# IPC - Interprocess Communications

## 뇌를 자극하는 TCP/IP 소켓 프로그래밍

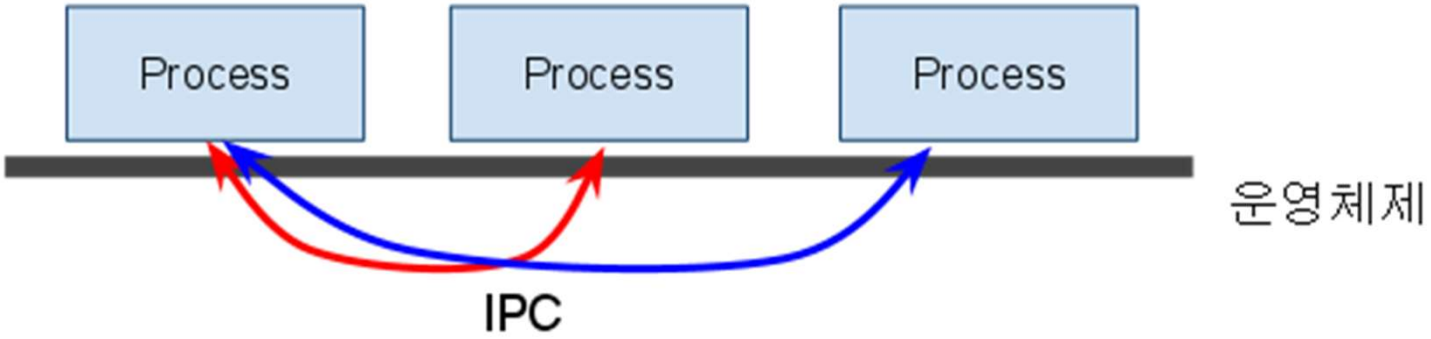
# 네트워크 프로그램과 소켓 프로그래밍

- 주로 사용하는 소켓 함수는 20개 내외로 제한적
- (서버) 응용 프로그램 개발시에는 시스템 프로그래밍 기술이 중요
  - 파일제어
  - 입/출력 제어
  - 프로세스 제어
  - **IPC(Interprocess Communications)**
  - 쓰레드 제어

# IPC - InterProcess Communication

- IPC : 프로세스간 통신을 위한 운영체제 지원 설비
  - 프로세스간 자원 공유
  - 공유 자원에 대한 접근 설정

- 한 노드 내부 통신을 위한 여러 도구의 총칭
  - 다양한 IPC 설비가 존재
  - 상황에 맞는 IPC 설비를 선택해서 사용



# IPC 의 계보

- **System V IPC** : AT & T 에서 개발
  - 오래된 인터페이스로 POSIX IPC에 비해 사용이 까다롭다.
  - 모든 유닉스에서 지원.

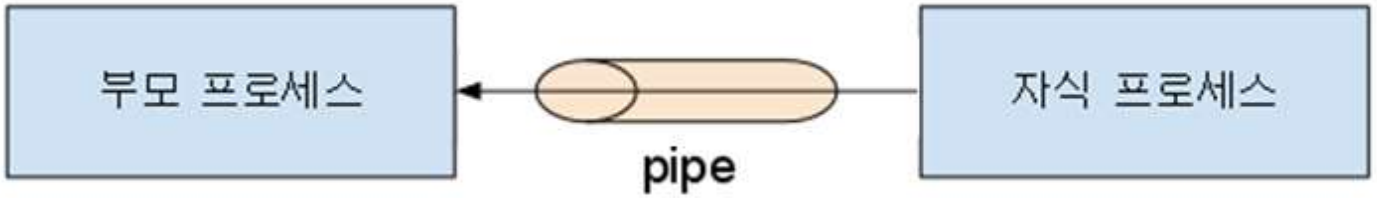
- **POSIX IPC** : 유닉스 기반의 표준 인터페이스
  - 비교적 최근에 개발. 사용하기 쉬운 세련된 인터페이스 제공
  - 유닉스 버전에 따라 사용에 제약이 있을 수 있다.

# IPC 의 종류

- 파이프(익명 파이프)
- 네임드 파이프
- 유닉스 도메인 소켓
- 시그널
- 공유 메모리
- 세마포어
- 메시지 큐

# PIPE

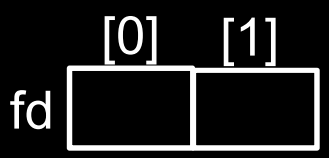
- 단방향(**one-way communication**)의 데이터 통신
- 부모 프로세스와 자식 프로세스간 통신을 위해서 사용
  - 외부 프로세스와의 통신에 사용할 수 없음.
- 익명 파이프 (**none named pipe**)라고도 불림




```
#include <unistd.h>
int pipe (int filedes[2]) ;
```

- 읽기와 쓰기 전용의 2개의 파일을 open 한다.
- filedes**[0]** will be the file descriptor for the **read end** **of pipe**.
- filedes**[1]** will be the file descriptor for the **write end** **of pipe**.

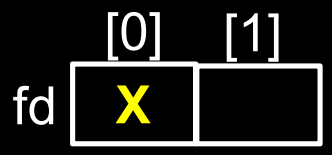
```c
int fd[2];
int buf;
int i=0;
int pid;

if (pipe(fd) < 0)
{
        perror("pipe error : ");
        return 1;
}

if ((pid = fork()) < 0)
{
        return 1;
}

// 만약 자식프로세스라면 파이프에 자신의 PID 정보를 쓴다.
else if (pid == 0)
{
        close(fd[0]);
        while(1)
        {
                i++;
                write(fd[1], (void *)&i, sizeof(i));
                sleep(1);
        }
}
// 만약 부모프로세스라면 파이프에서 데이터를 읽어 들인다.
else
{
                close(fd[1]);
                while(1)
                {
                        read(fd[0], (void *)&buf, sizeof(buf));
                        printf("> %d\n", buf);
                }
}
return 1;
```

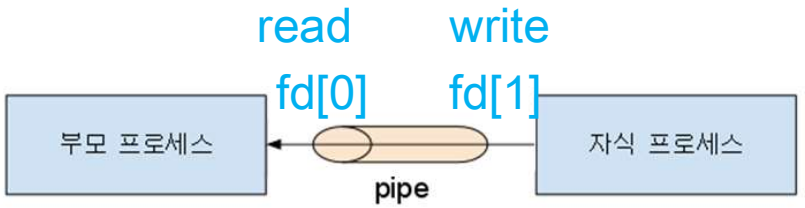


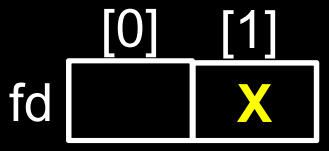

```
osnw2021@student04--10:~/lab06$ ./pipe
> 1
> 2
> 3
> 4
> 5
> 6
> 7
```

Lab pipe.c

```
osnw2021@student04--10:~/lab06$ ps -ef | grep pipe
osnw2021 16419 16402  0 10:29 pts/1    00:00:00 ./pipe
osnw2021 16420 16419  0 10:29 pts/1    00:00:00 ./pipe
```

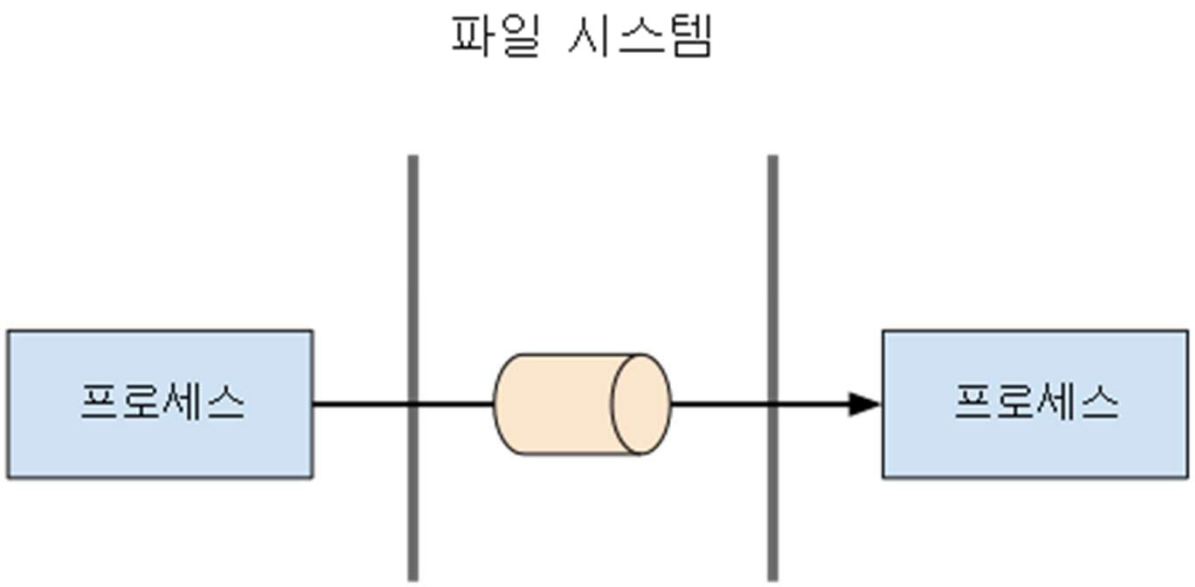



# Named PIPE

- 단방향의 데이터 통신
- 이름 있는 파이프**(named pipe)** 라고도 불림
  - (PIPE 타입의) 파일로 존재
- 부모/자식 프로세스 뿐 아니라 외부 프로세스도 사용할 수 있음.
  - 파일 이름을 이용해서 참조



# Named PIPE

- A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem.
- It can be opened by multiple processes for reading or writing.
- When processes are exchanging data via the FIFO, the kernel passes all data internally *without writing it to the filesystem*.

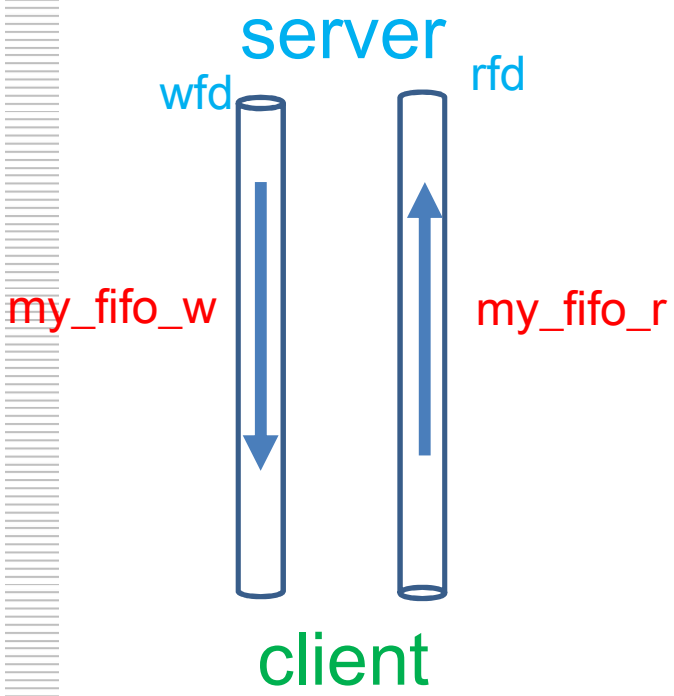
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with name pathname.
mode specifies the FIFO's permissions.

# Named PIPE

echo_server_pipe.c

Lab

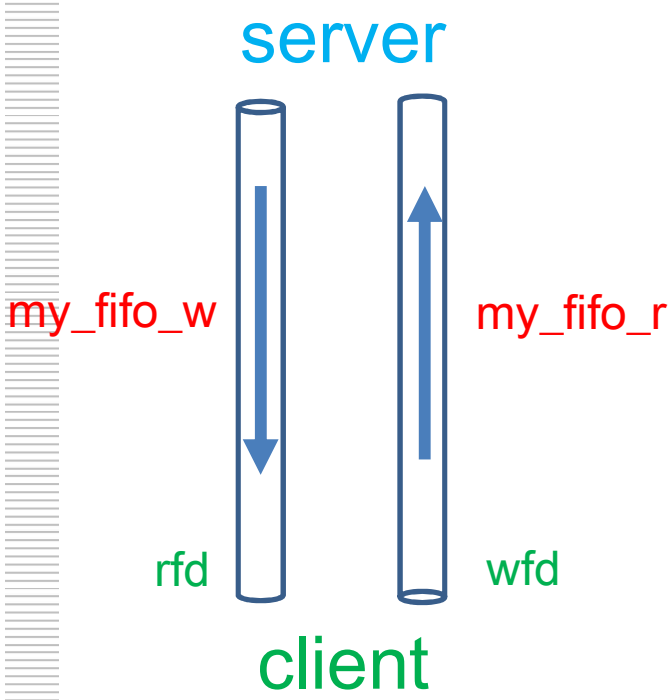



```c
#define MAXLINE 1024
int main(int argc, char **argv)
{
        int rfd, wfd;
        char buf[MAXLINE];

        mkfifo("/tmp/myfifo_r", S_IRUSR|S_IWUSR);
        mkfifo("/tmp/myfifo_w", S_IRUSR|S_IWUSR);
        if( (rfd = open("/tmp/myfifo_r", O_RDWR)) == -1)
        {
                perror("rfd error");
                return 0;
        }
        if ( (wfd = open("/tmp/myfifo_w", O_RDWR)) == -1)
        {
                perror("wfd error");
                return 0;
        }
        while(1)
        {
                memset(buf, 0x00, MAXLINE);
                if(read(rfd,buf,MAXLINE) < 0)
                {
                        perror("Read Error");
                        return 1;
                }
                printf("Read : %s", buf);
                write(wfd, buf, MAXLINE);
                lseek(wfd, 0, SEEK_SET);
        }
}
```

# Named PIPE

echo_client_pipe.c

Lab




```c
rfd = open("/tmp/myfifo_w", O_RDWR);
if(rfd < 0)
{
        perror("read open error\n");
        return 1;
}
wfd = open("/tmp/myfifo_r", O_RDWR);
if(wfd < 0)
{
        perror("write open error\n");
        return 1;
}
while(1)
{
        printf("> ");
        fflush(stdout);
        memset(buf, 0x00, MAXLINE);
        if(read(0, buf, MAXLINE) < 0)
        {
                printf("error\n");
                return 1;
        }
        if(strncmp(buf, "quit\n",5) == 0) break;
        write(wfd, buf, strlen(buf));
        read(rfd, buf, MAXLINE);
        printf("Server -> %s",buf);
}
close(wfd);
close(rfd);
```

# Named PIPE

echo_server_pipe.c
echo_client_pipe.c

**Lab**

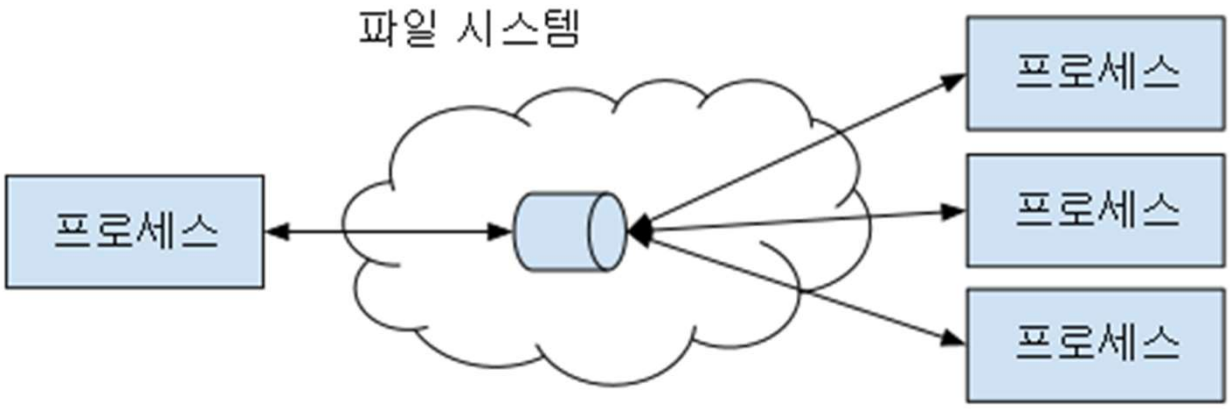
```
osnw00000000@osnw00000000-osnw:/tmp$ ls -al
total 48
drwxrwxrwt 12 root root 4096 Nov  1 02:43 .
drwxr-xr-x 19 root root 4096 Sep  5 19:04 ..
drwxrwxrwt  2 root root 4096 Sep  5 19:03 .ICE-unix
drwxrwxrwt  2 root root 4096 Sep  5 19:03 .Test-unix
drwxrwxrwt  2 root root 4096 Sep  5 19:03 .X11-unix
drwxrwxrwt  2 root root 4096 Sep  5 19:03 .XIM-unix
drwxrwxrwt  2 root root 4096 Sep  5 19:03 .font-unix
drwx------  3 root root 4096 Sep  5 19:05 snap-private-t
drwx------  3 root root 4096 Sep  5 19:04 systemd-privat
3c816bd7f6-systemd-logind.service-GX3tIr
drwx------  3 root root 4096 Sep 20 05:21 systemd-privat
3c816bd7f6-systemd-resolved.service-IQNFuS
drwx------  3 root root 4096 Sep 20 05:25 systemd-privat
3c816bd7f6-systemd-timesyncd.service-Ydwyly
drwx------  2 root root 4096 Oct  4 06:31 tmp.XBveGEpyDR
osnw00000000@osnw00000000-osnw:/tmp$ ls -al
total 48
drwxrwxrwt 12 root         root         4096 Nov  1 02:43 .
drwxr-xr-x 19 root         root         4096 Sep  5 19:04 ..
drwxrwxrwt  2 root         root         4096 Sep  5 19:03 .ICE-unix
drwxrwxrwt  2 root         root         4096 Sep  5 19:03 .Test-unix
drwxrwxrwt  2 root         root         4096 Sep  5 19:03 .X11-unix
drwxrwxrwt  2 root         root         4096 Sep  5 19:03 .XIM-unix
drwxrwxrwt  2 root         root         4096 Sep  5 19:03 .font-unix
prw-------  1 osnw00000000 osnw00000000    0 Nov  1 02:44 myfifo_r
prw-------  1 osnw00000000 osnw00000000    0 Nov  1 02:44 myfifo_w
drwx------  3 root         root         4096 Sep  5 19:05 snap-private-tmp
drwx------  3 root         root         4096 Sep  5 19:04 systemd-private-71f349
4b63ba4465863ce13c816bd7f6-systemd-logind.service-GX3tIr
drwx------  3 root         root         4096 Sep 20 05:21 systemd-private-71f349
4b63ba4465863ce13c816bd7f6-systemd-resolved.service-IQNFuS
drwx------  3 root         root         4096 Sep 20 05:25 systemd-private-71f349
4b63ba4465863ce13c816bd7f6-systemd-timesyncd.service-Ydwyly
drwx------  2 root         root         4096 Oct  4 06:31 tmp.XBveGEpyDR
osnw00000000@osnw00000000-osnw:/tmp$
```

```
osnw00000000@osnw00000000-osnw: ~/week09
osnw00000000@osnw00000000-osnw:~/week09$ ./echo_server_pipe
Read : osnw2023
Read : hello world !!!
```

```
osnw00000000@osnw00000000-osnw: ~/week09
osnw00000000@osnw00000000-osnw:~/week09$ ./echo_client_pipe
> osnw2023
Server -> osnw2023
> hello world !!!
Server -> hello world !!!
>
```

# Unix Domain Socket

- 소켓 함수와 기술을 그대로 사용
- 유닉스 영역(같은 시스템)에서의 통신에서 사용
- 유닉스 영역에 서버/클라이언트 환경을 만들 수 있음
- 서버/클라이언트 환경 구축에 용이
- 중대형 애플리케이션에서 주로 사용




```
struct sockaddr_un sockaddr;
socket(AF_UNIX, SOCK_STREAM, 0)
sprintf(sockaddr.sun_path, "filename.sock");
```

# Unix Doma

Lab

echo_server_udomain.c

```c
if( (listen_fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
{
        perror("Error : socket");
        return 0;
}

memset((void *)&server_addr, 0x00, sizeof(server_addr));
server_addr.sun_family = AF_UNIX;
strncpy(server_addr.sun_path, argv[1], strlen(argv[1]));

if(bind(listen_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1)
{
        perror("Error : bind");
        return 0;
}

if (listen(listen_fd, 5) == -1)
{
        perror("Error : listen");
        return 1;
}

while(1)
{
        memset((void *)&client_addr, 0x00, sizeof(client_addr));
        printf("accept wait\n");
        client_fd = accept(listen_fd, (struct sockaddr *)&client_addr, &addrlen);
        while(1)
        {
                if(client_fd == -1)
                {
                        printf("Accept Error ");
                        return 0;
                }
                memset(buf, 0x00, MAXLINE);
                readn = read(client_fd, buf, MAXLINE);
                if(readn == 0) break;
                printf("==> %s", buf);
                write(client_fd, buf, strlen(buf));
        }
}
return 0;
```

# Unix Domain Socket

echo_client_udomain.c

Lab

```
sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

memset((void *)&sock_addr, 0x00, sizeof(sock_addr));
sock_addr.sun_family = AF_UNIX;
strncpy(sock_addr.sun_path, argv[1], strlen(argv[1]));
clilen = sizeof(sock_addr);
connect(sockfd, (struct sockaddr *)&sock_addr, clilen);

while(1)
{
        memset(buf, 0x00, MAXLINE);
        read(0, buf, MAXLINE);
        if(strncmp(buf, "quit\n", 5) == 0)
        {
                break;
        }
        write(sockfd, buf, strlen(buf));
        read(sockfd, buf, MAXLINE);
        printf("Server -> %s", buf);
}
return 0;
```

# Unix Domain Socket

echo_server_udomain.c
echo_client_udomain.c

**Lab**

```
osnw2021@student04--10:/tmp$ ls -l
total 8
drwx------ 3 root root 4096 Aug 30 13:53 systemd-private-810
ec76e3af9-systemd-resolved.service-RRGR0G
drwx------ 3 root root 4096 Aug 30 13:53 systemd-private-810
ec76e3af9-systemd-timesyncd.service-tW1O3s
osnw2021@student04--10:/tmp$ ls -l
total 8
srwxrwxr-x 1 osnw2021 osnw2021    0 Oct 29 11:14 echo_test
drwx------ 3 root     root     4096 Aug 30 13:53 systemd-pri
f859cd1eec76e3af9-systemd-resolved.service-RRGR0G
drwx------ 3 root     root     4096 Aug 30 13:53 systemd-pri
f859cd1eec76e3af9-systemd-timesyncd.service-tW1O3s
```

```
osnw2021@student04--10:~/lab06$ ./echo_client_udomain /tmp/echo_test
osnw2021
Server -> osnw2021
hello world unix socket
Server -> hello world unix socket
quit
```

```
osnw2021@student04--10:~/lab06$ ./echo_server_udomain /tmp/echo_test
accept wait
==> osnw2021
==> hello world unix socket
accept wait
```

# 시그널

- 비동기적 사건을 알려주기 위해서 사용
- 시간을 동기화 하기 위해서 사용.

# 시그널의 종류

- 다양한 의미를 가지는 시그널이 존재
- **kill -l** 로 시그널 확인

```
user1@myubuntu:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
user1@myubuntu:~$
```

# 시그널의 종류

- 대표적인 시그널 들

표 6-4 주요 시그널

| 시그널 | 번호 | 기본 처리 | 의미 |
|---|---|---|---|
| SIGHUP | 1 | 종료 | 터미널과의 연결이 끊어졌을 때 발생한다. |
| SIGINT | 2 | 종료 | 인터럽트로 사용자가 Ctrl+c를 입력하면 발생한다. |
| SIGQUIT | 3 | 종료, 코어덤프 | 종료 신호로 사용자가 Ctrl+\을 입력하면 발생한다. |
| SIGKILL | 9 | 종료 | 이 시그널을 받은 프로세스는 무시할 수 없으며 강제로 종료된다. |
| SIGALRM | 14 | 종료 | 알람에 의해 발생한다. |
| SIGTERM | 15 | 종료 | kill 명령이 보내는 기본 시그널이다. |

# 시그널을 받았을 프로세스의 처리 방식

1. **default(기본 행동)**을 취한다.
   - 대부분의 기본행동은 프로세스 종료.

2. 시그널을 무시한다.
   - SIGSTOP, SIGKILL 은 무시할 수 없다.

3. 미리 정의한 시그널 핸들러를 실행한다.
   - 미리 지정된대로 동작
   - 프로그래머가 소스내에 미리 작성

# 시그널 함수 사용

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);
```

- handler : signum 번호를 가지는 시그널이 발생했을 때, 실행할 시그널 핸들러

  1. **SIG_DFL :** 시그널 기본 행동
  2. **SIG_IGN :** 시그널 무시
  3. 시그널 핸들러 등록

# 시그널 함수 사용

- signal 함수를 이용한 기본 행동 변경

```
int main()
{
  int i = 0;
  signal(SIGINT, SIG_IGN);
  while(1)
  {
    printf("%d\n", ++i);
    sleep(1);
  }
}
```

# 시그널 함수 사용

Lab

- signal handler를 이용한 시그널 제어

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

void sig_handler(int signo)
{
    printf("SIGINT received : %d\n“, signo);
}
int main()
{
    int i = 0;
    signal(SIGINT, (void *) sig_handler);
    while(1)
    {
            printf("%d\n", i); i++;  sleep(1);
    }
    return 1;
}
```

```
osnw2021@student04--10:~/lab07$ ./mysignal
0
1
2
3
4
^CSIGINT received : 2
5
6
7
^CSIGINT received : 2
8
9
10
```

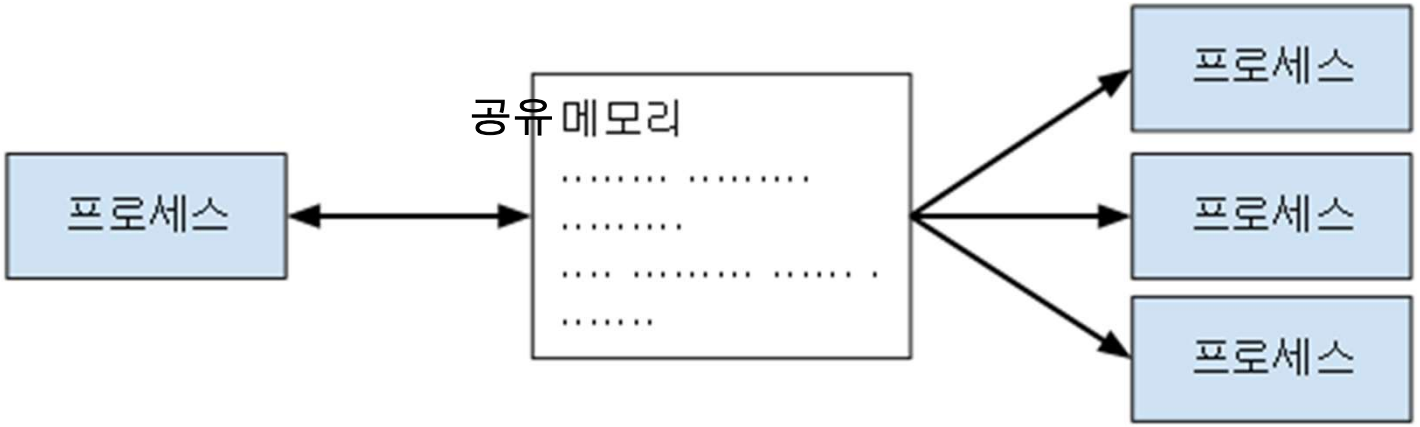
# 시그널의 장점과 단점

- (자세한) 메시지를 전달할 수 없다.
- 대기열을 가지고 있지 않다.
  - 시그널을 잃어 버릴 수 있다.
- 매우 빠르다.
- 비동기적인 사건을 다룰 수 있다.

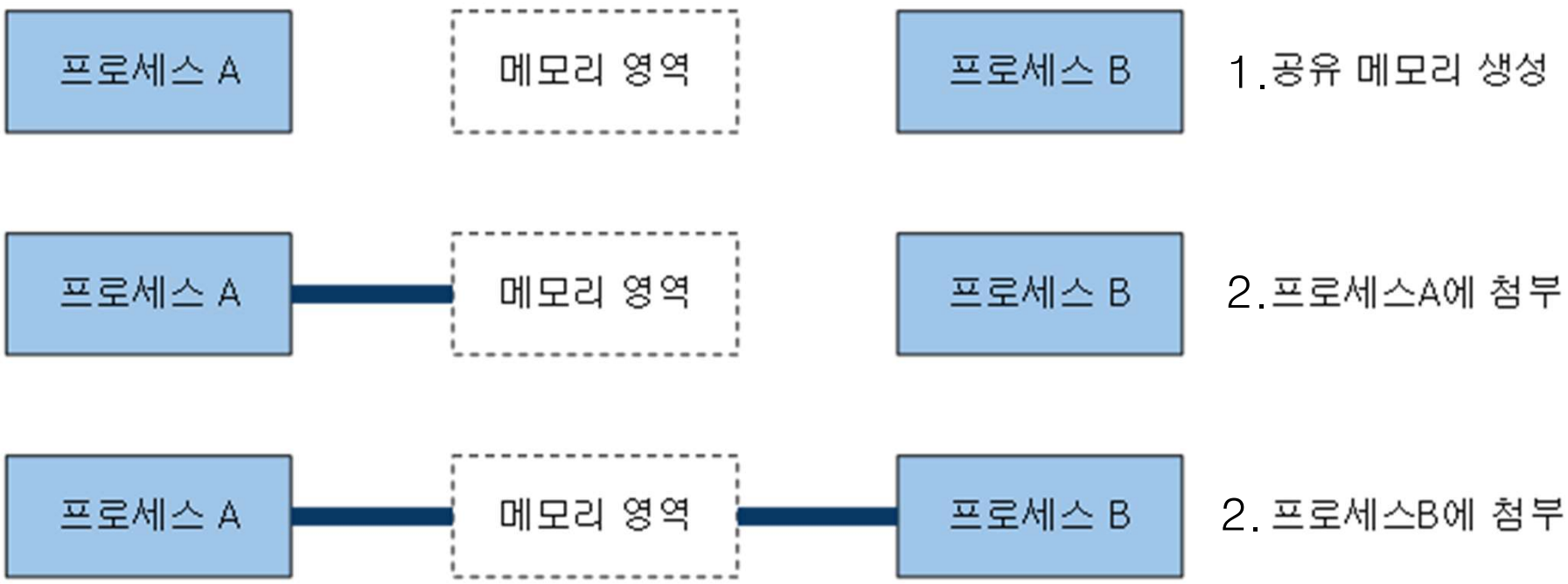
# 공유 메모리(Shared Memory)

- 프로세스는 자신만의 메모리를 가지며 외부 프로세스가 침입할 수 없도록 커널이 관리

- 공유메모리
  - 필요하다면, 여러 프로세스들이 메모리 공간을 공유
  - 다른 많은 프로세스들이 마치 자신의 메모리처럼 사용
  - 대량의 정보를 다수의 프로세스이 접근
  - 빠르고 효율적임
  - 공유 메모리 공간에 대한 접근 제어가 필요



# 공유 메모리 생성과 첨부



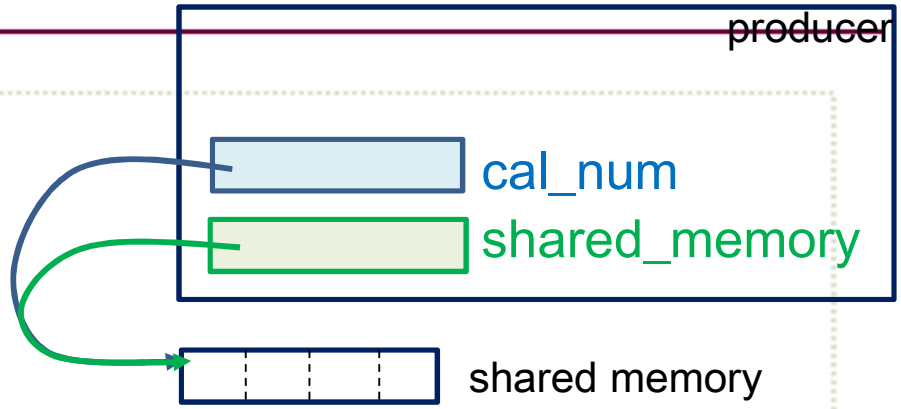

- 공유 메모리 공간을 만들고
- 이를 각 프로세스가 첨부(attach)하는 방식으로 작동

# 공유 메모리 생성과 첨부

```
#include <sys/types.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmid;
void *shared_memory = NULL;
int *num;
shmid = shmget(1234, sizeof(int), 0666 | IPC_CREAT);

shared_memory = shmat(shmid, NULL, 0);
num = (int *)shared_memory;
```

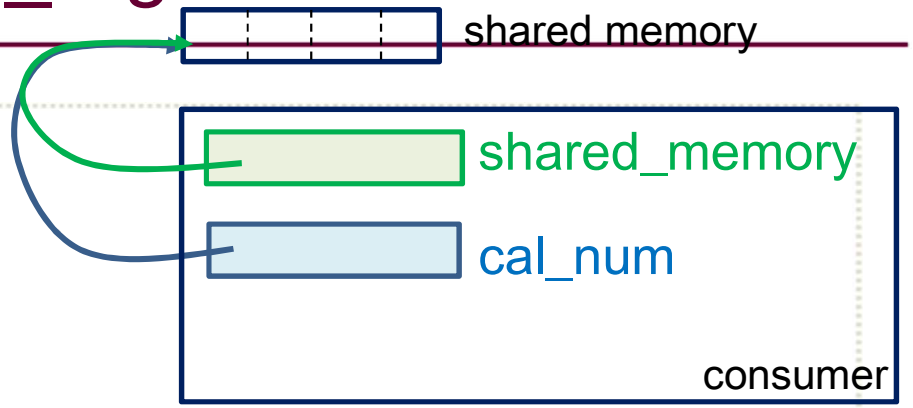
- key : 공유 메모리를 가리키는 Key,
- size : 공유 메모리의 크기
- shmflg : 공유메모리를 제어하기 위한 flag

# 예제 프로그램 shm_producer_org.c



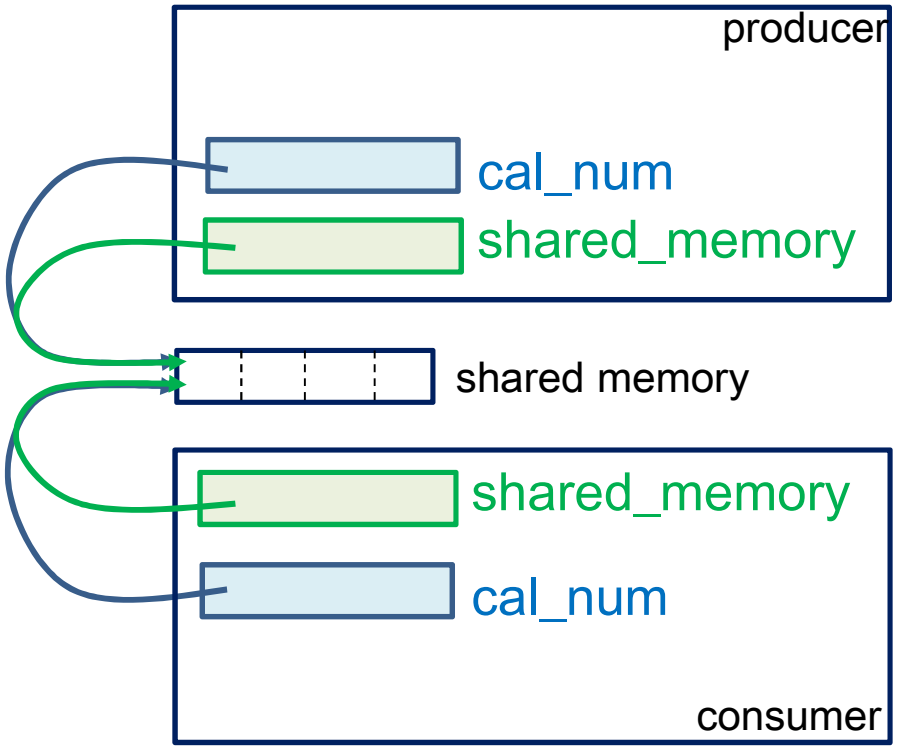

```
int main(int argc, char **argv)
{       int shmid;
        int *cal_num;
        void *shared_memory = NULL;

        shmid = shmget((key_t)1234, sizeof(int), 0666| IPC_CREAT);
        shared_memory = shmat(shmid, NULL, 0);
        cal_num = (int *)shared_memory;

        *cal_num  = 0;
        while(1)
        {
                *cal_num  = *cal_num + 2;       //  begin of critical section
               sleep(1);                        // end of critical section
        }
}
```

# 예제 프로그램 shm_consumer_org.c




```
int main(int argc, char **argv)
{       int shmid;
        int *cal_num;
        void *shared_memory = NULL;

        shmid = shmget((key_t)1234, sizeof(int), 0);
        shared_memory = shmat(shmid, NULL, 0);
        cal_num = (int *)shared_memory;

        while(1)
        {
                sleep(1);
                                                        //  begin of critical section
                printf(" Read Data : %d₩n", *cal_num); // end of critical section
        }
}
```

# shm_producer_org.c, shm_consumer_org.c



# shm_producer_org.c, shm_consumer_org.c

Lab

```
osnw00000000@osnw00000000-osnw: ~/week10
osnw00000000@osnw00000000-osnw:~/week10$ ./shm_producer_org
^C
```

```
osnw00000000@osnw00000000-osnw: ~/week10
osnw00000000@osnw00000000-osnw:~/week10$ ./shm_consumer_org
Read Data : 8
Read Data : 10
Read Data : 12
Read Data : 14
Read Data : 16
Read Data : 18
Read Data : 20
Read Data : 22
Read Data : 24
Read Data : 26
```

```
osnw00000000@osnw00000000-osnw: ~/week10
osnw00000000@osnw00000000-osnw:~/week10$ ./shm_consumer_org
Read Data : 24
Read Data : 26
Read Data : 28
Read Data : 30
Read Data : 32
```

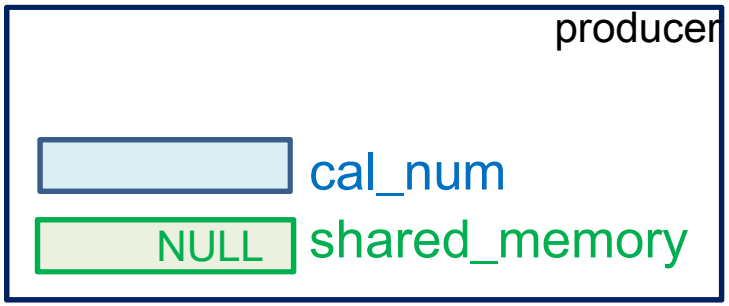
```
osnw00000000@osnw00000000-osnw: ~
osnw00000000@osnw00000000-osnw:~$ ps -ef | grep shm
osnw000+  111722  111633  0 02:21 pts/1    00:00:00 ./shm_producer_org
osnw000+  111724  111576  0 02:21 pts/0    00:00:00 ./shm_consumer_org
osnw000+  111726  111691 10 02:22 pts/2    00:00:00 grep --color=auto shm
osnw00000000@osnw00000000-osnw:~$
```

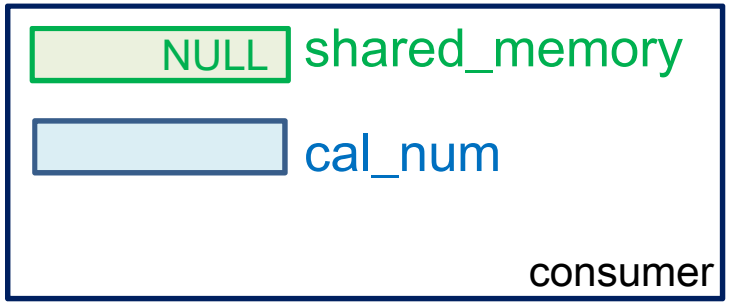
# ipcs, ipcrm

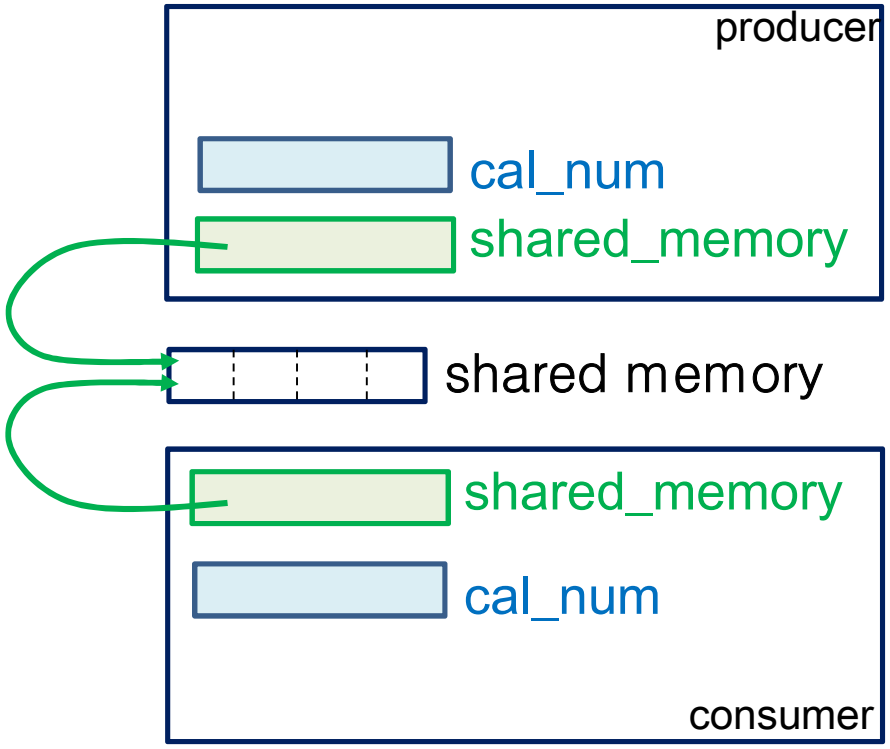
Lab

```
osnw00000000@osnw00000000-osnw: ~
osnw00000000@osnw00000000-osnw:~$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000004d2 0          osnw000000 666        4          0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

osnw00000000@osnw00000000-osnw:~$ ipcrm shm 0
resource(s) deleted
osnw00000000@osnw00000000-osnw:~$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

# shm_producer_race.c, shm_consumer_race.c



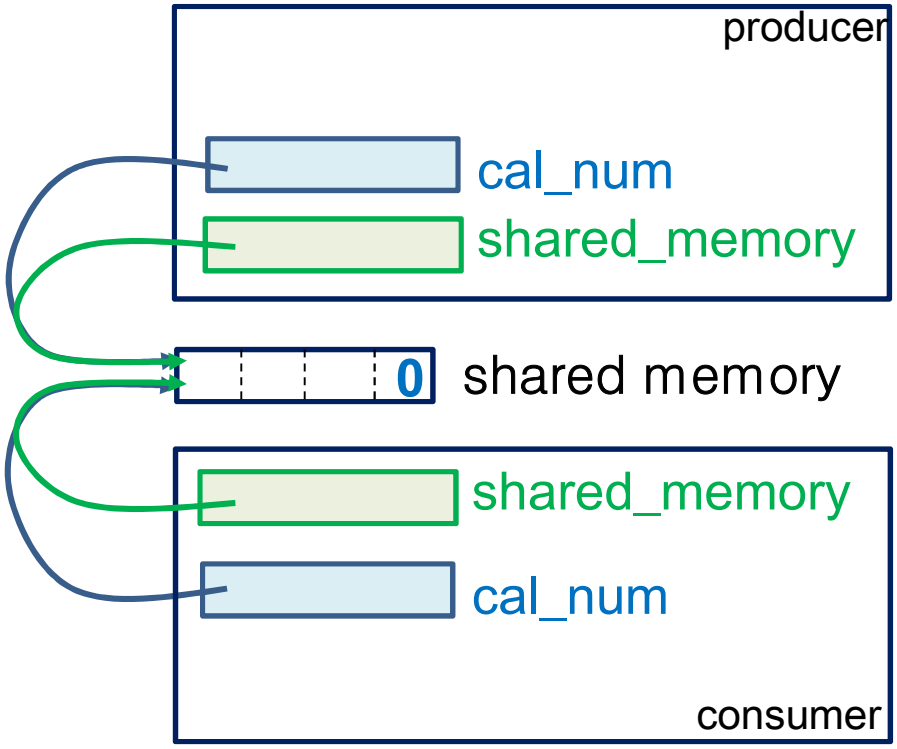




```
int *cal_num;
void *shared_memory = NULL;
```

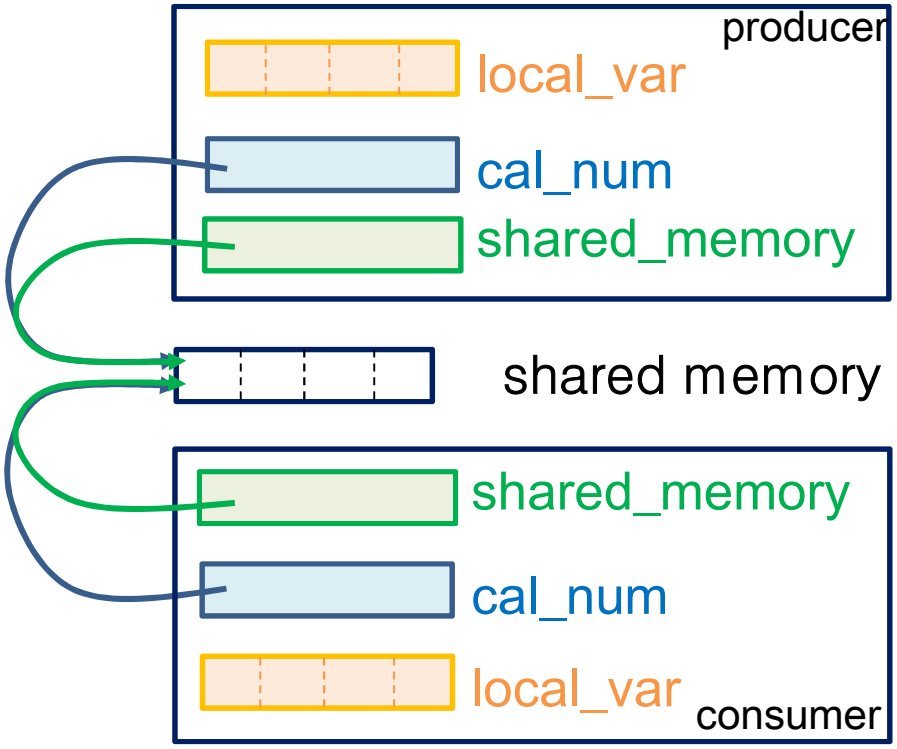
```
// producer
shmid = shmget((key_t)1234, sizeof(int),
0666|IPC_CREAT);
shared_memory = shmat(shmid, NULL, 0);

// consumer
shmid = shmget((key_t)1234, sizeof(int), 0);
shared_memory = shmat(shmid, NULL, 0);
```

# shm_producer_race.c, shm_consumer_race.c



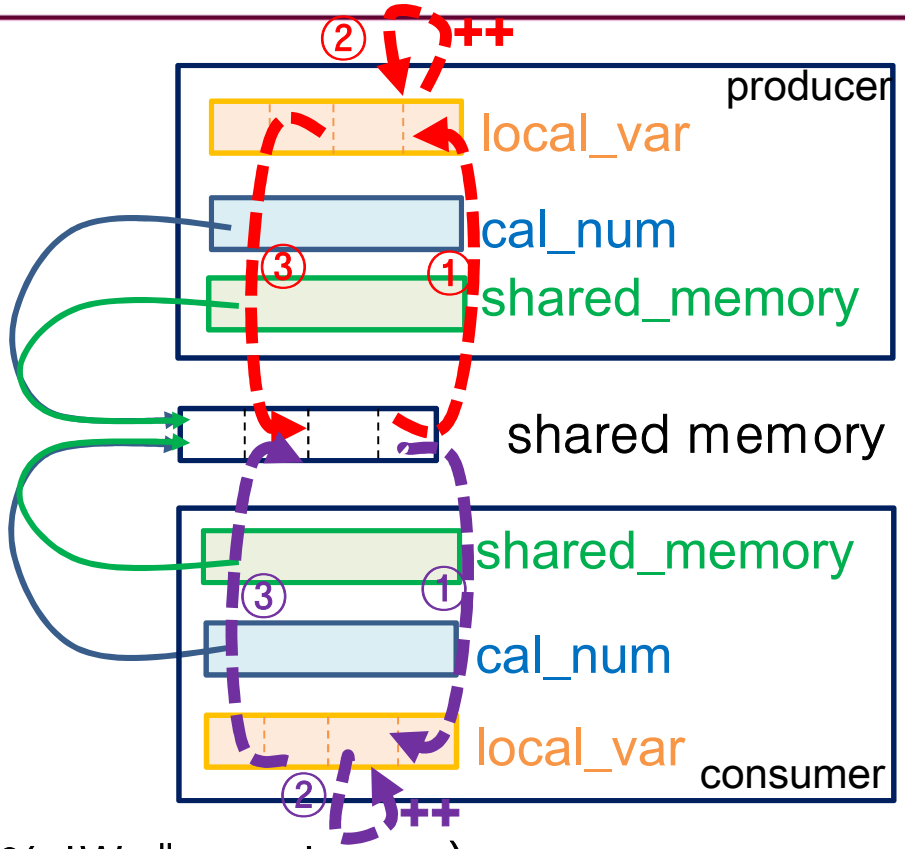

```
cal_num = (int *)shared_memory;
*cal_num = 0;
```

# shm_producer_race.c, shm_consumer_race.c




```
while(1)
{
        int local_var;
        local_var = *cal_num;
        local_var ++;
        sleep(1);
        *cal_num = local_var;
        printf("Producer/Conumer : %d₩n", *cal_num);
}
// while(1)
// {
//      (*cal_num)++;
//      sleep(1);
//      printf("Producer/Conumer : %d₩n", *cal_num);
// }
```

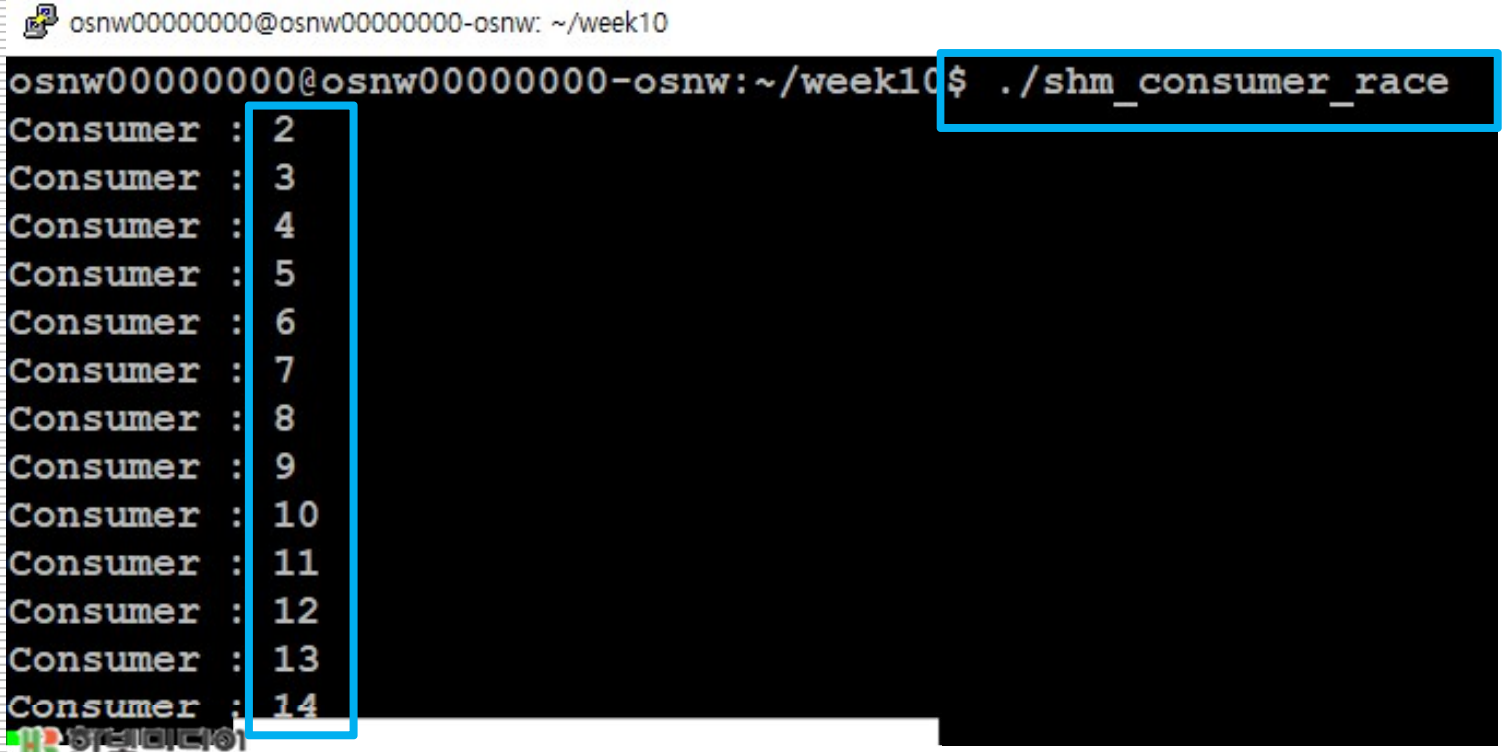
# shm_producer_race.c, shm_consumer_race.c




```
while(1)
{
        int local_var;
        local_var = *cal_num; ①
        local_var ++;          ②
        sleep(1);
        *cal_num = local_var; ③
        printf("Producer/Conumer : %d₩n", *cal_num);
}
// while(1)
// {
//      (*cal_num)++;
//      sleep(1);
//      printf("Producer/Conumer : %d₩n", *cal_num);
// }
```

# shm_producer_race.c, shm_consumer_race.c

Lab

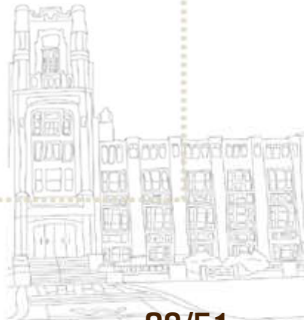



# 세마포어(Semaphore)

- 여러 프로세스가 하나의 자원(예, 공유메모리) 에 접근할 경우 **race condition**이 발생하므로 접근 제어 필요

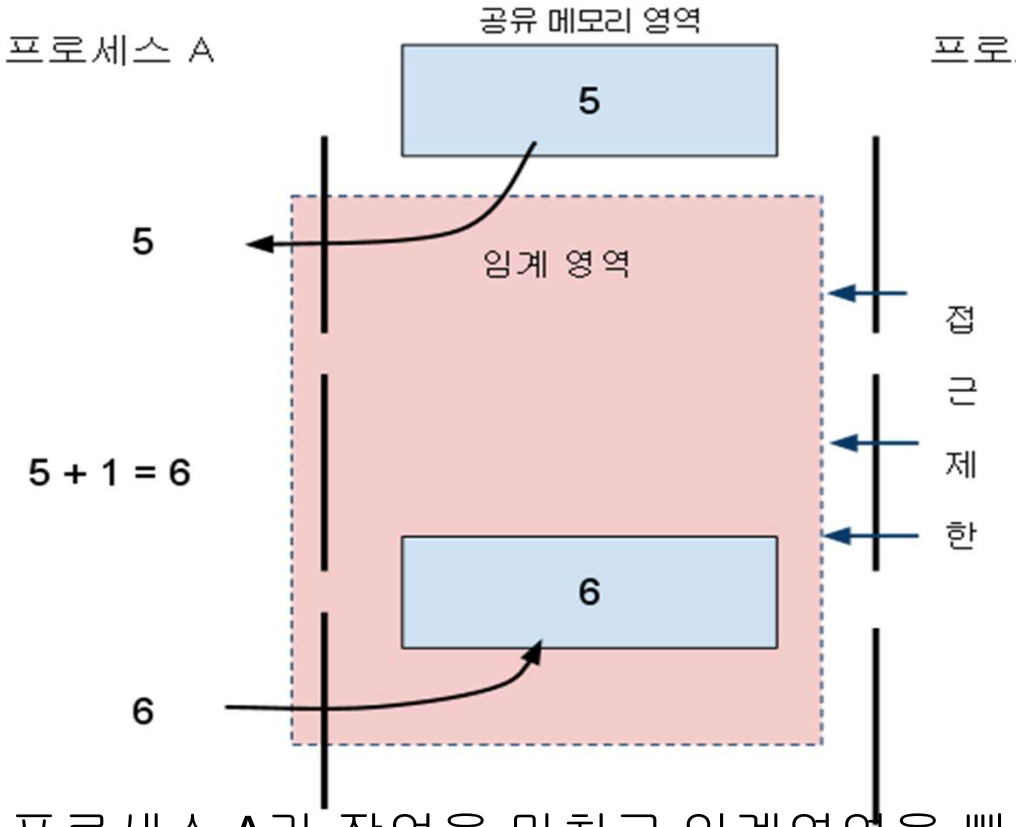
  ex) 공유메모리 : 5 (저장)
  - 프로세스 A : read (5)
  - 프로세스 B : read (5)
  - 프로세스 A : +1 연산 (6)
  - 프로세스 B : +1 연산 (6)

- 세마포어 → 공유 자원에 대한 접근 제어 메커니즘을 제공



# 세마포어

- 임계영역(Critical Section) 설정
  - 한번에 하나의 프로세스만 진입할 수 있는 영역
- 임계영역을 접근하려면 "세마포어"를 획득해야 함




- 프로세스 A가 작업을 마치고 임계영역을 빠져나올 때까지. 프로세스 B는 임계영역에 진입할 수 없으며 대기 상태에 놓임

# 세마포어 만들기

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

- **nsems : 세마포어의 개수**
  - 임계영역 설정에는 보통 1을 사용.
- semflg : 세마포어 동작제어를 위한 flag

# 세마포어 얻기

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsop);

  struct sembuf
  {
    short sem_num;
    short sem_op;
    short sem_flg;
  }
```

- semop 함수를 이용 세마포어를 얻거나 돌려줌
  - sembuf의 멤버 변수값으로 제어

# 세마포어 얻기과 돌려주기

- 세마포어를 얻을 경우, i.e., **P(sops), wait(sops)**

```
struct sembuf sops;
sops.sem_num = 0;
sops.sem_op = -1;
sops.sem_flg = SEM_UNDO;
semop(semid, &sops, 1);
```

- 세마포어를 돌려줄 경우, i.e., **V(sops), signal(sops)**

```
struct sembuf sops;
sops.sem_num = 0;
sops.sem_op = 1;
sops.sem_flg = SEM_UNDO;
semop(semid, &sops, 1);
```

# 예제 프로그램 shm_producer_sem.c

```
union semun
{               int val;
};
int main(int argc, char **argv)
{               int shmid;
                int semid;
                int *cal_num;
                void *shared_memory = NULL;
                union semun sem_union;
                struct sembuf semopen = {0, -1, SEM_UNDO};
                struct sembuf semclose = {0, 1, SEM_UNDO};
                shmid = shmget((key_t)1234, sizeof(int), 0666|IPC_CREAT);
                semid = semget((key_t)3477, 1, IPC_CREAT|0666);
                shared_memory = shmat(shmid, NULL, 0);
                cal_num = (int *)shared_memory;
                sem_union.val = 1;
                semctl( semid, 0, SETVAL, sem_union );
                while(1)
                {
                        int local_var = 0;
                        semop(semid, &semopen, 1);   // P(), wait()
                        local_var = *cal_num + 1;                       //  begin of critical section
                        sleep(1);
                        *cal_num = local_var;
                        printf("producer semaphore : %d\n", *cal_num);  // end of critical section
                        semop(semid, &semclose, 1);   // V(), signal()
                }
}
```

# 예제 프로그램 shm_consumer_sem.c

```
int main(int argc, char **argv)
{
        int shmid;
        int semid;
        int *cal_num;
        void * shared_memory = NULL;
        struct sembuf semopen = {0, -1, SEM_UNDO};
        struct sembuf semclose = {0, 1, SEM_UNDO};
        shmid = shmget((key_t)1234, sizeof(int), 0666);
        semid = semget((key_t)3477, 0, 0666);
        shared_memory = shmat(shmid, NULL, 0);
        cal_num = (int *)shared_memory;
        while(1)
        {
                int local_var=0;
                semop(semid, &semopen, 1);              // P(), wait()
                local_var = *cal_num+1;                 //  begin of critical section
                sleep(2);
                *cal_num = local_var;
                printf("consumer semaphore : %d\n", *cal_num);      // end of critical section
                semop(semid, &semclose, 1);             // V(), signal()
        }
}
```

# shm_producer_sem.c, shm_consumer_sem.c

Lab

```
osnw2021@student04--10:~/lab07$ ./shm_producer_sem
Producer semaphore : 1
Producer semaphore : 2
Producer semaphore : 3
Producer semaphore : 4
Producer semaphore : 6
Producer semaphore : 8
Producer semaphore : 10
Producer semaphore : 12
Producer semaphore : 14
```

```
osnw2021@student04--10:~/lab07$ ./shm_consumer_sem
Consumer semaphore : 5
Consumer semaphore : 7
Consumer semaphore : 9
Consumer semaphore : 11
Consumer semaphore : 13
Consumer semaphore : 15
Consumer semaphore : 17
```

```
osnw2021@student04--10:~/lab07$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 98304      lightdm    600        524288     2          dest
0x00000000 131073     lightdm    600        33554432   2          dest
0x000004d2 229378     osnw2021   666        4          0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000d95 0          osnw2021   666        1
```

# Thank You !

뇌를 자극하는 TCP/IP 소켓 프로그래밍