루어짐!
 - e.g., open, write, read

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
    int fd;
    char buf[124]={0,};
    fd = open ("temp.txt", O_RDWR);
    write(fd, "Test", 5);
    lseek(fd, 0, SEEK_SET);
    read(fd, buf, 5);
    printf(buf);
    close(fd);
    return 0;
}
```

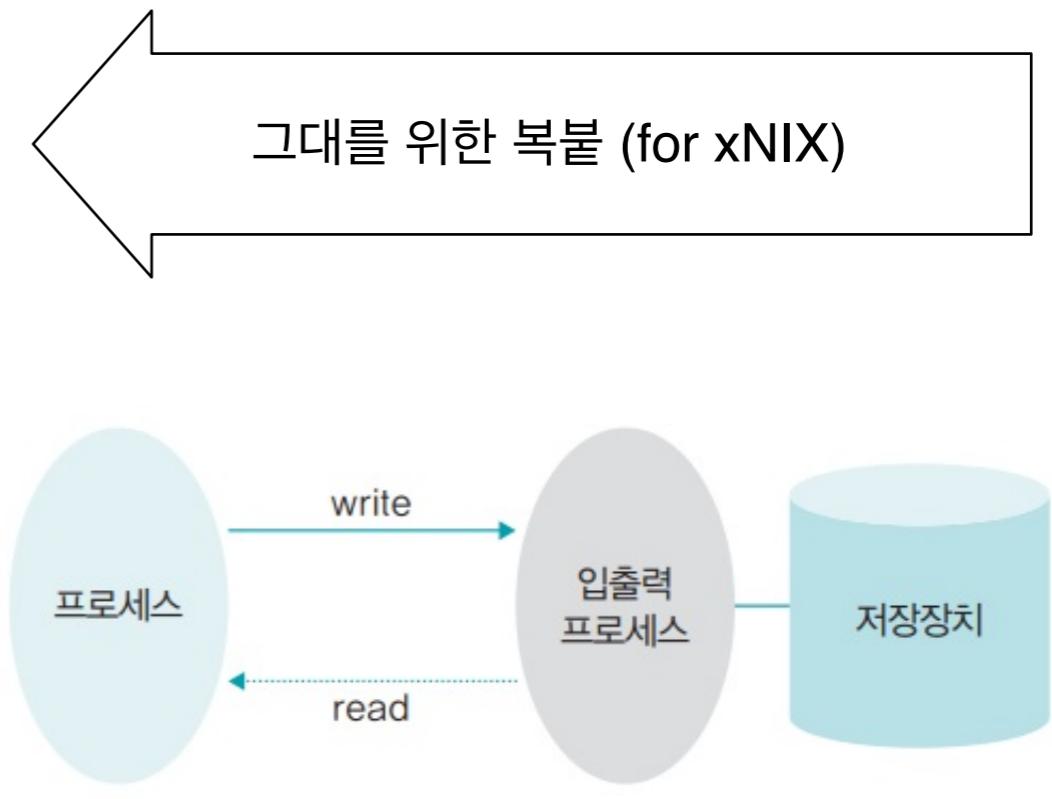


그림 5-7 파일 입출력 연산

프로세스 간 통신 방법 (2)

- ▶ Fork! 파일 디스크립터를 공유해서 사용 (부모-자식간 통신)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
    char str[128] = {0,};
    int fd = open("temp.txt", O_RDWR);
    pid_t pid;
    pid = fork();
    if(pid > 0){ // parent
        write(fd, "Can you hear me?", 17);
        sleep(2);
        lseek(fd, 17, SEEK_SET);
        read(fd, str, 12);
        printf("parent(%X): %s \n", getpid(), str);
        close(fd);
    }
    else if(pid == 0) { // child
        sleep(1);
        lseek(fd, 0, SEEK_SET);
        read(fd, str, 17);
        printf("child(%X): %s \n", getpid(), str);
        write(fd, "Yes, I can!", 12);
    }
    return 0;
}
```

그대를 위한 복불 (for xNIX)

프로세스 간 통신 방법 (3)

- ▶ 파이프를 이용한 통신
 - Linux에서 "\$ ls | grep passed" ← 이런거도 파이프
 - 운영체제가 제공하는 동기화 통신 방식으로, 파일 입출력과 같이 open() 함수로 Pipe descriptor를 열고 작업을 한 후 close() 함수로 마무리
 - 파이프에 쓰기 연산을 하면 데이터가 전송되고 읽기 연산을 하면 데이터를 받음 → 단방향!

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(){
    int pipefd[2]; // [0] for read, [1] for write
    pid_t pid;
    char buf[128];
    if(pipe(pipefd) < 0){ // int pipe(int pipefd[2]);
        printf("pipe error\n");
        return 1;
    }
    pid = fork();
    if(pid > 0){ //parent process
        close(pipefd[0]);
        strcpy(buf, "Hello?\n");
        write(pipefd[1], buf, strlen(buf));
    }
    else if(pid == 0) { //child process
        close(pipefd[1]);
        read(pipefd[0], buf, 128);
        printf(">> I am child! I can hear the parent!!:
%s\n",buf);
    }
    else {
        printf("fork error\n");
    }
    return 0;
}
```

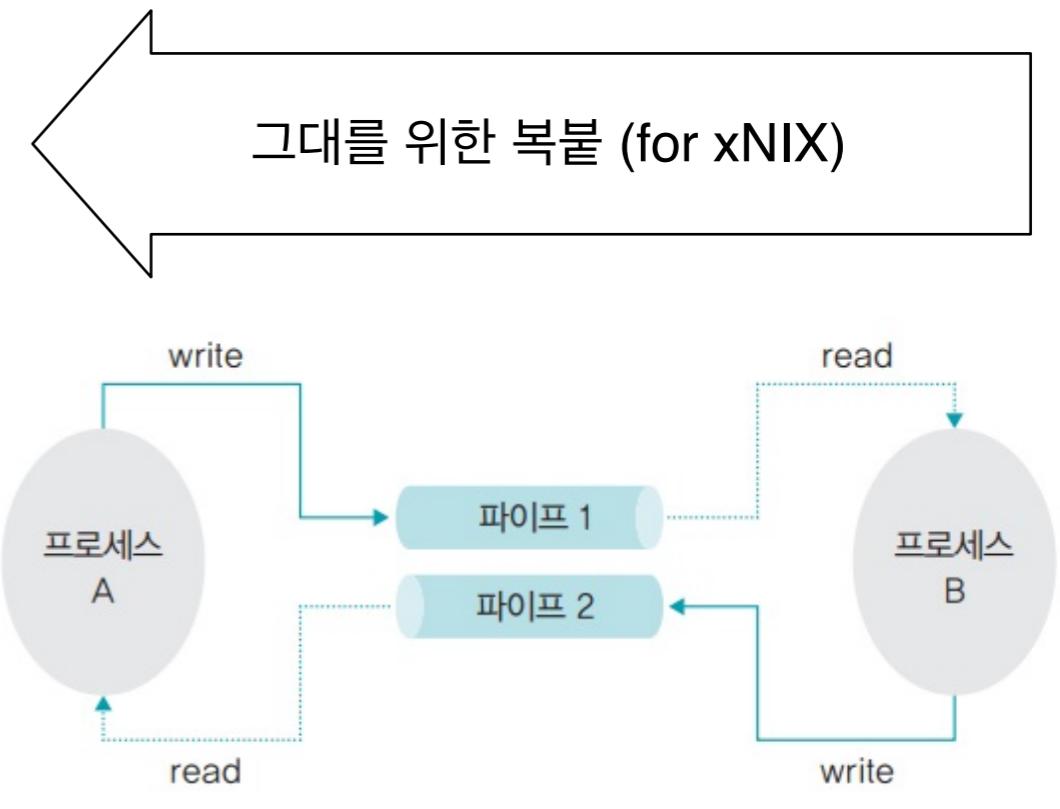


그림 5-8 파이프를 이용한 통신

파이프는 두 종류가 있어요!

- ▶ 이름 없는 파이프 (걍 PIPE)
 - 일반적으로 파이프라고 하면 이름 없는 파이프를 가리킴
- ▶ 이름 있는 파이프(명명 파이프, **Named Pipe, FIFO**)
 - FIFO라 불리는 특수 파일을 이용하여 서로 관련 없는 프로세스 간 통신에 사용
- ▶ 차이점
 - pipe는 열려져 있는 pipe에 대해서는 open할 수 없지만, FIFO는 가능!
 - 즉, 서로 다른 (부모자식이 아닌) process간에 data를 pipe는 주고받지 못하지만 FIFO는 가능
 - pipe는 파일시스템에 이미지를 생성하지 않는 반면에 FIFO는 파일시스템상에 이미지를 가짐

Named pipe (FIFO) 예제

receive

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main( void )
{
    int counter = 0;
    int fd;
    char buff[128];
    if ( mkfifo("temp.txt", 0666) < 0 ) {
        perror( "mkfifo() error" );
        return 1;
    }
    fd = open("temp.txt", O_RDWR);
    // <- why RW? It's for both read and write access.
    while(1) {
        memset( buff, 0, 128 );
        read(fd, buff, 128);
        printf( "%d: %s\n", counter++, buff );
    }
    close(fd);
}
```

send

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
int main( void )
{
    int fd;
    char *str = "I am the sender!";
    fd = open("temp.txt", O_WRONLY);
    write(fd, str, strlen(str));
    close(fd);
}
```

```
taejune@Taejunes-MBP2021 codes % ls -al temp.txt
prw-r--r-- 1 taejune staff 0 4 5 16:10 temp.txt
```

프로세스 간 통신 방법 (4)

▶ 소켓을 이용한 통신

- 일종의 네트워크 프로그래밍 ~ Server and client
- 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

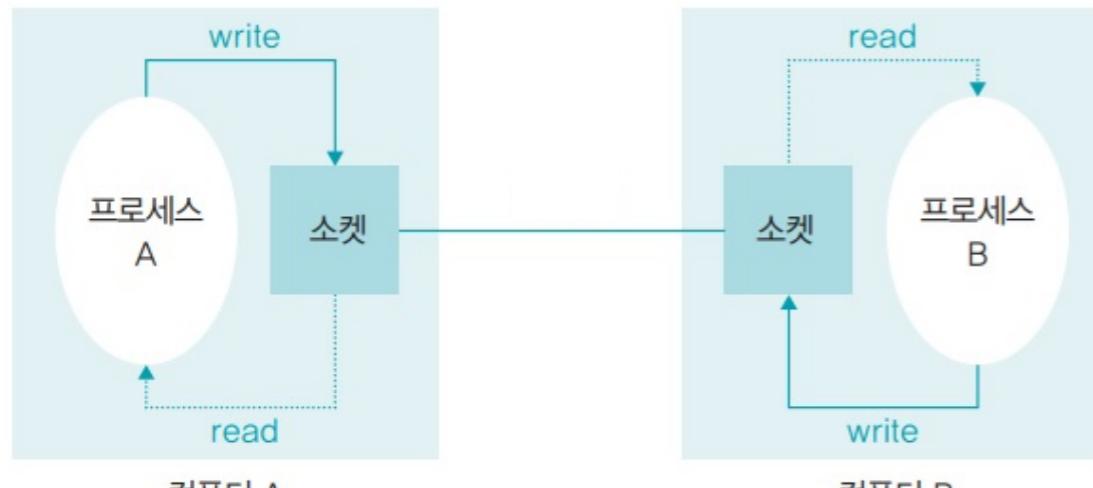
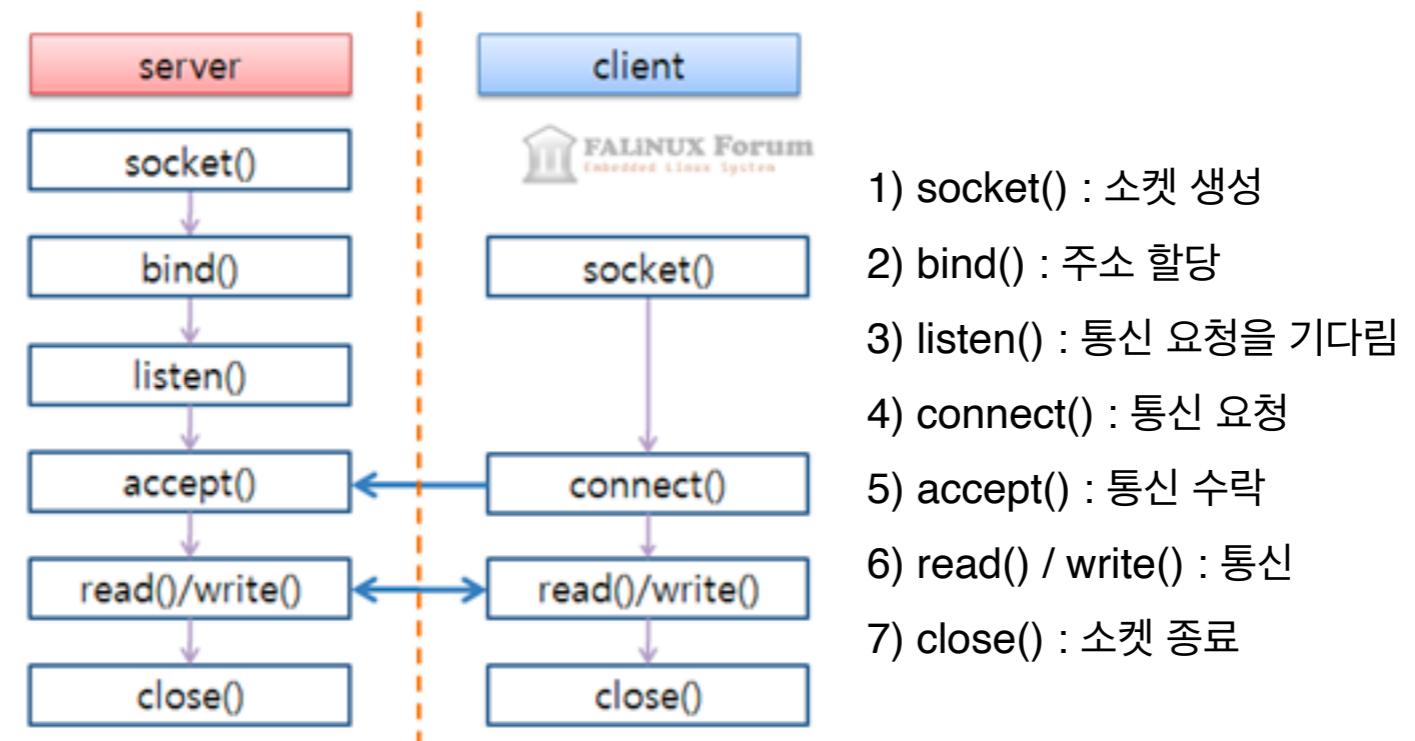


그림 5-9 소켓을 이용한 통신



Socket 예제

Server

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
//#include "sock.h"

int main() {
    char buffer[256];
    char buffer2[256];
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) return -1;

    struct sockaddr_in saddr;
    memset(&saddr, 0, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(7777); // port number

    // open a server
    if (bind(fd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
        return -1;
    if (listen(fd, 16) < 0) return -1;

    while (1) {
        struct sockaddr_in caddr;
        int len = sizeof(caddr);
        int client_fd = accept(fd, (struct sockaddr*) &caddr, &len);
        if (client_fd < 0) continue;

        // read from client
        memset(buffer, 0, sizeof(buffer));
        read(client_fd, buffer, sizeof(buffer));
        printf(buffer);

        sprintf(buffer2, "I got you!: %s", buffer);
        write(client_fd, buffer2, strlen(buffer2));

        close(client_fd);
    }
    return 0;
}
```

Client

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>

int main() {
    char buffer[256];
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) return -1;

    struct hostent* hptr = gethostbyname("127.0.0.1");
    if (hptr->h_addrtype != AF_INET) return -1;

    struct sockaddr_in saddr;
    memset(&saddr, 0, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr =
        ((struct in_addr*) hptr->h_addr_list[0])->s_addr;
    saddr.sin_port = htons(7777); // port number

    // connect to the server
    if (connect(sockfd, (struct sockaddr*) &saddr, sizeof(saddr)) < 0)
        return -1;

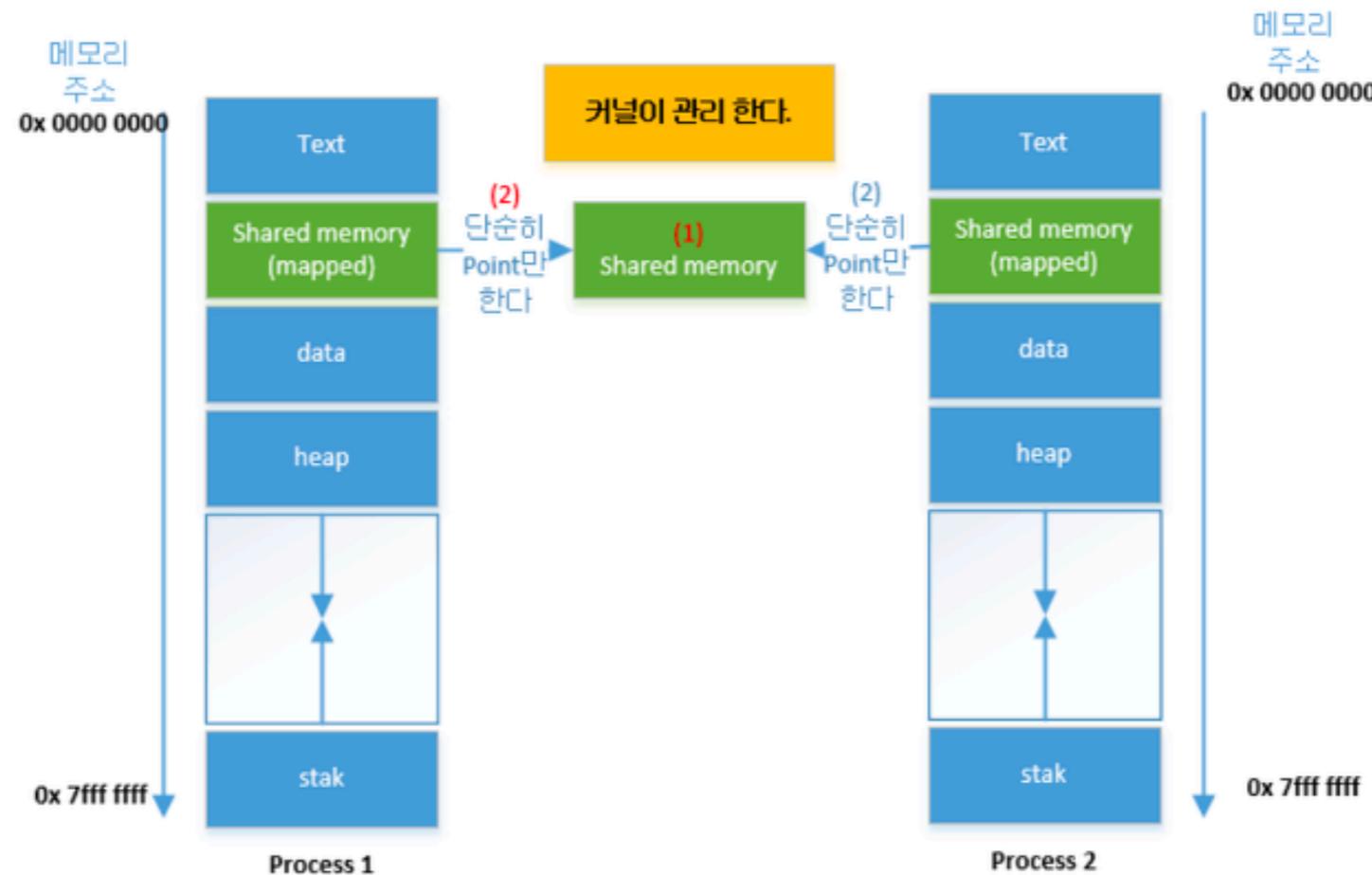
    char *msg = "This is a socket communication!!";
    write(sockfd, msg, strlen(msg));
    memset(buffer, 0, sizeof(buffer));
    read(sockfd, buffer, sizeof(buffer));
    printf(buffer);
    close(sockfd);

    return 0;
}
```

프로세스 간 통신 방법 (5)

▶ Shared memory

- 커널이 관리되는 일정한 크기의 공유 메모리 공간을 통해 통신 (전역변수와 비슷)
- 사용하려는 프로세스간 할당한 크기가 동일해야 사용 가능
- 동기화가 중요한 포인트!



shared memory 예제

Write

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main()
{
    int shmid = 0;
    char *shared_memory = 0;

    // get shared memory
    shmid = shmget((key_t)7777, 128, 0666|IPC_CREAT);
    if (shmid == -1) return -1;

    // attach to the shared memory
    shared_memory = (char *)shmat(shmid, (void *)0, 0);
    if (shared_memory == -1) return -1;

    while(1)
    {
        strcpy(shared_memory, "Hello?\n");
        sleep(1);
        strcpy(shared_memory, "Can you hear me?\n");
        sleep(1);
    }
}
```

```
taejune@Taejunes-MBP2021 codes % ipcs -m
IPC status from <running system> as of Tue Apr  5 18:42:06 KST 2022
T      ID      KEY          MODE        OWNER      GROUP
Shared Memory:
m 458752 0x00001e61 --rw-rw-rw- taejune    staff
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main()
{
    int shmid;
    char *shared_memory = 0;

    // get shared memory
    shmid = shmget((key_t)7777, 128, 0666|IPC_CREAT);
    if (shmid == -1) return -1;

    // attach to the shared memory
    shared_memory = (char *)shmat(shmid, (void *)0, 0666|IPC_CREAT);
    if (shared_memory == -1) return -1;

    while(1){
        printf(shared_memory);
        sleep(1);
    }
}
```

clear

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main()
{
    int shmid;
    char *buff = NULL;
    void *shared_memory = (void *)0;

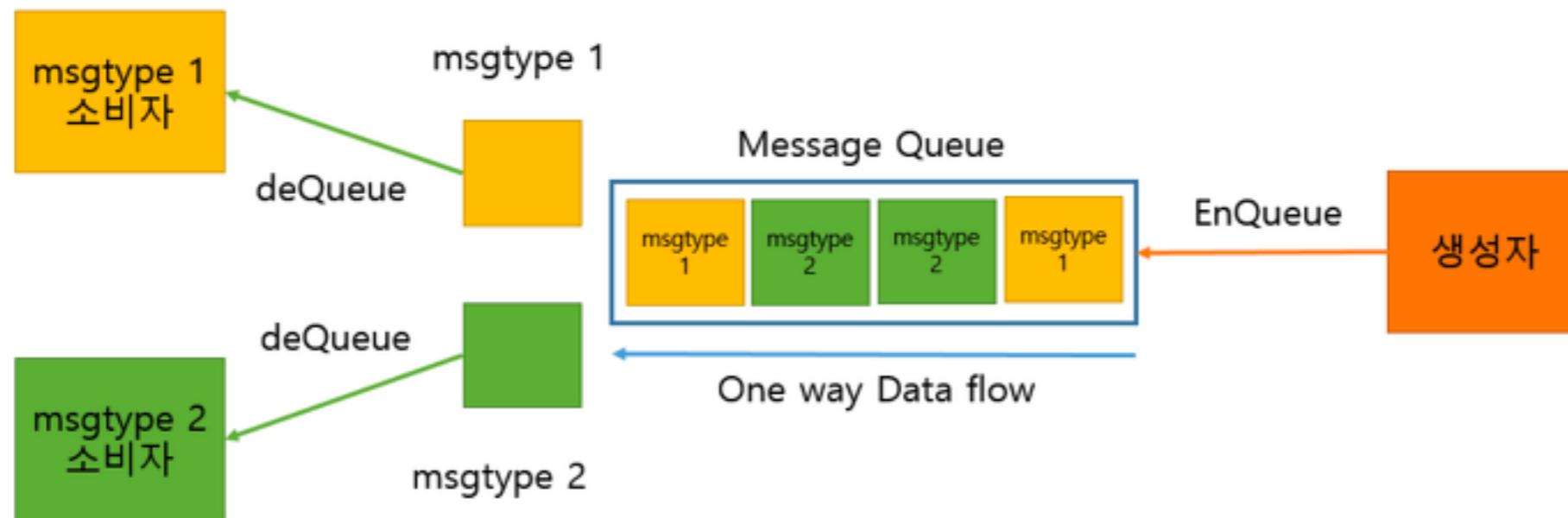
    // get shared memory
    shmid = shmget((key_t)7777, 128, 0666|IPC_CREAT);
    if (shmid == -1) return -1;

    printf("%X \n", shmctl(shmid, IPC_RMID, 0));
}
```

프로세스 간 통신 방법 (6)

▶ Message queue

- 일종의 파이프, 커널 메모리 공간을 활용함
- 구조체를 기반으로 작동
- 동기화를 고려할 필요 없음.



Message queue 예제

Sender

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>

struct msgbuf {
    long mtype;          /* message type, must be > 0 */
    char mtext[128];     /* message data */
};

int main()
{
    key_t key_id;

    struct msgbuf msg = {5, {0,}}; // type, text

    // Message Queue
    key_id = msgget((key_t)7777, IPC_CREAT|0666);
    if (key_id == -1) return -1;

    while (1) {
        strcpy(msg.mtext, "This is MQ!\n");
        msgsnd(key_id, &msg, sizeof(struct msgbuf), IPC_NOWAIT);
        sleep(1);

        strcpy(msg.mtext, "Wow~~!!\n");
        msgsnd(key_id, &msg, sizeof(struct msgbuf), IPC_NOWAIT);
        sleep(1);
    }
    return 0;
}
```

Receiver

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>

struct msgbuf {
    long mtype;          /* message type, must be > 0 */
    char mtext[128];     /* message data */
};

int main()
{
    key_t key_id;
    struct msgbuf msg;

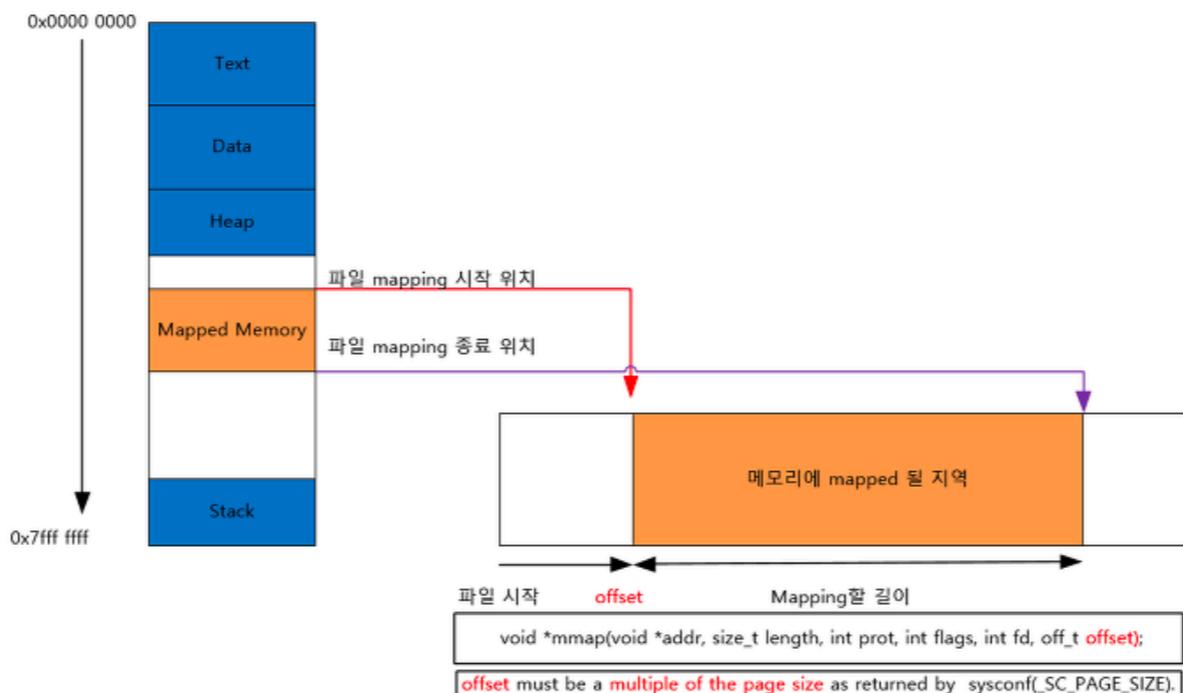
    key_id = msgget(7777, IPC_CREAT|0666);
    if (key_id < 0) return -1;

    while(1) {
        msgrcv(key_id, &msg, sizeof(struct msgbuf), 5, 0);
        printf("%s",msg.mtext);
    }
    return 0;
}
```

프로세스 간 통신 방법 (7)

▶ Memory map

- 파일을 프로세스의 메모리에 일정 부분 맵핑 키션 사용
 - Shared Memory 공간과 마찬가지로 메모리를 공유한다는 측면에 있어서는 서로 비슷
 - 차이점은 Memory Map의 경우 열린파일을 메모리에 맵핑시켜서 공유
- 파일로 대용량 데이터를 공유 할 때 유리
- 메모리에 맵핑된 파일을 읽거나 쓰면 read() 나 write() 시스템 콜을 사용할 때 발생하는 불필요한 복사를 방지(추가적인 복사는 사용자 영역의 버퍼로 데이터를 읽고 써야 하기 때문에 발생)



Message map 예제

Sender

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
    int fd;
    char *file = NULL;
    struct stat sb;
    int flag = PROT_WRITE | PROT_READ;

    fd = open("temp.txt", O_RDWR|O_CREAT);

    if (fstat(fd, &sb) < 0) return -1;
    file = (char *) mmap(0, 40, PROT_WRITE | PROT_READ, MAP_SHARED, fd, 0);

    while(1)
    {
        strcpy(file, "This is the memory map!\n");
        sleep(1);
        strcpy(file, "wow!!\n");
        sleep(1);
    }

    munmap(file, 40);
    close(fd);

    return 0;
}
```

Receiver

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    char *file = NULL;
    struct stat sb;
    int flag = PROT_WRITE | PROT_READ;

    fd = open("temp.txt", O_RDWR|O_CREAT);

    if (fstat(fd, &sb) < 0) return -1;
    file = (char *) mmap(0, 40, PROT_WRITE | PROT_READ, MAP_SHARED, fd, 0);

    while (1)
        printf("%s", file);
        sleep(1);
    }

    munmap(file, 40);
    close(fd);

    return 0;
}
```

정리!

IPC 종류	사용 시기	공유 매개체	통신 단위	통신 방향	통신 가능 범위
PIPE	부모 자식 간 단 방향 통신 시	파일	Stream	단 방향	동일 시스템
Named PIPE	다른 프로세스와 단 방향 통신 시	파일	Stream	단 방향	동일 시스템
Message Queue	다른 프로세스와 단 방향 통신 시	메모리	구조체	단 방향	동일 시스템
Shared Memory	다른 프로세스와 양 방향 통신 시	메모리	구조체	양 방향	동일 시스템
Memory Map	다른 프로세스와 양 방향 통신 시	파일+메모리	페이지	양 방향	동일 시스템
Socket	다른 시스템간 양 방향 통신 시	소켓	Stream (Packet)	양 방향	동일 + 외부 시스템

구분

- ▶ "Advanced Programming in the UNIX Environment"
- ▶ 어떤 IPC를 쓸 것이냐는, 각 IPC의 속도보다, 하고자하는 일의 종류에 따라 달라짐
 - 1. IPC를 쓰는 프로세스가 parent-child 관계이냐 여부.
 - 2. IPC proc의 server/client에서 client가 하나냐 다수냐의 여부.
 - 3. IPC로 주고받을 데이터의 크기
 - 4. IPC 처리 과정이 file descriptor 처리와 비슷하냐의 여부. 즉, select() / poll()을 쓰면 쉽냐 여부.
 - 5. IPC 프로그램이 쉽게 파일을 지정해서 쓸 수 있느냐 여부.
 - 아무 생각 없이 제일 쓰기 편한 것: 소켓.

공유자원의 개념

일단 전제 하나! 프로세스/쓰레드를 섞어가며 쓸텐데, 동일한 문제입니다!

공유 자원 (공유 데이터)

- ▶ 하나를 두고 여럿이 쓰는 것
 - 다중 프로그래밍에서는 CPU도 공유자원!
- ▶ 이번에 말하는 공유자원은 여러 프로세스/쓰레드가 공동으로 이용하는 변수, 메모리, 파일 등...!
 - 공동으로 이용되기 때문에 누가 언제 데이터를 읽거나 쓰느냐에 따라 그 결과가 달라질 수 있음

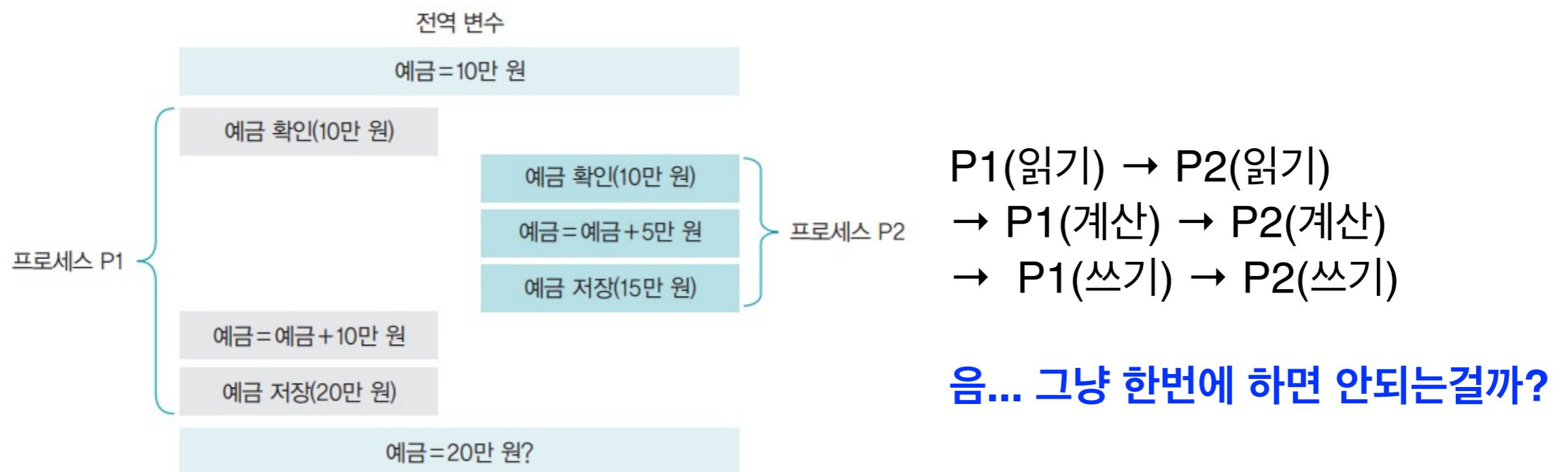
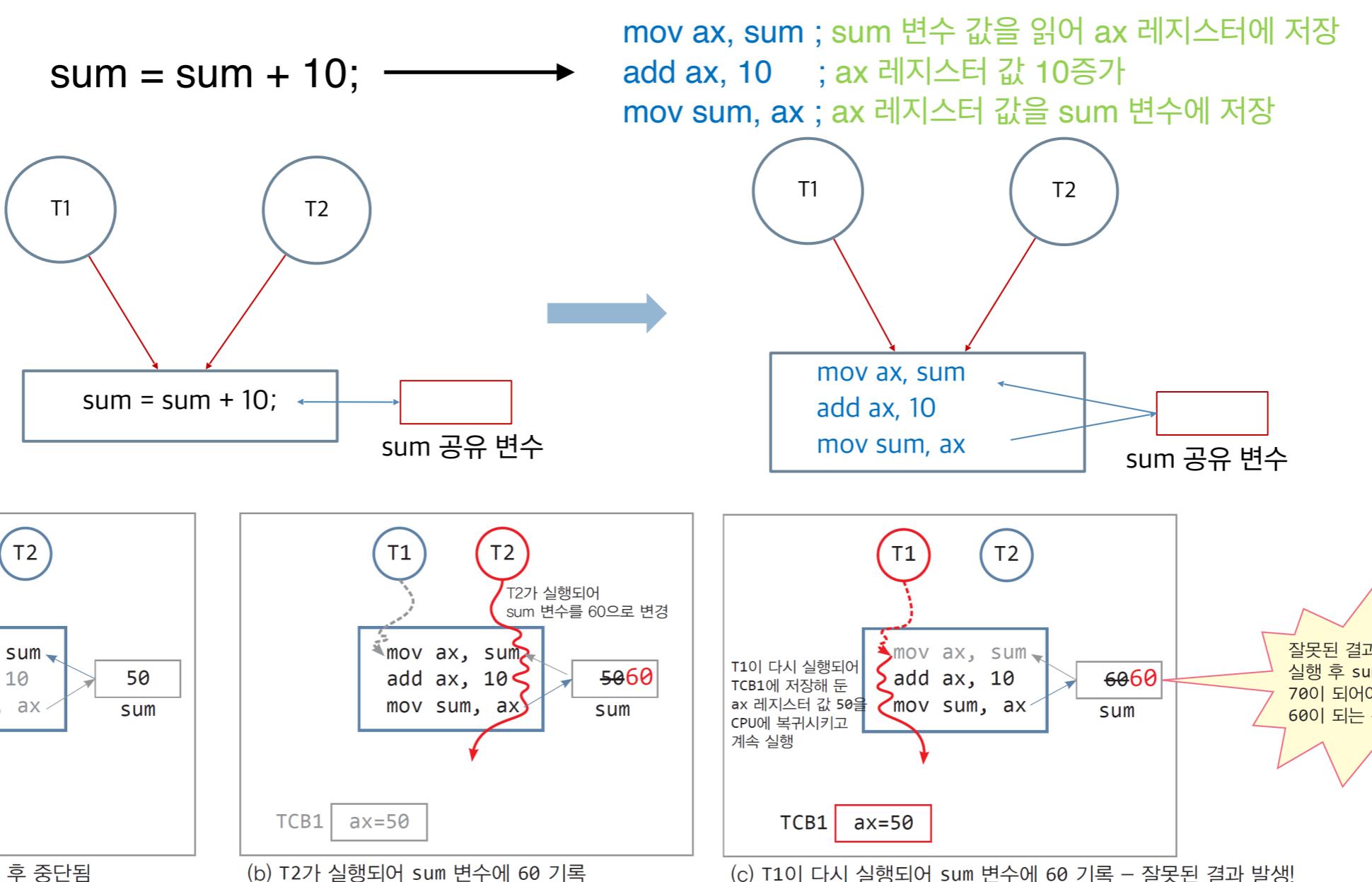


그림 5-10 공유 자원의 접근

발생하는 원인을 자세히 살펴보면...

- 명령어를 처리중에 스케줄링이 걸리면 (e.g., timeout) 부득이하게 명령어가 분리되요!
 - 두 쓰레드가 공유자원 'sum'에다가 10을 더한다고 했을 때...



Race condition

- ▶ 경쟁 상태 (조건)
 - 2개 이상의 프로세스가 공유 자원을 병행적으로 읽거나 쓰는 상황
 - 경쟁 조건이 발생하면 공유 자원 접근 순서에 따라 실행 결과가 달라질 수 있음
 - 데이터가 오염됨 (Corrupted)
 - 밀티스레드 환경에서 매우매우 자주 발생! (커널에서도!)
 - ▶ 일상생활에서도 흔히 발견 가능...'ㅅ'
 - git
 - 여러분의 조별과제

```
import java.util.Map;
import static com.intellij.vcs.log.data.index.VcsLogMessagesTrigramIndex.TRIGRAMS;
public class VcsLogMessagesTrigramIndex extends VcsLogMessagesIndex {
    private static final String TRIGRAMS = "trigram";
    public VcsLogMessagesTrigramIndex(@NotNull String logId, @NotNull DisposableParent disposableParent) {
        super(logId, TRIGRAMS, getVersion(), new TrigramProcessor(disposableParent));
    }
    @NotNull
    public static Collection<File> getStorageFiles() {
        return Collections.singletonList(getStorageFile());
    }
}
@Nullable
public ValueContainer.IntIterator getCommitsFor(@NotNull TrigramProcessor trigramProcessor, @NotNull Map<String, TrigramProcessor> commitProcessors) {
    return null;
}
```

조별과제 원본 파일



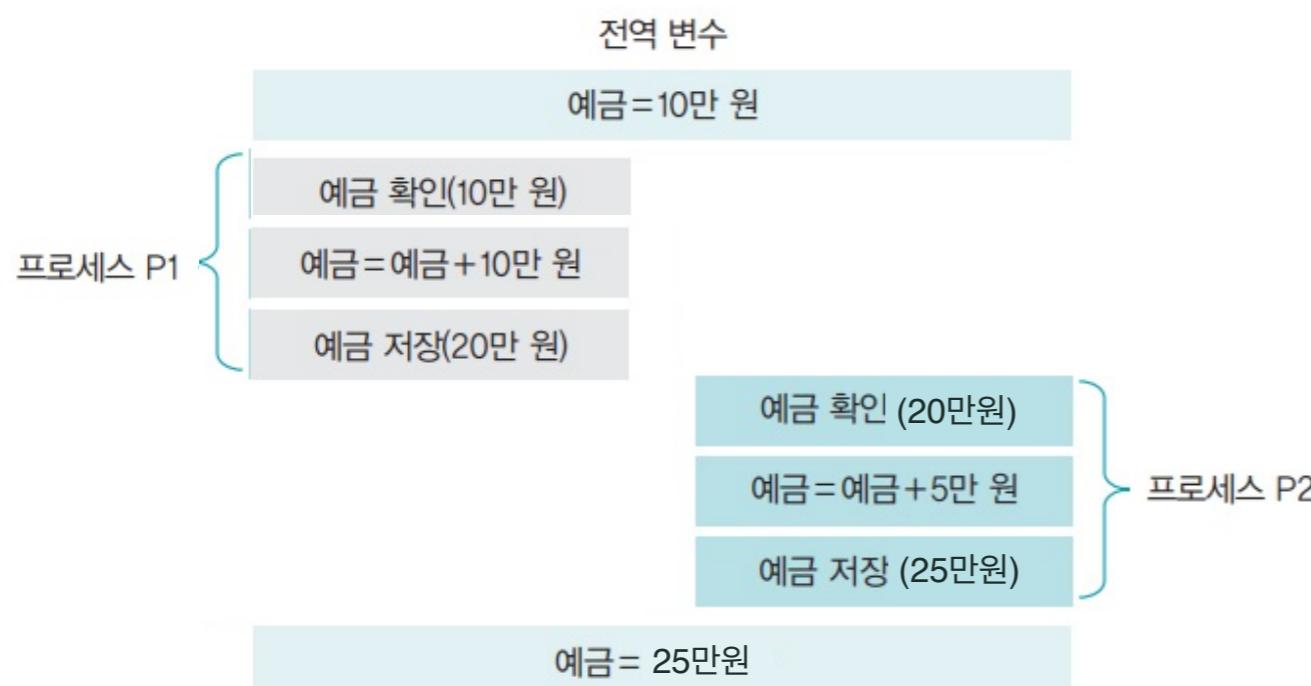
친구A: 수정

친구B: 수정

합치기...???

Synchronization (동기화)

- ▶ 공유자원에 대해 다수가 동시에 접근할 때 공유데이터가 훼손되는 문제의 해결책
- ▶ 공유 데이터를 접근하고자 하는 다수의 프로세스 또는 쓰레드가 충돌 없이 공유 데이터에 접근하기 위해 상호 협력하는 것
 - 한 프로세스/쓰레드가 공유데이터를 **배타적 독점적**으로 접근
== 동시에 접근하는 것이 아니라 차례대로, 순서대로 접근 == 스케줄링!
 - e.g., 조별과제: "어 잠깐, 나부터 먼저 수정할께!"



P1(읽기) → [다른 프로세스의 접근을 막음]
→ P1(계산) → P1(쓰기) → [접근 허용]
→ P2(읽기) → P2(계산) → P2(쓰기)

이런 작업을 '상호배제' 라고 부릅니다!

- ▶ 그리고 상호배제가 필요한/적용된 구간을 임계영역이라고 부릅니다.
- ▶ **Critical section**
 - 공유 자원 접근 순서에 따라 실행 결과가 달라지는 프로그램의 영역
 - == 공유 자원에 접근하는 프로그램 코드 영역
 - == Shared data가 있는 영역
 - Shared data == Critical data
- ▶ **Mutual exclusion**
 - 임계구역이 오직 한 프로세스만 배타적 독점적으로 사용되도록 하는 기술
 - 임계구역에 먼저 진입한 스레드가 임계구역의 실행을 끝낼 때까지 다른 프로세스가 진입하지 못하도록 보장
 - 어떤 프로세스가 임계구역에 들어가면 다른 프로세스는 임계구역 밖에서 기다려야 하며 임계구역의 프로세스가 나와야 들어갈 수 있음

요론게 잘 되면 thread-safe!

- ▶ 안되면 thread-safe하지 않음

The screenshot shows a Microsoft Learn Documentation page for Windows App Development. The left sidebar has a tree view of topics under Windows Web Services, including Windows Web Services Reference and Handles. The 'WS_MESSAGE' topic is highlighted in a grey box. The main content area shows the title 'WS_MESSAGE', a brief description, and a C++ code snippet: `typedef struct _WS_MESSAGE WS_MESSAGE;`. Below this is a 'Remarks' section with a note about thread safety, and a 'Requirements' section with a table.

Microsoft | Learn Documentation Training Certifications Q&A Code Samples Assessments Shows Events

Windows App Development Explore Development Platforms Resources

Filter by title

... / Apps / Win32 / Desktop Technologies / Networking and Internet / Windows

WS_MESSAGE

Article • 12/12/2020 • 2 minutes to read • 3 contributors

The opaque type used to reference a [message](#) object.

C++

```
typedef struct _WS_MESSAGE WS_MESSAGE;
```

Remarks

This object is not thread safe. For more information, see [thread safety](#).

Requirements

Requirement	Value
Minimum supported client	Windows 7 [desktop apps UWP apps]

Mutual exclusion

문을 걸어 잠그는 방법.

개념

- ▶ 원칙(목표): 임계영역에는 오직 1개의 프로세스/스레드만 진입
 - 입구에 '게이트'를 두고, 이를 잠글 수 있는 열쇠 '**Lock**' 을 만들자
 - **Lock**이 on이 되어있으면? → 못들어간다! 열릴때 까지 입구에서 대기
- ▶ Mutual Exclusion primitives
 - enterCS()
 - 임계 영역 진입 전 검사하는 것 - 똑똑!! 누가 안에 있는지 검사하는 것
 - 다른 프로세스가 임계 영역안에 있는지 검사
 - exitCS()
 - 임계 영역을 벗어날 때 후처리 - 기다리는애 들어와! 라고 알리는 것
 - 다른 프로세스가 임계 영역을 떠났음을 알림.



상호 배제를 포함 하는 프로그램

▶ 1. 일반 코드(non-critical code)

- 공유 데이터를 액세스하지 않는 코드

▶ 2. 임계구역 진입 코드(entry code)

- 상호배제를 위해 필요한 코드 (enterCS())
 - 임계구역에 진입하기 전 필요한 코드 블록
- 현재 임계구역을 실행 중인 스레드가 있는지 검사
 - 없다면, 다른 스레드가 들어오지 못하도록 조치
 - 있다면, 진입이 가능해질 때까지 대기

▶ 3. 임계구역 코드(critical code)

▶ 4. 임계구역 진출 코드(exit code)

- 상호배제를 위해 필요한 코드 (exitCS())
 - 임계구역의 실행을 마칠 때 실행되어야 하는 코드 블록
- entry code에서 대기중인 스레드가 임계구역에 진입할 수 있도록 entry code에서 취한 조치를 해제하는 코드



그런데 문을 아무렇게나 만들면 안됩니다!

- ▶ 다음과 같은 조건을 만족시켜야 합니다...! 그런데 이게 영 쉬운게 아니더라구요...!

- ▶ **상호 배제 (mutual exclusion)**

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

- ▶ **한정 대기 (bounded waiting)**

- 어떤 프로세스도 무한 대기하지 않아야 함

- ▶ **진행의 융통성 (progress flexibility)**

- 한 프로세스가 다른 프로세스의 진행을 방해해서는 안 된다는 것

문을 만드는 방법 두가지

- ▶ 소프트웨어적 방법
 - 두개의 프로세스에 대해...: 데커(Dekker) 알고리즘, 피터슨(Peterson) 알고리즘 등
 - 여러 프로세스에 대해...: Lamport의 뺑집 알고리즘 등
- ▶ 하드웨어적 방법 → 오늘날 대부분 하드웨어 기반 동작
 - 임계 구역 진입/진출 코드에 구현
 - 방법 1) 인터럽트 서비스 금지
 - 방법 2) 원자 명령(atomic instruction) 사용

상호배제구현 방법 1 – 인터럽트 서비스 금지

- 임계구역 entry 코드에서 인터럽트 서비스를 금지하는 명령 실행
 - 장치로부터 인터럽트가 발생해도 CPU가 인터럽트를 무시함 / 인터럽트 루틴을 실행하지 않음
 - 인터럽트를 금지한다 → Scheduling에 의한 context switching도 자연스럽게 금지됨.
 - 쓰레드도 중단되지 않음!

문제점

- 모든 인터럽트가 무시되는 문제
 - 시스템의 효율적인 운영을 방해하기 쉬움
- 멀티코어 CPU 또는 다중 CPU에서는 활용이 불가능
 - 모든 CPU의 인터럽트가 멈춤
 - 진행의 융통성 조건 불충족**

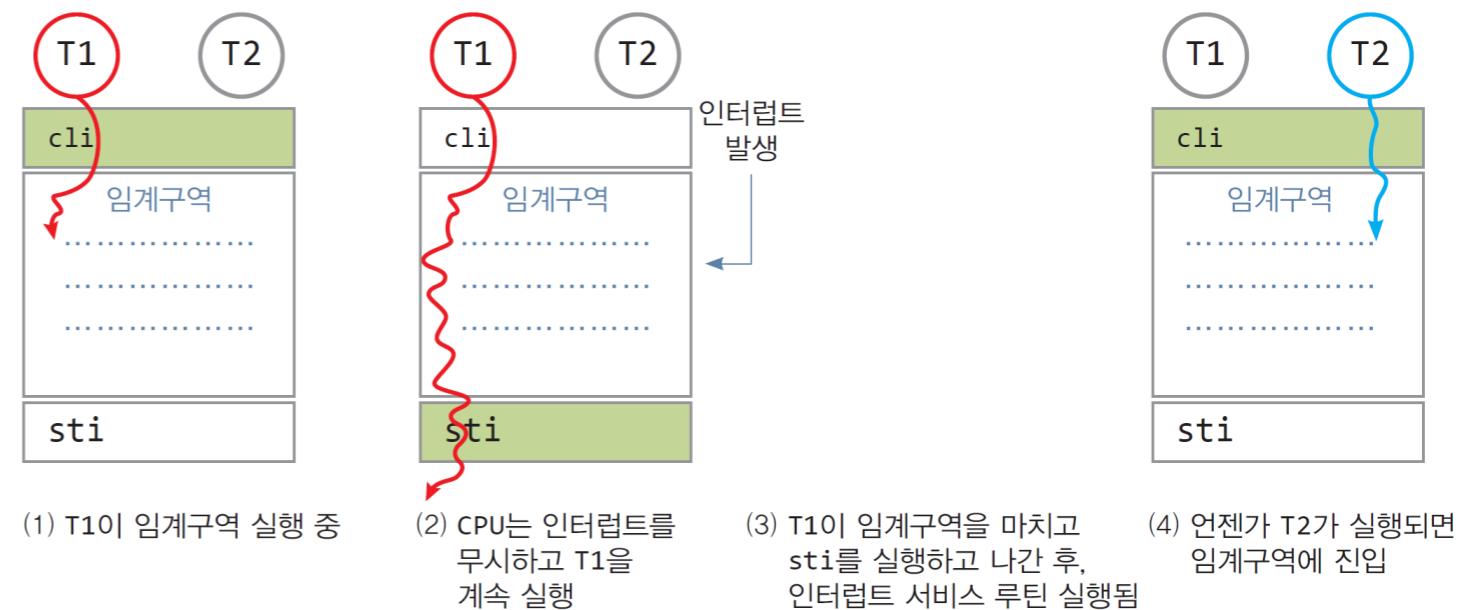
해결책:

- Lock을 현재 엑세스 하고 있는 메모리에만 국한시켜야함

`cli ; entry 코드. 인터럽트 서비스 금지 명령 (clear interrupt flag)`

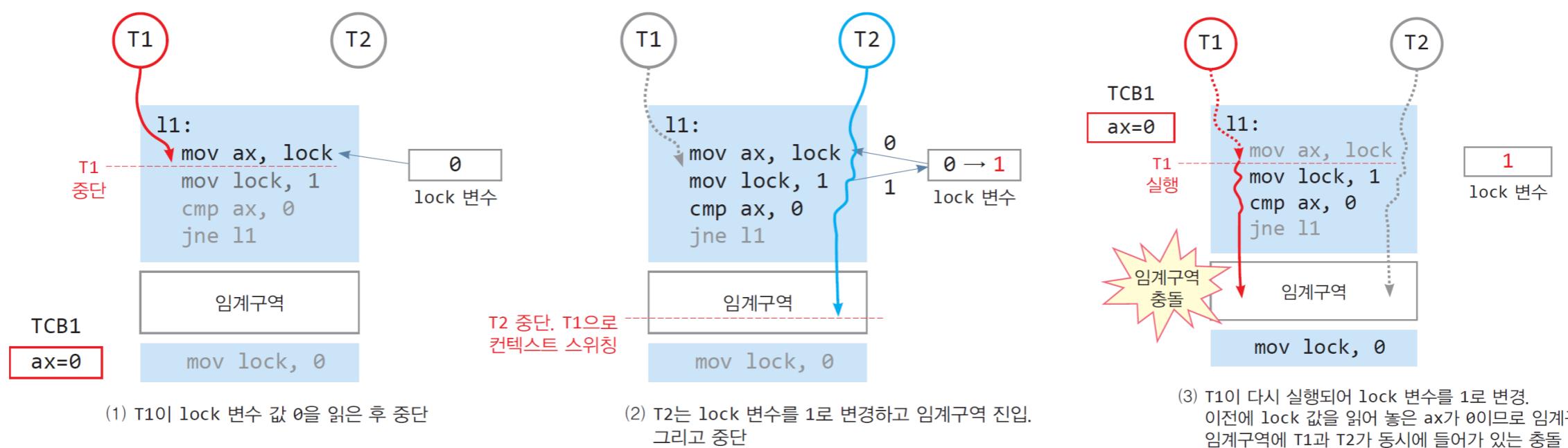
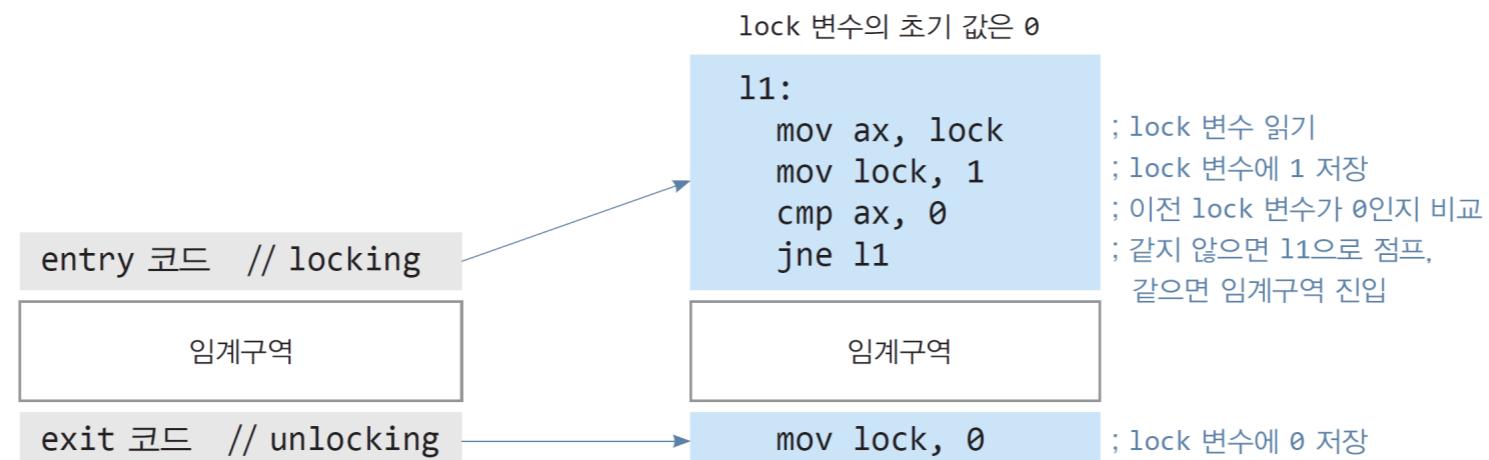
임계구역 코드

`sti ; exit 코드. 인터럽트 허용 명령(set interrupt flag)`



그렇다고 Lock이 단순하게 on/off가 되어선 안되요!

- 문이 열려있는지 닫혀있는지 확인하기 위한 Lock도 공유 데이터 → 동기화 이슈!
 - 예, lock 변수가 1이면 잠금 상태, 0이면 열린 상태 일 때...



직접 보자.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int count = 200000;

int sum = 0; // global variable, shared data
bool lock = false; // lock

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        while(lock == true); // critical section
        lock = true;
        sum += 1;
        lock = false;
    }
    return 0;
}

void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        while(lock == true); // critical section
        lock = true;
        sum -= 1;
        lock = false;
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

    printf("sum = %d\n", sum);
    return 0;
}
```

그대를 위한 복불 (for xNIX)

```
----- Run "sync_lock01.c" -
Start!
sum = -73943
```

상호배제 조건 불충족

이러한 문제를 해결하기 위해선?

- ▶ 서로의 상태까지 확인이 가능하도록 만들어줌

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int count = 200000;

int sum = 0; // global variable, shared data
bool lock1 = false; // lock1
bool lock2 = false; // lock2

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        lock1=true;
        while(lock2 == true); // critical section
        sum += 1;
        lock1 = false;
    }
    return 0;
}

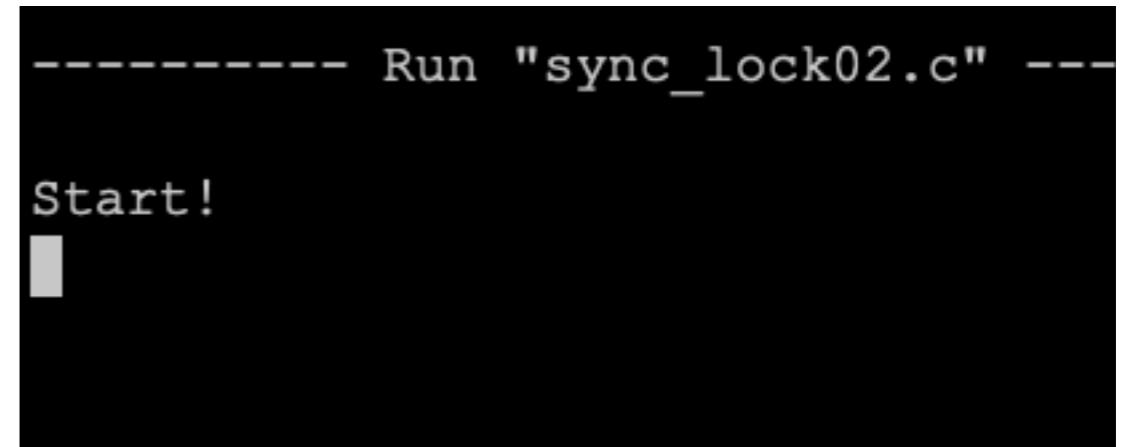
void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        lock2=true;
        while(lock1 == true); // critical section
        sum -= 1;
        lock2 = false;
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

    printf("sum = %d\n", sum);
    return 0;
}
```



그런데, 무한루프에 빠지기 쉬워요!
lock1이 true가 된 상태에서 컨텍스트 스위칭이 발생
→ lock2가 true가 되고 다시 컨텍스트 스위칭이 발생
→ 둘다 while구문에서 대기하여야 함
→ 무한 루프...!!

한정대기 조건 불충족

제대로 해결하기 위해서?

- ▶ 서로 사용권 자체를 주고 받자.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int count = 200000;

int sum = 0; // global variable, shared data
int lock = 1; // lock

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        while(lock == 2); // critical section
        sum += 1;
        lock = 2;
    }
    return 0;
}

void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        while(lock == 1); // critical section
        sum -= 1;
        lock = 1;
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

    printf("sum = %d\n", sum);
    return 0;
}
```

```
----- Run "sync_lock03.c" ----

Start!
sum = 0
```

```
taejune@Taejunes-MBP2021 codes % time ./a.out
lock01.c
Start!
sum = -150628
./a.out 0.00s user 0.00s system 103% cpu 0.006 total
taejune@Taejunes-MBP2021 codes % time ./a.out
lock03.c
Start!
sum = 0
./a.out 0.07s user 0.00s system 181% cpu 0.038 total
taejune@Taejunes-MBP2021 codes %
```

진행의 융통성 조건 불충족

상호배제구현 방법 2 - 원자 명령(atomic instruction)

- lock 변수를 이용한 상호배제의 실패 원인?
 - 실패 원인은 entry 코드에 있음
 - lock 변수 값을 읽는 명령과 lock 변수에 1을 저장하는 2개의 명령 사이에 컨텍스트 스위칭이 될 때 문제 발생

```
11:  
    mov ax, lock  
    mov lock, 1  
    cmp ax, 0  
    jne 11
```

컨텍스트 스위칭

lock 값을 읽고

lock에 1 기록

해결책: 원자 명령(atomic instruction) 도입

- lock을 읽어 들이는 명령과 lock 변수에 1을 저장하는 2개의 명령을 한번에 처리하는 명령어 필요
- 원자 명령 : TSL(Test and Set Lock)

```
mov ax, lock  
mov lock, 1
```

→ TSL ax, lock

원자명령

entry 코드

```
11:  
    mov ax, lock  
    mov lock, 1  
    cmp ax, 0  
    jne 11
```

임계구역

exit 코드

```
    mov lock, 0
```

(a) 상호배제 실패

원자명령 사용

```
11:  
    TSL ax, lock  
    cmp ax, 0  
    jne 11
```

임계구역

```
    mov lock, 0
```

(b) 원자명령 TSL를 사용하여 상호배제 성공

Test and Set

- ▶ 하드웨어의 지원을 받은 Lock 구현!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdatomic.h>

const int count = 200000;

int sum = 0; // global variable, shared data
//bool lock = false; // lock
atomic_flag lock = ATOMIC_FLAG_INIT;

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        while( atomic_flag_test_and_set(&lock) );
        sum += 1;
        atomic_flag_clear(&lock);
    }
    return 0;
}

void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        while( atomic_flag_test_and_set(&lock) );
        sum -= 1;
        atomic_flag_clear(&lock);
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

    printf("sum = %d\n", sum);
    return 0;
}
```

```
----- Run "sync_lock04.c" ----

Start!
sum = 0
-----
```

```
taejune@Taejunes-MBP2021 codes % time ./a.out
Start!
sum = 0
./a.out 0.04s user 0.00s system 168% cpu 0.023 total
```

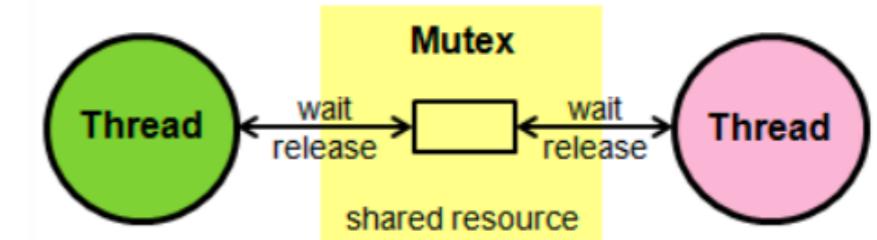
Process / Thread synchronization 기법

문을 어떻게 관리하나 방법들

동기화 3대장

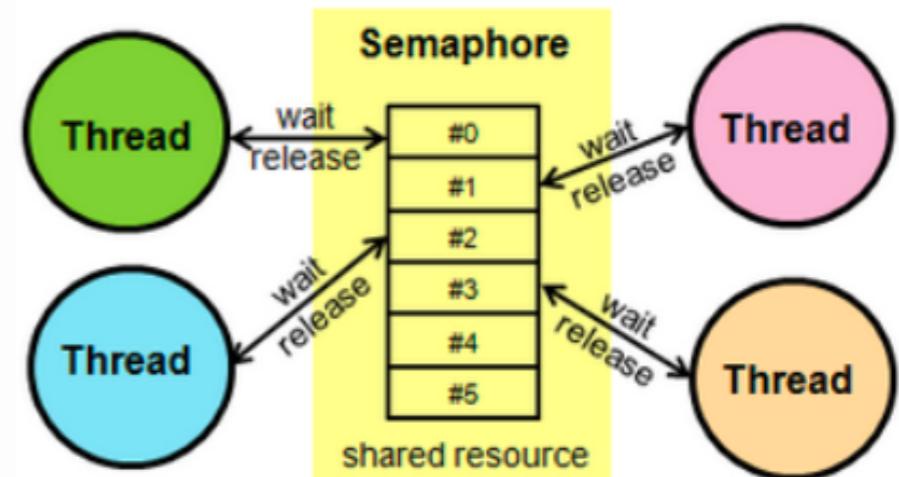
▶ locks 방식 : 뮤텍스(mutex), 스핀락(spinlock)

- 상호배제가 되도록 만들어진 락(lock) 활용
- Mutex는 동기화 대상이 only 1개일 때 사용
- 락을 소유한 스레드만이 임계구역 진입
- 락을 소유하지 않은 스레드는 락이 풀릴 때까지 대기
- 둘의 차이는 busy-wait, sleep-wait (큐 존재)



▶ wait-signal 방식 : 세마포어(semaphore)

- n개 자원을 사용하려는 m개 멀티스레드의 원활한 관리
- 카운터를 활용!
- 자원을 소유하지 못한 스레드는 대기(wait)
- 자원을 다 사용한 스레드는 알림(signal)
- (사실 얘도 Busy-wait, sleep-wait로 구분됨)



Mutex (사실 Mutex 자체가 Mutual Exclusion 의 줄임말)

▶ 잠김/열림 중 한 상태를 가지는 락 변수 이용

- 한 스레드만 임계구역에 진입시킴, 다른 스레드는 큐에 대기
- sleep-waiting lock 기법

▶ 구성 요소

• 1. 락 변수

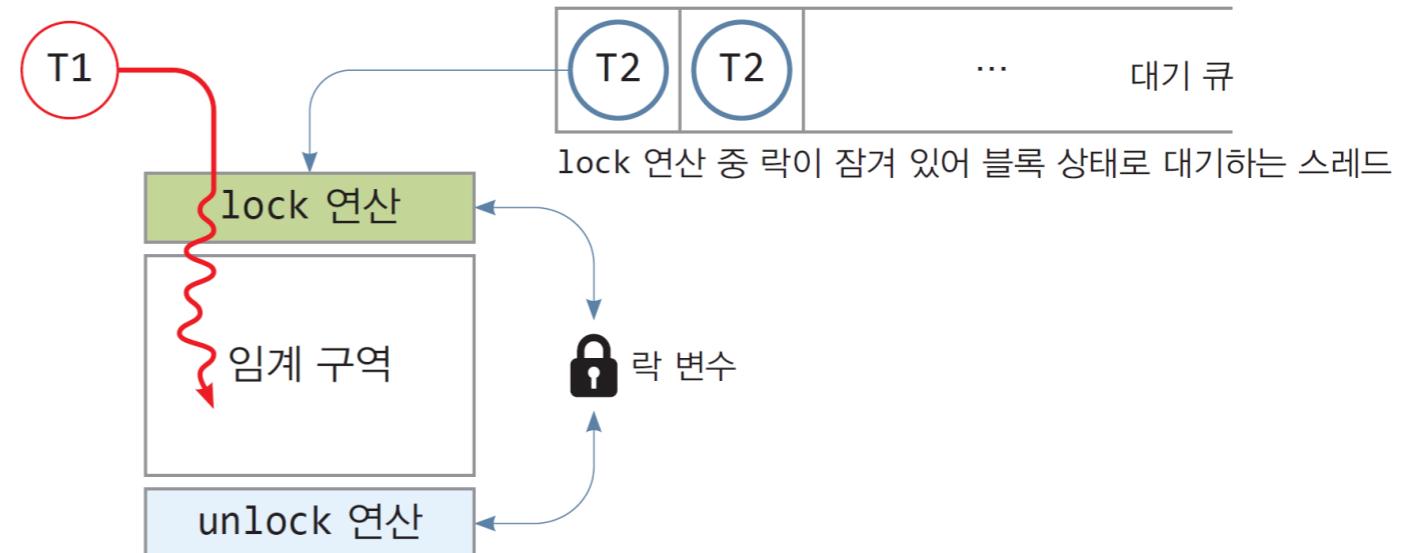
- true/false 중 한 값
- true : 락을 잠근다. 락을 소유한다.
- false : 락을 연다. 락을 해제한다.

• 2. 대기 큐

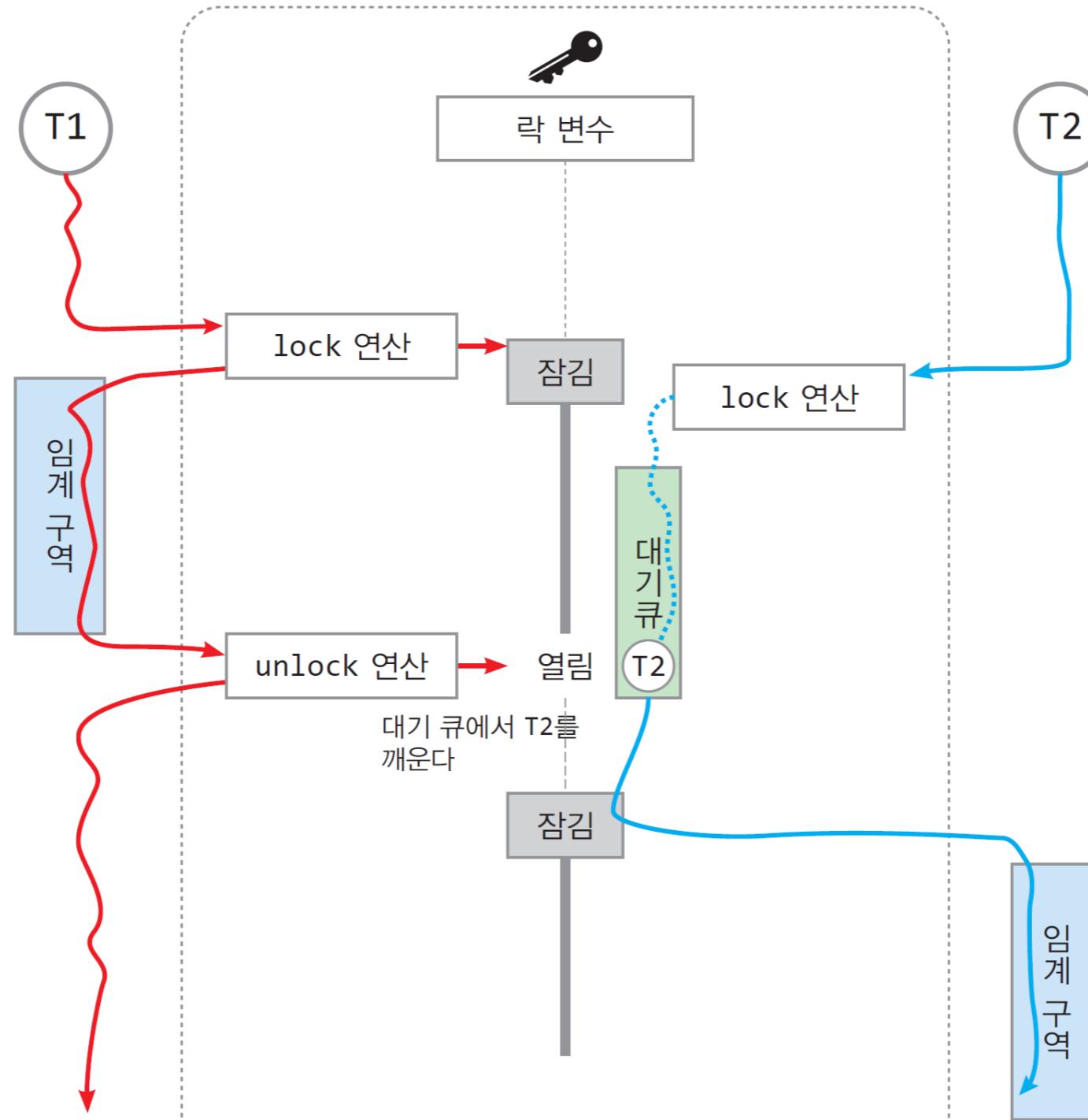
- 락이 열리기를 기다리는 스레드 큐

• 3. 연산

- lock 연산(임계구역의 entry 코드)
 - 락이 잠김 상태(lock = true)이면, 현재 스레드를 Block 상태로 만들고 대기 큐에 삽입
 - 락이 열린 상태이면, 락을 잠그고 임계구역 진입
- unlock 연산(임계구역의 exit 코드)
 - lock = false, 락을 열린 상태로 변경
 - 대기 큐에서 기다리는 스레드 하나 깨움



Mutex figure



mutex example

```
pthread_mutex_t lock;           // 뮤텍스락 변수 생성
pthread_mutex_init(&lock, NULL); // 뮤텍스락 변수 초기화
pthread_mutex_lock(&lock);     // 임계구역 enrty코드. 뮤텍스락 잠그기

... 임계구역코드 ...

pthread_mutex_unlock(&lock); // 임계구역 exit코드. 뮤텍스락 열기
```

```
----- Run "sync_mutex.c" -

Start!
sum = 0
-----
```

```
taejune@Taejunes-MBP2021 codes % time ./a.out
Start!
sum = 0
./a.out 0.01s user 0.01s system 127% cpu 0.019 total
taejune@Taejunes-MBP2021 codes %
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int count = 200000;

int sum = 0; // global variable, shared data
pthread_mutex_t lock;

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        pthread_mutex_lock(&lock); // entry
        sum += 1;
        pthread_mutex_unlock(&lock); // exit
    }
    return 0;
}

void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        pthread_mutex_lock(&lock); // entry
        sum -= 1;
        pthread_mutex_unlock(&lock); // exit
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_mutex_init(&lock, NULL); // init mutex

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

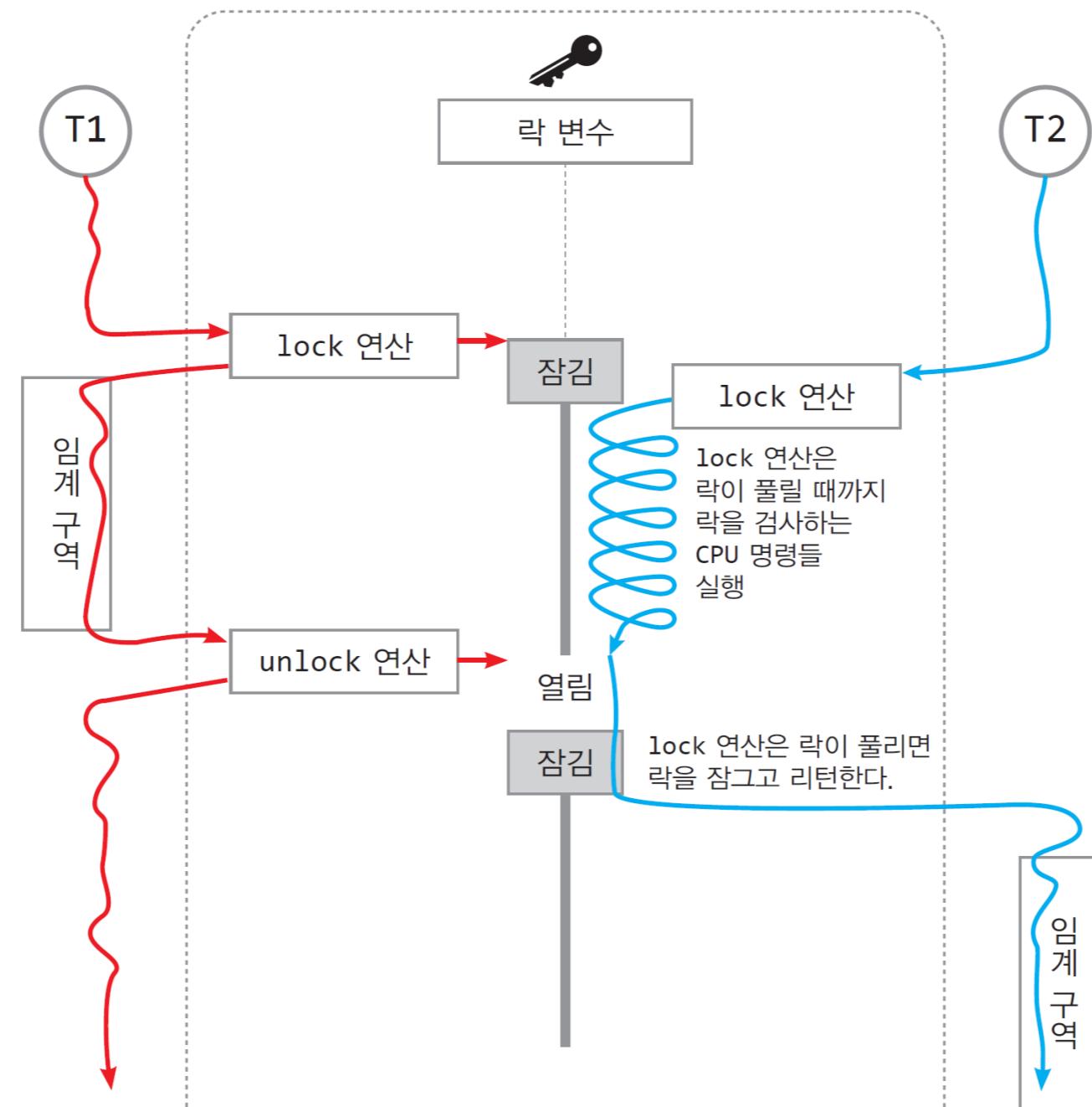
    printf("sum = %d\n", sum);

    pthread_mutex_destroy(&lock); // destroy mutex
    return 0;
}
```

Spinlock

- ▶ 앞선 예제에서 **while(lock)**으로 대기하던 것.
- ▶ 뮤텍스의 non-blocking 모델 (Busy-waiting)
 - 락이 잠겨 있을 때 블록되지 않고 락이 풀릴 때까지 검사 코드 실행
 - 큐가 아닌 While loop로 대기!
 - 단일 CPU(단일 코어)를 가진 운영체제에서 비효율적, 멀티 코어에 적합
 - 단일 코어 CPU에서 의미 없는 CPU 시간 낭비
 - 스피너락을 검사하는 스레드의 타임 슬라이스가 끝날 때까지 다른 스레드 실행 안 됨, 다른 스레드의 실행 기회 뺏음
 - 락을 소유한 다른 스레드가 실행되어야 락이 풀림.
 - 임계구역의 실행 시간이 짧은 경우 효과적

Spinlock figure



spinlock example

그대를 위한 복불
only for LINUX !!!

```
pthread_spinlock_t lock;          // 스피너 변수 생성
pthread_spin_init(&lock, NULL);   // 스피너 변수 초기화
pthread_spin_lock(&lock);        // 임계구역 entry 코드. 스피너 잠그기
... 임계구역코드 ....
pthread_spin_unlock(&lock);     // 임계구역 exit 코드. 스피너 열기
```

```
taejune@raspberrypi:~ $ time ./a.out
Start!
sum = 0

real    0m0.129s
user    0m0.097s
sys     0m0.019s
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

const int count = 200000;

int sum = 0; // global variable, shared data
pthread_spinlock_t lock;

void* myThread1(void *p) {
    for(int i = 0; i < count; i++) {
        pthread_spin_lock(&lock); // entry
        sum += 1;
        pthread_spin_unlock(&lock); // exit
    }
    return 0;
}

void* myThread2(void *p) {
    for(int i = 0; i < count; i++) {
        pthread_spin_lock(&lock); // entry
        sum -= 1;
        pthread_spin_unlock(&lock); // exit
    }
    return 0;
}

int main() {
    pthread_t tid1, tid2; // thread id
    int count = 200000;
    int *ret1, *ret2;

    printf("Start!\n");

    pthread_spin_init(&lock, NULL); // init spin

    pthread_create(&tid1, NULL, myThread1, NULL);
    pthread_create(&tid2, NULL, myThread2, NULL);
    pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
    pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

    printf("sum = %d\n", sum);

    pthread_spin_destroy(&lock); // destroy spin
    return 0;
}
```

Mutex vs Spinlock

▶ 1. 락이 잠기는 시간이 긴 경우 : 뮤텍스

- 락을 얻지 못했을 때, CPU를 다른 스레드에게 양보하는 것이 효율적
- 락이 잠기는 시간이 짧은 경우 : 스피너락이 효율적

▶ 2. 단일 CPU를 가진 시스템 : 뮤텍스

- 단일 CPU에서 스피너락은 크게 의미 없음

▶ 3. 멀티 코어(멀티 CPU)를 가진 시스템 : 스피너락

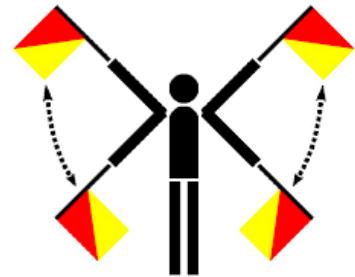
- 잠자고 깨는 컨텍스트 스위칭 없이 바로 자원 사용
- 임계구역은 가능한 짧게 작성하므로

▶ 4. 사용자 응용프로그램 : 뮤텍스, 커널 코드 : 스피너락

- 커널 코드나 인터럽트 서비스 루틴을 빨리 실행되어야 하고, 인터럽트 서비스 루틴 내에서 잠잘 수 없기 때문

▶ 5. 스피너락을 사용하면 기아 발생 가능

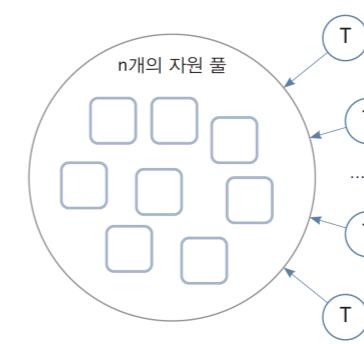
- 스피너락은 무한 경쟁 방식이어서 기아가 발생 가능
- 락을 소유한 스레드가 락을 풀지 않고 계속 실행하거나 종료해버린 경우, 코딩이 잘못된 경우 시



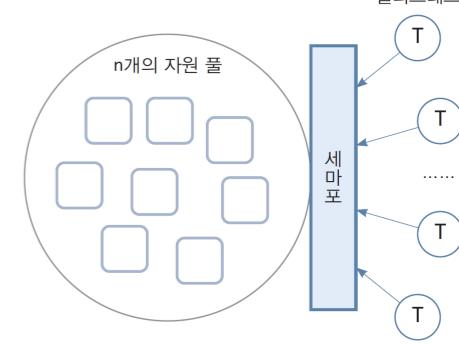
세마포어

- ▶ **n개의 공유 자원을 다수 스레드가 공유하여 사용하도록 돋는 자원 관리 기법**
 - 보다 다수의 자원을 다수의 프로세스/쓰레드를 대상으로...
 - e.g., n개의 프린터가 있는 경우, 프린터를 사용하고자 하는 다수 스레드의 프린터 관리
 - 세마포어는 동시에 여러 개의 프로세스/스레드가 임계 구역에 접근할 수 있도록 카운터를 가지고 있는 입구! → 일정 수가 초과하면 못들어갑니다!

- ▶ **구성 요소 (counter semaphore)**
 - 1. 자원 : n 개
 - 2. 대기 큐 : 자원을 할당받지 못한 스레드들이 대기하는 큐
 - 3. counter 변수
 - 사용 가능한 자원의 개수를 나타내는 정수형 전역 변수
 - n으로 초기화(counter = n)
 - 4. P/V 연산
 - P 연산 (wait 연산): 자원 요청 시 실행하는 연산 (자원 사용 허가를 얻는 과정)
 - V 연산 (signal 연산): 자원 반환 시 실행하는 연산 (자원 사용이 끝났음을 알리는 과정)



(a) 멀티스레드가 n개의 자원을 활용하려는 상황



(b) 세마포를 이용하여 멀티스레드가 n개의 자원을 원활히 사용할 수 있도록 관리

P 연산과 V 연산

- ▶ P/V를 wait/signal로 표기하기도 함
 - P 연산 : 자원 사용을 허가하는 과정, 사용가능 자원 수 1 감소(counter--)
 - V 연산 : 사용 사용을 마치는 과정, 사용가능 자원 수 1 증가(counter++)
- ▶ 세마포 종류 2가지 - 자원을 할당받지 못한 경우의 행동에 따라 구분

```
// busy-wait 세마포
P(S) {
    while S <=0; // 아무것도 하지 않음 (반복문)
    S--;
}

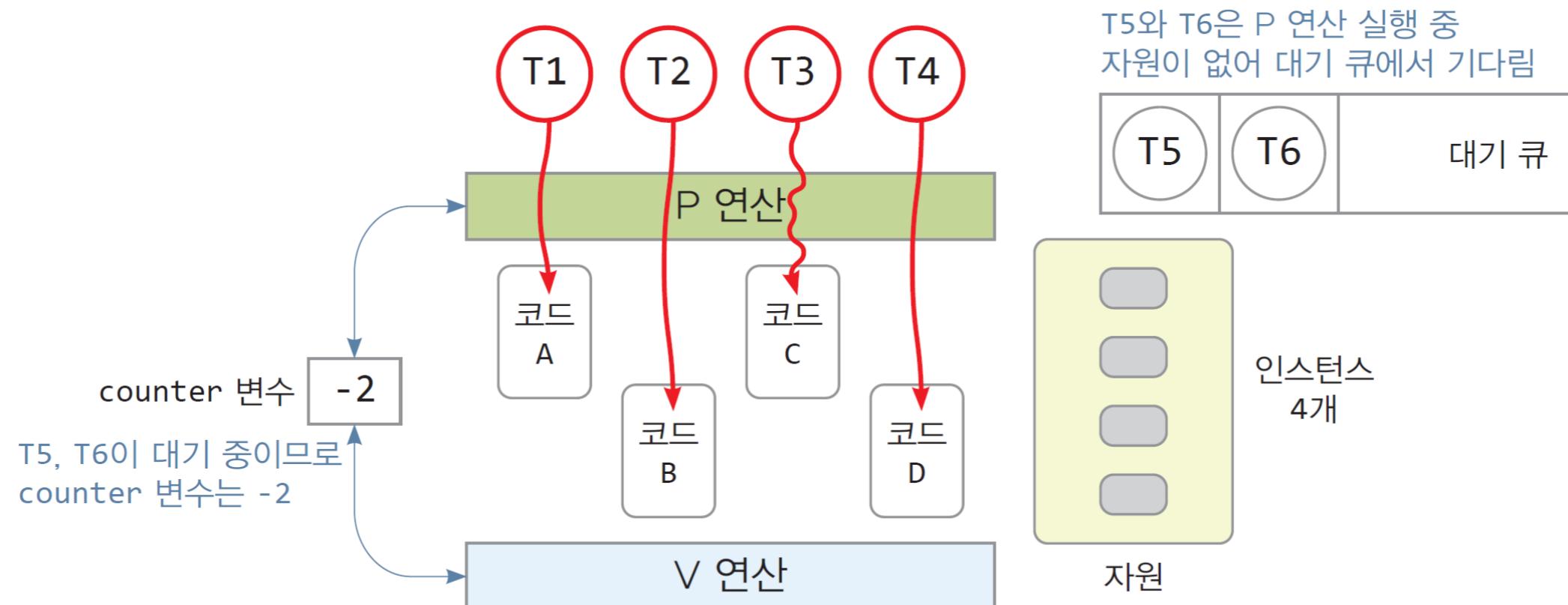
V(S) {
    S++;
}
```

```
// sleep-wait 세마포
P(S) {
    S--;
    if S < 0
        // 이 프로세스를 재움 큐에 추가 (잠듦)
}

V(S) {
    S++;
    if S <= 0
        // 재움 큐로부터 프로세스를 제거 (깨어남)
}
```

세마포어를 통한 자원 관리 예제

- 4개의 인스턴스를 가진 자원에 대해, 4개의 스레드(T1~T4)가 할당 받아 사용, 2개의 스레드 T5와 T6는 자원을 기다리고 있는 상태
- counter 변수는 사용 가능한 자원의 개수를 나타냄
 - 음수이면 대기 중인 스레드의 수를 나타냄



세마포어의 특별한 경우: 이진 세마포어

- ▶ 이진 세마포(binary semaphore)
 - 자원이 1개 있는 경우 멀티스레드 사이의 자원 관리
 - 1개의 자원에 대해 1개의 스레드만 액세스할 수 있도록 보호
 - 뮤텍스와 매우 유사
- ▶ 구성 요소
 - 1. 세마포 변수 S
 - 0과 1 중 하나를 가지는 전역 변수, S는 1로 초기화
 - 2. 대기 큐
 - 사용 가능한 자원이 생길 때까지 스레드들이 대기하는 큐
 - 스레드 스케줄링 알고리즘 필요
 - 3. 2개의 원자 연산
 - wait 연산(P 연산) – 자원 사용 허가를 얻는 과정
 - S가 1 감소시키고, 0보다 작으면 대기 큐에서 잠듦, 0보다 크거나 같으면, 자원 사용하는 코드 실행
 - signal 연산(V 연산) – 자원 사용이 끝났음을 알리는 과정
 - S를 1증가시키고, 0보다 크면 그냥 리턴, 0보다 작거나 같으면

semaphore example

그대를 위한 복불
only for LINUX !!!

```
sem_t sem;           // 세마포 구조체 생성
sem_wait(&sem);    // P 연산. 자원사용 요청
... 할당받은 자원 활용 ...
sem_post(&sem);    // V 연산. 자원사용 끝
```

```
taejune@raspberrypi:~ $ gcc temp.c -lpthread ; ./a.out
Start!
1. Semaphore: 3
t4 uses the resource. Remaining 2
t3 uses the resource. Remaining 1
t5 uses the resource. Remaining 0
t4 finished using the resource. Remaining 1
t2 uses the resource. Remaining 0
t3 finished using the resource. Remaining 1
t5 finished using the resource. Remaining 2
t1 uses the resource. Remaining 1
t2 finished using the resource. Remaining 2
t1 finished using the resource. Remaining 3
2. Semaphore: 3
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t sem;
void* myThread1(void *p) {
    int cnt = -1;

    sem_wait(&sem); // P
    sem_getvalue(&sem, &cnt); // get the semaphore cnt
    printf("%s uses the resource. Remaining %d \n", (char*)p, cnt);
    sleep(1);
    sem_post(&sem); // V

    sem_getvalue(&sem, &cnt); // get the semaphore cnt
    printf("%s finished using the resource. Remaining %d \n", (char*)p, cnt);
}

int main() {
    int cnt = -1;
    pthread_t tid[5];
    char *name[] = {"t1", "t2", "t3", "t4", "t5" };

    printf("Start!\n");

    sem_init(&sem, 0, 3); // set the semaphore count to 3
    sem_getvalue(&sem, &cnt);
    printf("1. Semaphore: %d \n", cnt);

    for(int i = 0; i < 5 ; i++)
        pthread_create(&tid[i], NULL, myThread1, (void*)name[i]);
    for(int i = 0; i < 5 ; i++)
        pthread_join(tid[i], NULL);

    sem_getvalue(&sem, &cnt);
    printf("2. Semaphore: %d \n", cnt);

    sem_destroy(&sem);
    return 0;
}
```

뮤텍스와 세마포어의 차이점

- ▶ 가장 큰 차이점은 동기화 대상의 갯수
 - Mutex는 동기화 대상이 only 1개일 때 사용
 - Semaphore는 동기화 대상이 1개 이상일 때 사용
- ▶ 세마포어는 뮤텍스가 될 수 있지만, 뮤텍스는 세마포어가 될 수 없음
 - Mutex는 0, 1로 이루어진 이진 상태를 가지므로 Binary Semaphore!
- ▶ Mutex는 자원 소유 가능 + 책임을 가지는 반면, Semaphore는 자원 소유 불가
 - 뮤텍스는 상태 0, 1 뿐이므로 Lock 가질 수 있음
- ▶ Mutex는 소유하고 있는 스레드만이 현재 Mutex를 해제할 수 있음
 - 반면, Semaphore는 Semaphore를 소유하지 않는 스레드가 Semaphore를 해제할 수 있음

세마포어의 오사용

- ▶ 세마포어를 잘 못 쓴다면...

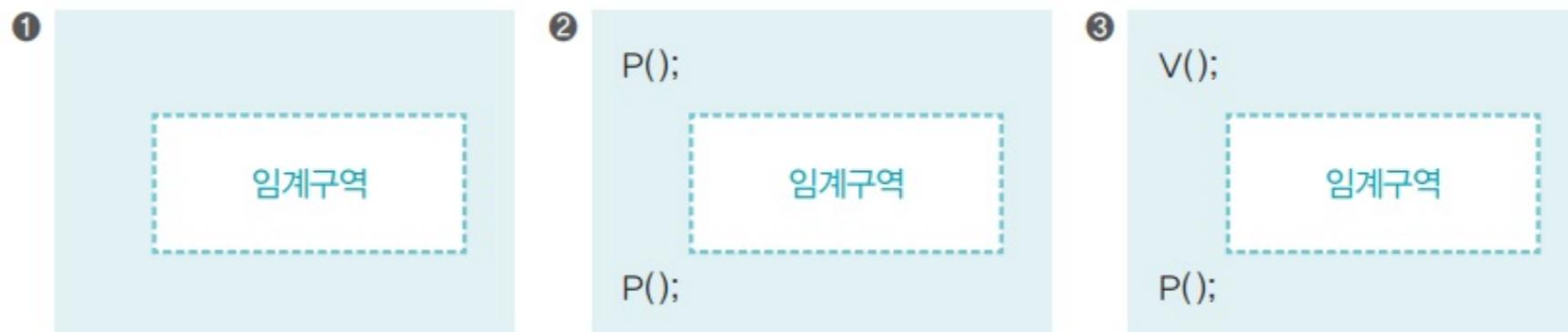


그림 5-29 세마포어의 잘못된 사용 예

- 프로세스가 세마포어를 사용하지 않고 바로 임계구역에 들어간 경우로 임계구역을 보호할 수 없음
- `P`를 두 번 사용하여 `wake_up` 신호가 발생하지 않은 경우로 프로세스 간의 동기화가 이루어지지 않아 세마포어 큐에서 대기하고 있는 프로세스들이 무한 대기에 빠짐
- `P`와 `V`를 반대로 사용하여 상호 배제가 보장되지 않은 경우로 임계구역을 보호할 수 없음
- ▶ (C/C++에서의 포인터 문제와 유사!)

Monitor (모니터)

- ▶ 공유 자원을 내부적으로 숨기고 공유 자원에 접근하기 위한 인터페이스만 제공함으로써 자원을 보호하고 프로세스 간에 동기화를 시킴 (\rightarrow 추상화)
- ▶ 작동원리
 - 임계구역으로 지정된 자원에 접근하고자 하는 프로세스는 직접 P나 V를 사용하지 않고 모니터에 작업 요청
 - 모니터는 요청받은 작업을 모니터 큐에 저장한 후 순서대로 처리하고 그 결과만 해당 프로세스에 알려줌
- ▶ 일종의 클래스 개념
 - c.f., 스마트 포인터



그림 5-30 모니터의 작동 원리

P1 increase(10);
P2 increase(5);

```
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

동기화 때문에 발생할 수 있는 문제

우선순위 역전!

- ▶ Priority inversion
 - 스레드의 동기화로 인해 높은 순위의 스레드가 낮은 스레드보다 늦게 스케줄링되는 현상
 - 우선순위를 기반으로 스케줄링하는 실시간 시스템에서 스레드 동기화로 인해 발생
- ▶ 실시간 시스템의 근본이 붕괴된다는 점에서 큰 문제
 - 높은 순위의 쓰레드가 늦게 실행되면 심각한 문제 발생 가능
 - 낮은 순위의 스레드가 길어지면 더욱 더 심각한 문제 발생
 - 보통 높은 우선순위의 작업이 더 중요한 경우가 많기에...

예제

* 3가지 가정

1) 3개의 스레드

T3 : 높은 순위의 스레드

T2 : 중간 순위의 스레드

T1 : 낮은 순위의 스레드

2) T1과 T3가 공유 변수 사용

세마포로 동기화

3) T2는 공유 변수 사용하지 않음

* 상황 발생

1) T1이 먼저 도착, P 연산 후 자원 할당

2) 그 다음 T3 도착, T1 중단시키고 T3 실행, T3는 P 연산 내에서 잠듦

3) T1 다시 실행

4) T2 도착, T2는 T1보다 순위가 높고 공유 변수 사용하지 않기 때문에 실행
-> 우선순위 역전 발생!

5) T2 종료 후 T1 실행

6) T1의 V 연산 후 T3 실행



해결책

▶ 우선순위 올림(priority ceiling)

- 스레드가 공유 자원을 소유하게 될 때, 스레드의 우선순위를 미리 정해진 높은 우선순위로 일시적으로 올림
- 선점되지 않고 빨리 실행되도록 유도

▶ 우선순위 상속(priority inheritance)

- 낮은 순위의 스레드가 공유 자원을 가지고 있는 동안,
- 높은 순위의 스레드가 공유 자원을 요청하면,
- 공유 자원을 가진 스레드의 우선순위를 요청한 스레드보다 높게 설정하여 빨리 실행시킴

생산자-소비자 문제

아주아주 유명한 동기화 문제

Producer-consumer problem

- ▶ 공유버퍼를 사이에 두고, 공유버퍼에 데이터를 공급하는 생산자들과 데이터 읽고 소비하는 소비자들이 공유버퍼를 문제 없이 사용하도록 생산자와 소비자를 동기화시키는 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
 - ⇒ 그냥 'if(buf_cnt == 0)'로 하면 안되는 이유 :p

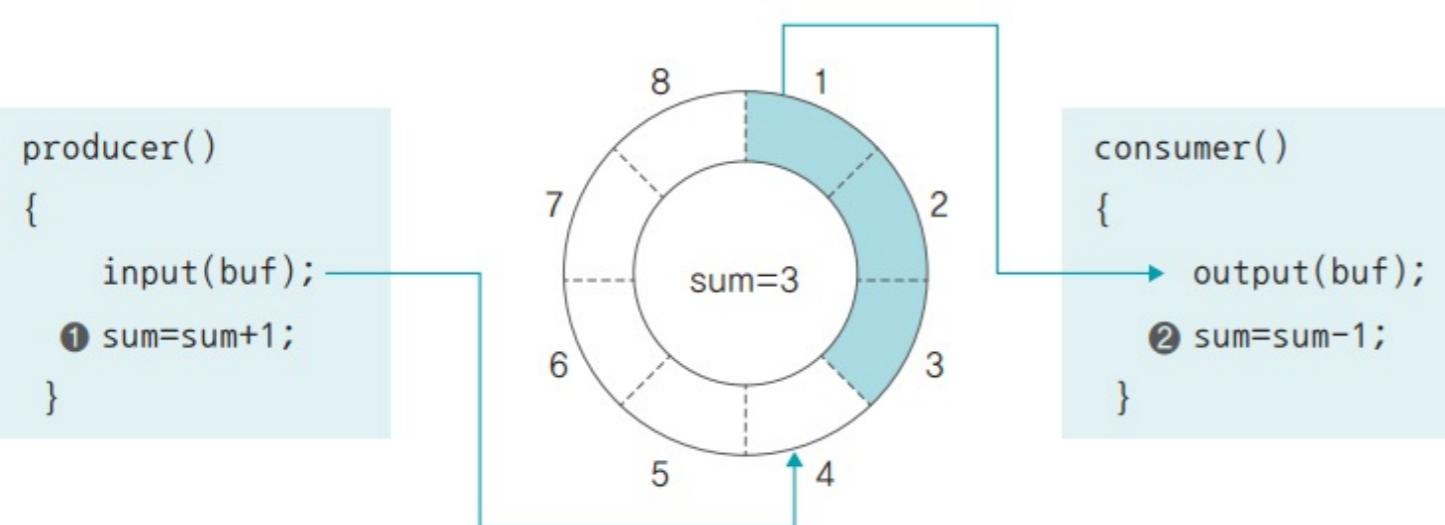


그림 5-12 생산자-소비자 문제

① 생산자 sum = 4
② 소비자 sum = 2
결과 sum = 2

(a) ①, ② 순서로 실행

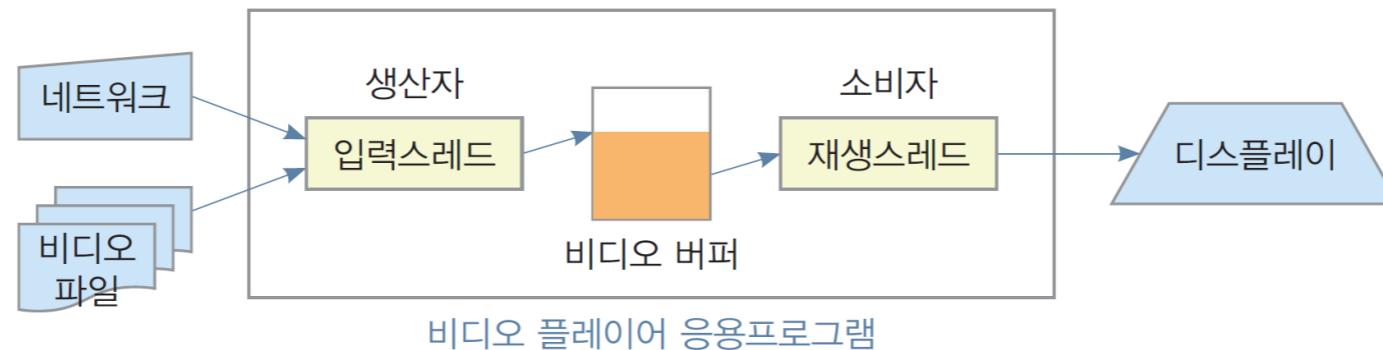
② 소비자 sum = 2
① 생산자 sum = 4
결과 sum = 4

(b) ②, ① 순서로 실행

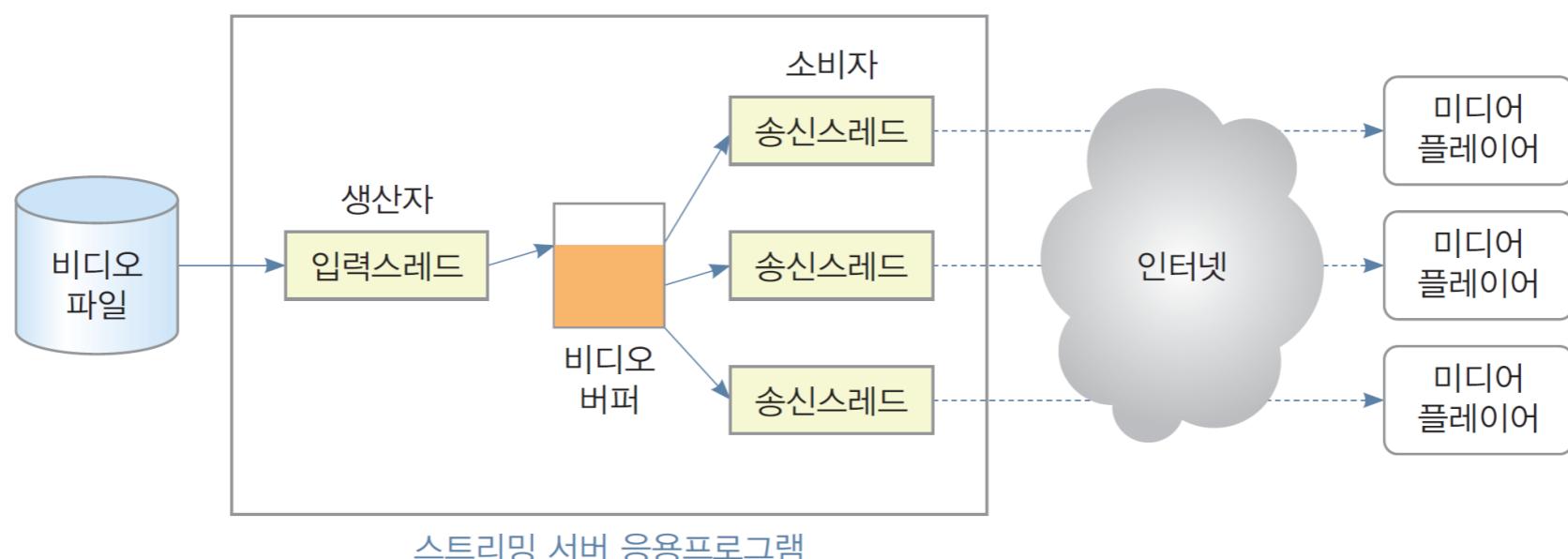
그림 5-13 실행 순서에 따른 결과 차이

자주 발생 하는 동기화 문제!

- ▶ 하다못해 키보드 입력도 생산자-소비자 관계..!



(a) 미디어 플레이어의 구조(1:1 생산자 소비자 관계)



(b) 스트리밍 서버의 구조(1:N 생산자 소비자 관계)

생산자-소비자에서 고려해야할 3가지 문제점

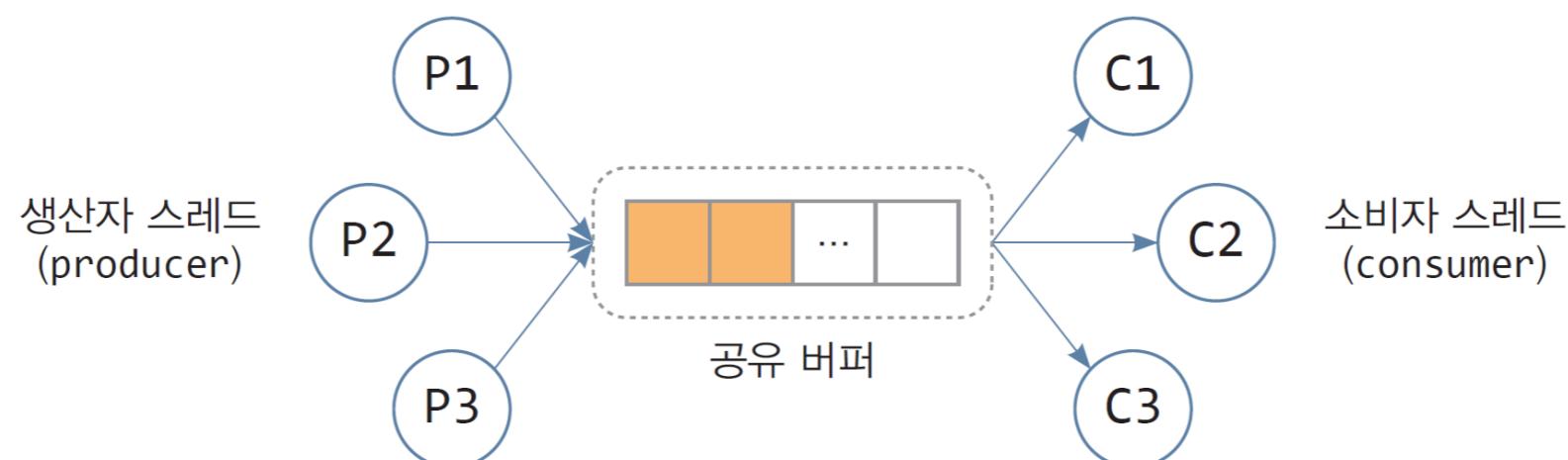
- ▶ 상호 배제 해결
 - 생산자들과 소비자들의 공유 버퍼에 대한 상호 배제

▶ 비어 있는 공유 버퍼 문제

- 비어 있는 공유버퍼를 소비자가 읽으면 안된다!

▶ 꽉 찬 공유버퍼 문제

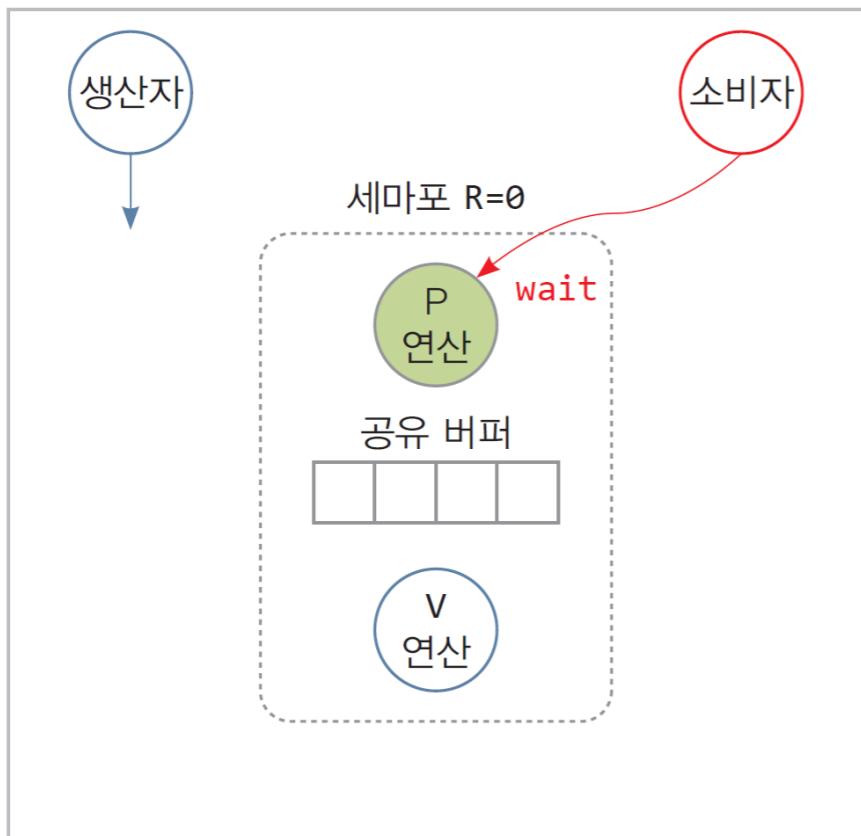
- 꽉 찬 공유버퍼에 생산자가 더 이상 데이터를 입력하면 안된다!



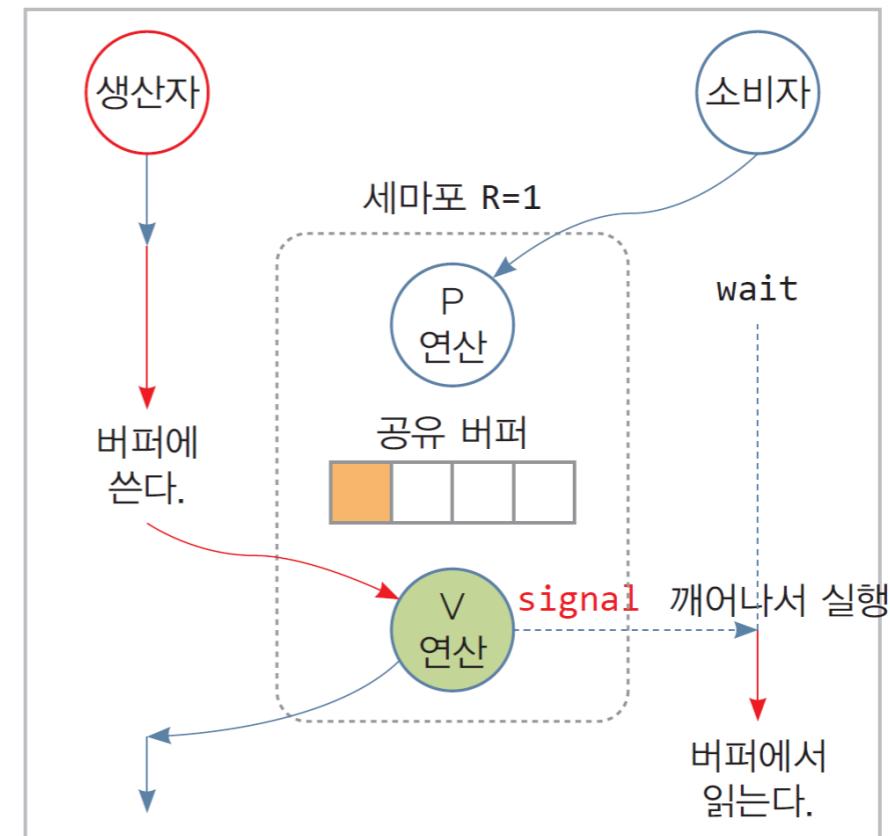
세마포어 2개를 이용하면 해결할 수 있습니다!

▶ 세마포어1. 읽기용 세마포어: 비어있는 버퍼 문제 해결 (읽기 가능한 버퍼 갯수 확인)

(1) 버퍼가 비어 있는 상태에서
소비자가 읽으려고 할 때



(2) 빈 버퍼에 생산자가 쓸 때



소비자 : 버퍼에서 읽기 전 P 연산 실행

P 연산 : 버퍼가 빈 경우(R=0),
소비자가 잠을 자면서 대기하도록 작성

생산자 : 버퍼에 데이터를 기록하고 V 연산 실행

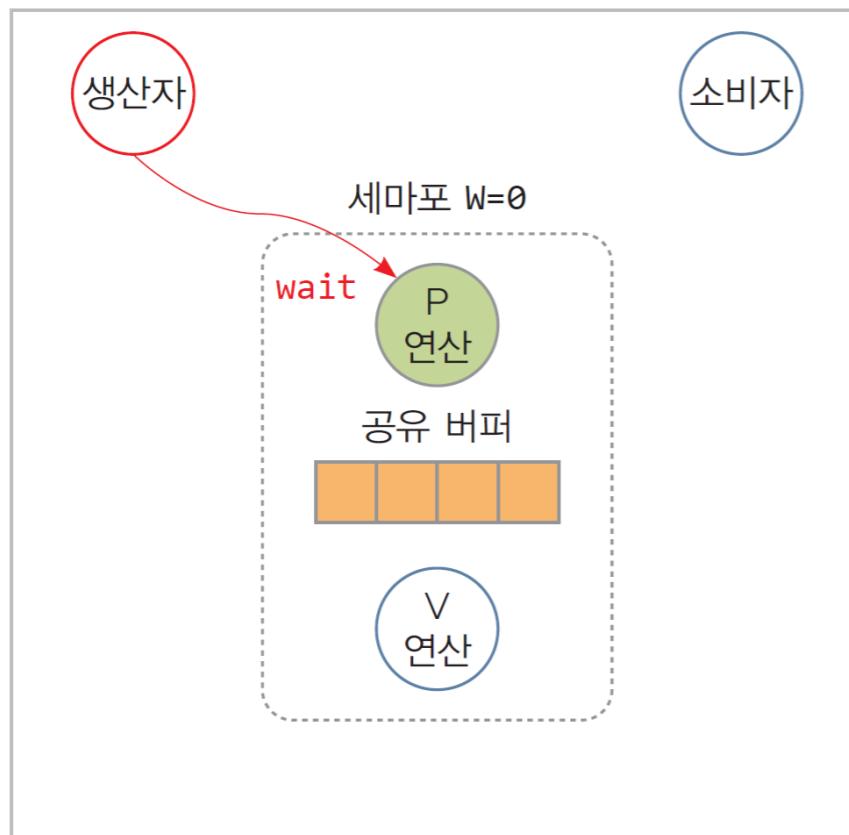
V 연산 : 세마포 변수 R을 1증가(R=1),
대기 중인 소비자를 깨우도록 작성

소비자 : 깨어나면 P 연산을 마치고 공유버퍼에서
읽는다. P 연산에서 세마포 R을 1감소(R=0)

세마포어 2개를 이용하면 해결할 수 있습니다!

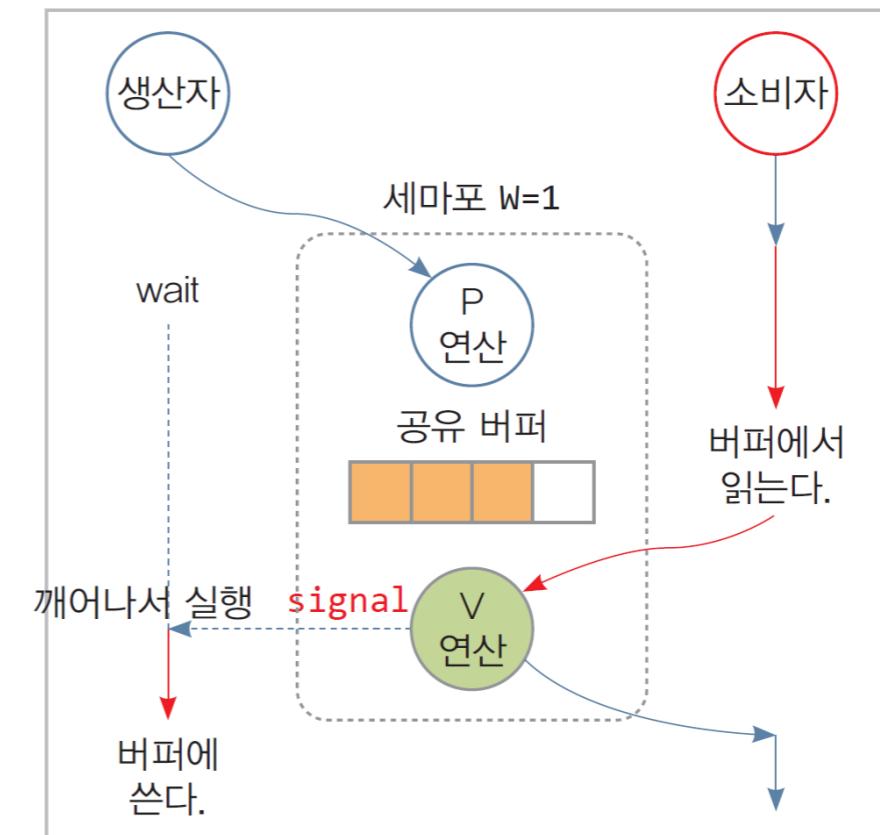
▶ 세마포어2. 쓰기용 세마포어: 비어있는 버퍼 문제 해결 (쓰기 가능한 버퍼 갯수 확인)

(1) 공유 버퍼가 꽉 찬 상태에서
생산자가 쓰려고 할 때



생산자 : 버퍼에 쓰기 전 P 연산 실행
P 연산 : 버퍼가 꽉 찬 상태($W=0$)이면,
생산자가 잠을 자면서 대기하도록 작성

(2) 버퍼에 데이터가 있는 경우
소비자가 읽을 때



소비자 : 버퍼에서 데이터를 읽은 후 V 연산 실행
V 연산 : 세마포 변수 W를 1증가시키고($W=1$),
대기 중인 생산자를 깨우도록 작성
생산자 : 깨어나면 P 연산을 마치고 공유버퍼에
쓴다. P 연산에서 세마포 W를 1감소($W=0$)

이것을 알고리즘으로 표현하면...

- R : 버퍼에 읽기 가능한 버퍼의 개수. 0이면(비어있는 경우) 대기
- W : 버퍼에 있는 쓰기 가능한 버퍼의 개수. 0이면(꽉 차있는 경우) 대기
- M : 뮤텍스. 생산자 소비자 모두 사용

Consumer { // 소비자 스레드

```
while(true) {
    P(R); // 세마포 R에 P/wait 연산을 수행하여
    // 버퍼가 비어 있으면(읽기 가능한 버퍼 수=0) 대기한다.
```

뮤텍스(M)를 잠근다.

공유버퍼에서 데이터를 읽는다. // 임계구역 코드

뮤텍스(M)를 얻다.

```
V(W); // 세마포 W에 대해 V/signal 연산을 수행하여
```

// 버퍼가 비기를 기다리는 **Producer**를 깨움

}

}

Producer { // 생산자 스레드

```
while(true) {
    P(W); // 세마포 W에 P/wait 연산을 수행하여
    // 버퍼가 꽉 차 있으면(쓰기 가능한 버퍼 수=0) 대기
```

뮤텍스(M)를 잠근다.

공유버퍼에 데이터를 저장한다. // 임계구역 코드

뮤텍스(M)를 얻다.

```
V(R); // 세마포 R에 대해 V/signal 연산을 수행하여
```

// 버퍼에 데이터가 저장되기를 기다리는 **Consumer**를 깨움.

}

}

Summary

- ▶ 프로세스간 통신
 - 프로세스들 간에는 기본적으로 통신이 안됨! (가상메모리) → 별도의 수단이 필요
 - Pipe, Named Pipe, Message queue, Shared memory, Memory map, Socket ...
- ▶ 하나의 자원을 여럿이서 두고 쓴다?
 - 공유자원의 개념 → 읽고 쓰기 순서 때문에... 데이터가 오염되요!
 - 이를 막기위해선? Mutual exclusion, 상호배제의 원칙이 적용되어야 함.
 - 한번에 한놈만!
- ▶ 동기화 기법들
 - Mutex, spinlock, Semaphore
 - 이외에도 피터슨, 데커 알고리즘등 소프트웨어적인 기법들이 존재합니다!
 - 동기화를 하면 일반적으로 속도가 조금 떨어질 수 밖에 없어요. 그리고 우선순위의 역전도 생길 수도!
- ▶ 생산자-소비자 문제
 - 비어있는데 읽으면 안되고, 차있는데 쓰면 안된다. → 생산과 소비의 동기화
 - 두개의 세마포어를 이용해서 해결
- ▶ 동기화 관련 내용들은 필드에서도 심심치않게 접하게 됩니다!