Embedded System Software

fork,IPC(Inter Process Communication)

Sungyong Park, Ph.D

<parksy@sogang.ac.kr>
Dept. of Computer Science and Engineering
Sogang University, Seoul, KOREA

fork

- 기존 프로세스가 새 프로세스를 생성할 때 사용

```
#include <unistd.h>

pid_t fork(void);
```

- **Return value**
  - **child process : 0**
  - **parent process : child 프로세스의 ID**
  - **failure : -1**
- 자식 프로세스와 부모 프로세스 모두 **fork** 호출 이후의 명령을 계속 실행

fork

- 자식은 부모의 복사본
  - 자식은 부모의 자료 구역, 힙, 스택의 복사본을 가짐
    이는 자식을 위한 복사본일 뿐, 부모와 자식이 동일한 메모리 영역을 공유하는 것은 아님
    (단, **text segment** 영역은 공유하지 않음)
  - 현재의 구현들이 부모의 자료,스택,힙 전체를 무조건 복사하지 않음
- **copy-on-write**
  - 자식 생성시에 자료구역,스택,힙 등을 부모와 자식이 공유
  - 이후 자식과 부모 중 하나라도 이 영역을 수정하면 필요한 만큼의 조각(보통 가상메모리의 한 **page**)만을 복사

fork

```
int glob = 6; // 외부 변수
int main(void) {
        pid_t pid;
        int var = 88; // 자동변수

        pid = fork();
        if(pid == -1) { printf("error₩n"); exit(0); }
        else if (pid == 0)  {
                // child
                glob++;  var++;              // 변수 수정
                printf("child : getpid() %d, getppid() %d₩n",getpid(),getppid());
                printf("child : pid %d₩n", pid);
                printf("child : glob %d var %d₩n", glob, var);
        } else {
                // parent
                printf("parent : getpid() %d₩n",getpid());
                printf("parent : pid %d₩n", pid);
                printf("parent : glob %d var %d₩n", glob, var);
        }
}
```

```
parent : getpid() 4215
parent : pid 4216
parent : glob 6 var 88
child : getpid() 4216, getppid() 4215
child : pid 0
child : glob 7 var 89
```

IPC

- IPC (Interprocess Communication)
  - Message passing 방식
  - Synchronization 방식
- 고급 IPC 설비
  - Message queue
  - Semaphore
  - Shared memory
- IPC 설비 키 (keys)
  - UNIX 시스템의 IPC객체를 식별하는데 사용하는 수
  - Data type : key_t (<sys/types.h>에서 정의)

IPC

- **IPC 생성 (IPC *get* 연산들)**
  - **IPC 객체를 생성하거나 기존의 IPC 객체에 접근**
  - **연산의 결과인 IPC 식별자는 다른 IPC 루틴 호출에 사용된다.**

    ex) mqid = msgget((key_t) 0100, 0644 | IPC_CREAT | IPC_EXCL);
  - ***msgget, semget, shmget***
- **다른 IPC 연산**
  - **제어 연산 (IPC control 연산들)**
    - 상태 정보를 얻거나 제어값을 지정
    - ***msgctl, semctl, shmctl***
  - **기타 각 설비 별 구체적인 연산들**

IPC

- **IPC 설비의 상태 자료구조**
  - **IPC 객체 생성시 시스템은 IPC 설비 상태 구조를 만들어 해당 객체에 해당하는 각종 관리 정보들을 저장**
  - **IPC 유형별 저장 정보가 다르나 공통적으로 허가 구조를 포함**

```
struct  ipc_perm {
        uid_t  uid;          /*owner's effective user id*/
        gid_t  gid;          /*owner's effective group id*/
        uid_t  cuid;         /*creator's effective user id*/
        gid_t  cgid;         /*creator's effective group id*/
        mode_t  mode;        /*access modes*/
        ulong  seq;          /*slot usage sequence number*/
        key-_t  key;         /*key*/
};
```

  - seq를 제외한 모든 필드는 IPC 생성시 초기화
  - 추후 IPC 제어 연산들을 통해 uid, gid, mode값은 변경 가능

Message Queue

- Message Queue 개요
- msgget
- msgsnd
- msgrcv
- msgctl

Message Queue 개요

- 메시지 큐(Message queue)
  - 프로세스간 메시지 큐를 통해 메시지 전달
  - ***msgget*** 함수로 메시지 큐를 생성 하거나 접근
  - 적당한 큐 허가를 가진 프로세스가 ***msgsnd*** 로 메시지를 큐에 추가
  - 다른 적당한 큐 허가를 가진 프로세스는 ***msgrcv*** 를 사용해 큐로부터 메시지를 읽고, 메시지는 큐에서 제거
  - ***msgctl*** 연산을 사용하여 메시지 큐의 상태정보를 읽거나 변경하고, 시스템 상에서 큐를 제거
- 메시지
  - 문자열이나 바이트 열로 구성되어 프로세스 사이에 전달

```
struct  mymsg {
        long  mtype;                    /*message type*/
        char  mtext[SOMEVALUE]    /*messgae text*/
};
```

Message Queue 개요

메시지 큐 상태 자료 구조 (msqid_ds)

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* An instance of the ipc_perm structure */
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ...
};
```

- 허가구조, 큐 내의 메시지 포인터, 큐 수정 시간 등

msgget 함수

```
#include <sys/msg.h>

int msgget (key_t key, int permflags);
returns: message queue ID if OK, -1 on error
```

- **key: 메시지큐를 시스템에서 식별시키는 단순한 숫자**
- **permflags (<sys/ipc.h>에 정의)**

IPC_CREAT	key에 해당하는 메시지 큐가 존재하지 않는 경우 msgget이 이를 생성하도록 지시한다. 파일에 비유하면 이 플래그는 msgget 호출이 마치 파일에서의 creat 호출과 같이 행동하게 하지만, 기존의 메시지 큐가 있는 경우 덮어 쓰지 않는다. IPC_CREAT 플래그가 설정되지 않으면, msgget은 그 키를 가진 메시지 큐가 존재하는 경우에만 메시지 큐 식별자를 리턴한다.
IPC_EXCL	IPC_CREAT와 같이 설정하여 단지 하나의 메시지 큐를 생성하기 위한 것이다. 따라서 key에 대한 큐가 이미 존재하는 경우에는 msgget은 실패하고 -1을 돌려준다. 이때 오류 변수 errno에는 EEXIST 값이 저장된다.

ex) mqid = msgget((key_t) 0100, 0666 | IPC_CREAT | IPC_EXCL)

msgctl 함수

```
#include <sys/msg.h>

int msgctl (int mqid, int command, struct msqid_ds *msq_stat);
returns: 0 if OK, -1 on error
```

- **mqid: 유효한 메시지 큐식별자**
- **command : 수행될 작업 (<sys/ipc.h>에 정의)**

IPC_STAT	메시지 큐의 상태정보를 가진 msqid_ds 구조를 msg_stat에 넣도록 시스템에 지시한다.
IPC_SET	msq_stat에 있는 정보에 따라 메시지 큐에 대한 제어변수들의 값을 지정하기 위해 사용되며 다음과 같은 사항을 변경한다 : msq_stat.msg_perm.uid, msq_stat.msg_perm.gid, msq_stat.msg_perm.mode, msq_stat.msg_qbytes IPC_SET을 수행 하려면 사용자가 수퍼유저이거나 msq_stat.msg_perm.uid가 가리키는 메시지 큐의 현재 소유자라야 한다. msq_stat.msg_qbytes 값은 한번에 큐에 존재할 수 있는 문자의 최대값으로 이 또한 수퍼유저에 의해서만 변경 가능하다.
IPC_RMID	이는 메시지 큐를 시스템에서 삭제하게 한다. 이것도 역시 수퍼유저나 큐의 소유자만이 실행할 수 있다. 만약 IPC_RMID로 command가 지정되면 msq_stat은 NULL로 지정된다.

- **msg_stat : msgid_ds structure의 주소**

msgctl 예제

```
#include <stdio.h>
#include <sys/msg.h>

main ()
{
        key_t key;
        int msgid;

        key = ftok ("keyfile", 1);
        msgid = msgget (key, IPC_CREAT|0644);
        if (msgid == -1) {
                perror ("msgget");
                exit (1);
        }

        printf ("Before IPC_RMID₩n");
        system ("ipcs -q");
        msgctl (msgid, IPC_RMID, (struct msqid_ds *) NULL);
        printf ("After IPC_RMID₩n");
        system ("ipcs -q");
}
```

msgsnd 함수

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int mqid, const void *message, size_t size, int flags);
returns: 0 if OK, -1 on error
```

- **mqid: msgget에서 받은 메시지 큐 식별자**
- **message: 보낼 메시지를 저장하고 있는 사용자 정의 메시지 자료구조 mymsg를 가리키는 포인터**
- **size: 실제 전달되는 메시지의 길이 (0 ~ SOMEVALUE 또는 시스템에서 정의한 최대값 중 작은 값)**
- **flag: IPC_NOWAIT**
  - **메시지를 보내기 위한 충분한 시스템 자원이 없을 경우 수면상태 (sleep)**
  - **flag 설정 시 메시지가 전달될 수 없을 경우 즉시 복귀. (-1값 리턴, errno는 EAGAIN)**
- **해당 메시지 큐에 write 접근 권한이 없는 경우, errno 값 EACCESS를 가지고 호출 실패**

msgrcv 함수

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int mqid, void *message, size_t size, long msg_type, int flags);
returns: the number of bytes of data read into buffer if OK, -1 on error
```

- **mqid : 유효한 메시지 큐 식별자**
- **message: 받아들인 메시지를 저장**
- **size: 이 구조에 저장할 수 있는 최대 길이**
- **meg_type: 메시지의 mtype 필드를 지정하여 읽을 메시지를 선택**
  - **0 : 첫번째 메시지**
  - **양수 : 그 값을 가지는 첫번째 메시지**
  - **음수: 그 절대값 보다 작거나 같은 것 중 최소값을 갖는 메시지**

  예) mtype : 999, 5, 1 인 경우

  0 → 999 메시지

  5 → 5 메시지

  -999 → 1 메시지, (두번째 호출시 5, 세번째 호출시 999)
- **flags: IPC_NOWAIT, MSG_NOERROR**
  - **큐에 메시지가 없으면 대기하나, IPC_NOWAIT가 설정되면 즉각 복귀**
  - **메시지의 내용이 size보다 길면 호출실패, MSG_NOERROR가 설정되면 초과분 절단**

msgsnd & msgrcv 예제

```
struct mymsgbuf {
  long mtype;
  char mtext[80];
}

main(){
  key_t key;
  int msgid;
  struct mymsgbuf mesg;

  key = ftok ("keyfile", 1);
  msgid = msgget (key, IPC_CREAT|0666);
  if (msgid == -1) {
       perror ("msgget");
       exit (1);
  }

  mesg.mtype = 1;
  strcpy (mesg.mtext, "test messgae₩n");
  msgsnd (msgid, (void *)&mesg, 80, 0);
}
```

```
struct mymsgbuf {
  long mtype;
  char mtext[80];
}

main() {
  key_t key;
  int msgid;
  struct mymsgbuf mesg;

  key = ftok ("keyfile", 1);
  msgid = msgget (key, 0);

  msgrcv (msgid, &mesg, 80, 0, 0);
  printf ("Received Msg = %s", mesg.mtext);
}
```

Semaphore

- Concept - Semaphore
- semget
- semctl
- semop

Concept - Semaphore

- 이론적 개념
  - 프로세스 동기화 문제에 대한 해결책으로 네덜란드 이론가 **E.W.Dijkstra**에 의해 제안
  - 다음과 같은 연산이 허용된 변수
    - p(sem) 또는 wait(sem)

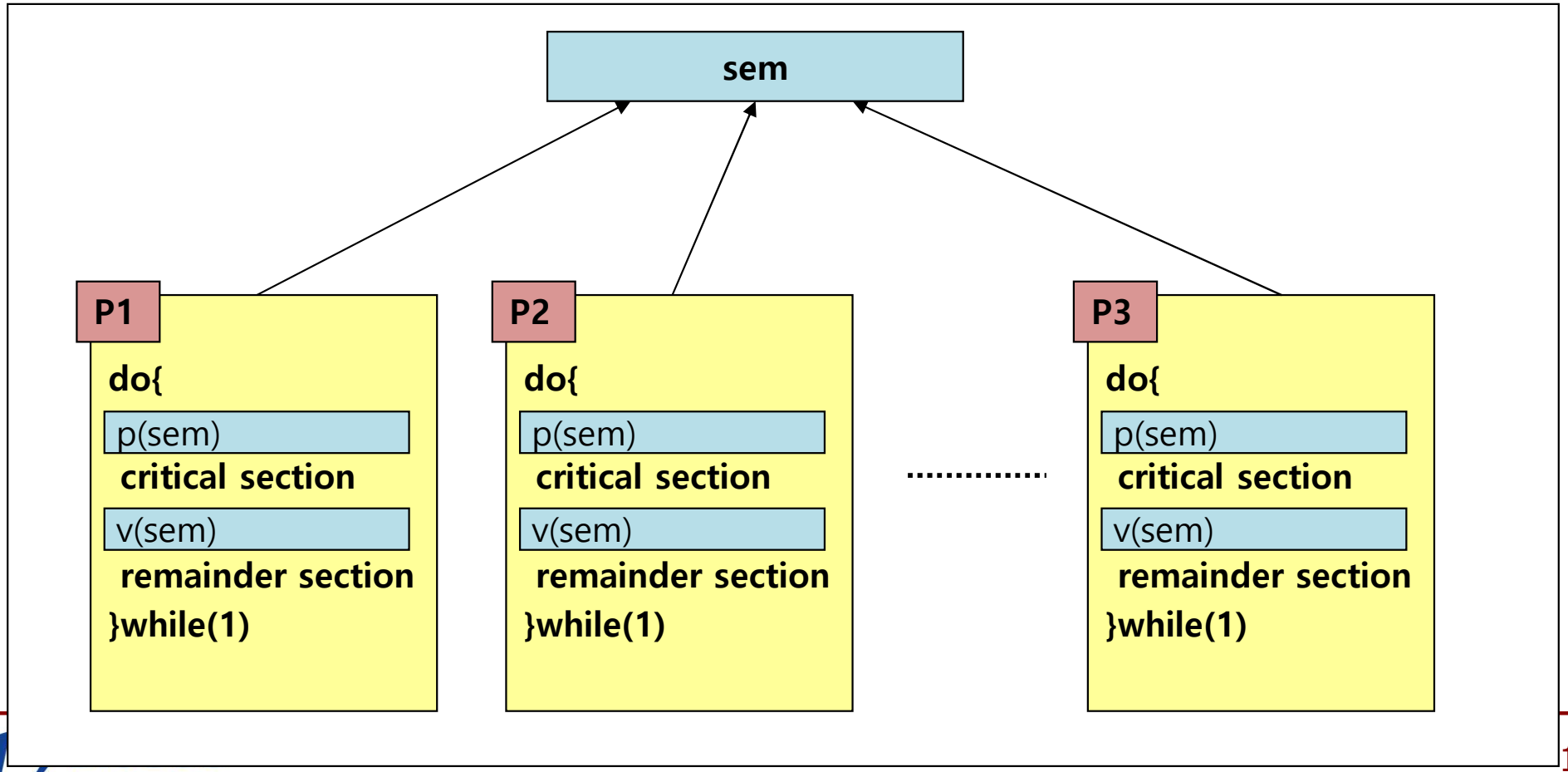
```
if (sem != 0)
      decrement sem by one
else
      wait until sem becomes non-sero, then decrement
```

    - v(sem) 또는 signal(Sem)

```
increment sem by one
if (queue of waiting processes not empty)
      restart first process in wait queue
```

Concept - Semaphore

- **Critical Section**
  - 한번에 하나의 프로세서만 접근해야 하는 공유자원



Concept - Semaphore

- 세미포 상태 자료구조 (semid_ds)

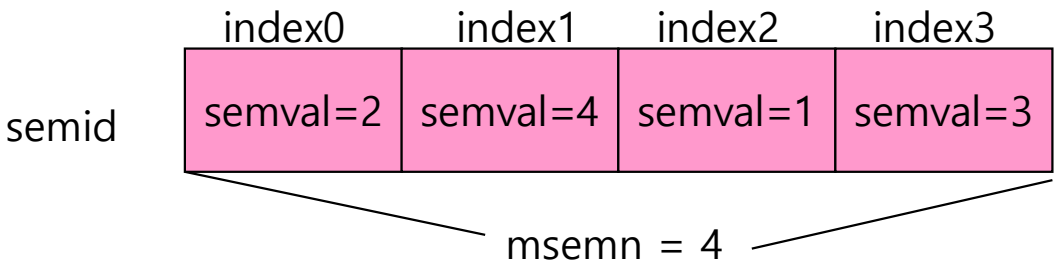
```
struct semid_ds
{
   struct  ipc_perm  sem_perm;          /*operation permission struct*/
   time_t  sem_otime;                   /*time of last semop()*/
   time_t  sem_ctime;                   /*time of the last semctl() operation that
                                              changed a semaphore in the set*/
   unsigned long  int  sem_nsems;       /*# of semaphores in set*/
   struct sem  *sem_base;               /* A pointer to the first semaphore */
};
```

semget 함수

```
#include <sys/sem.h>
int semget ( key_t key, int nsems, int semflags );
returns: semaphore ID if OK, -1 on error
```

- **key : 세마포 식별을 위한 키 값**
- **nsems: 세마포 집합의 크기**
  - 세마포어 집합을 생성시 반드시 지정해야함
  - 기존 집합을 참조시에는 **0**을 지정해도 됨
- **semflags**

IPC_CREAT	만약 커널에 해당 key 값으로 존재하는 세마포가 없다면, 새로 생성 한다.
IPC_EXCL	IPC_CREAT와 함께 사용하며, 해당 key 값으로 세마포가 이미 존재한다면 실패 값을 리턴한다.



semget 함수

세마포 자료구조

```
struct sem
{
   ushort_t  semval;          /*semaphore value, nonnegative*/
   short  sempid              /*PID of last successful semop(), SETVAL, SETALL*/
   ushort_t  semncnt          /*# awaiting semval > current val */
   ushort_t  semzcnt          /*# awaiting semval = 0*/
};
```

semget 함수 예제

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int  main()
{
   int semid;
   semid = semget ((key_t)12345, 1, 0666 | IPC_CREAT);
}
```

semctl 함수

```
#include <sys/sem.h>
int semctl (int semid, int sem_num, int command, union semun ctl_arg);
returns: nonnegative value if OK, -1 on error
```

- **semid : semget에서 return된 유효한 세마포 식별자**
- **command**

표준 IPC 기능	
IPC_STAT	세마포 상태값을 얻어오기 위해 사용되며, 상태값은 ctl_arg에 저장된다.
IPC_SET	semid_ds의 ipc_perm 정보를 변경함으로써 세마포에 대한 권한을 변경한다.
IPC_RMID	세마포를 삭제하기 위해서 사용한다.
단일 세마포 연산	
GETVAL	세마포의 값 semval을 돌려준다
SETVAL	세마포 값을 ctl_arg.val로 지정한다
GETPID	sempid의 값을 돌려준다
GETNCNT	semncnt를 돌려준다
GETZCNT	semzcnt를 돌려준다
전체 세마포 연산	
GETALL	모든 semvals의 값을 ctl_arg.array에 저장한다
SETALL	ctl_arg.array의 값을 사용하여 모든 semvals 값을 지정한다

semctl 함수

- **sem_num:** 단일세마포 연산을 위한 특정 세마포 식별

- **ctl_arg**

```
union semun {
int val;
struct semid_ds *buf;
unsigned short *array;
};
```

  - 유니온의 구성원은 **semctl** 유형에 따라 다른 유형의 값
    예) **command**가 **SETVAL**이면 **ctl_arg.val**로 **semval**값 지정

- **semctl**을 통해 세마포 초기값 지정

semop 함수

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *op_array, size_t num_ops);
returns: 0 if OK, -1 on error
```

- **semid : 세마포 식별자**
- **op_array: sembuf 구조의 배열**
  - **sembuf 구조 : 한 세마포에 대해 수행할 연산에 대한 명세 저장**
- **num_ops: 배열내의 sembuf 구조의 수**

- **semop 함수가 세마포 집합에 대해 수행하는 일련의 연산들은 모두 원자화되어야 한다 (atomic operation)**

semop 함수

sembuf 구조 : 한 세마포에 대해 수행할 연산에 대한 명세 저장

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

- **sem_num: 집합 내의 세마포 인덱스 저장**
- **sem_op: semop가 수행해야하는 기능을 정수로 표현**
  - **sem_op < 0: p() 연산의 일반화된 형태.**
    - 해당 세마포의 semval값이 충분히 크면 즉시 감소
    - 아니면 samval 값이 충분히 커질때 까지 기다린다.
    - sem_flag에 IPC_NOWAIT 가 지정되어 있으면 즉시 -1을 return (errno=EAGAIN)
  - **sem_op > 0: v() 연산**
    - sem_op값을 semval에 더하고, 해당 세마포 값이 증가하길 기다리는 프로세스들을 깨운다.
  - **sem_op = 0**
    - semval 값이 0이 될 때까지 기다린다.
    - sem_flag에 IPC_NOWAIT가 지정되면 있으면 즉시 -1을 return (errno=EAGAIN)
- **sem_flag : IPC_NOWAIT, SEM_UNDO**
  - **SEM_UNDO: 프로세스가 세마포를 돌려주지 않고 종료할 경우 커널에서 조정**

```
 // The calling  program must define this union as follows
typedef union{
   int val;
   struct semid_ds *buf;
   ushort *array;
} semun;

int initsem (key_t semkey) {
            int status = 0, semid;
            if ((semid = semget (semkey, 1, IPC_CREAT | IPC_EXCL|0660)) == -1) {
                        // 이미 존재할 경우
                        if (errno == EEXIST)  semid = semget (semkey, 1, 0);
            }
            else {
                        semun arg; arg.val = 1;
                        status = semctl (semid, 0, SETVAL, arg);
            }
            if (semid == -1 || status == -1) {
                        return (-1);
            }
            return (semid);
}
```

```
int p (int semid) {
        // p 연산
        struct sembuf p_buf;

        p_buf.sem_num = 0;
        p_buf.sem_op = -1;
        p_buf.sem_flg = SEM_UNDO;

        if (semop (semid, &p_buf, 1) == -1) exit (1);
        return (0);
}
int v(int semid) {
        // v 연산
        struct sembuf v_buf;

        v_buf.sem_num = 0;
        v_buf.sem_op = 1;
        v_buf.sem_flg = SEM_UNDO;

        if (semop (semid, &v_buf, 1) == -1) exit (1);
        return (0);
}
```

```
void handlesem (key_t skey) {
        int semid;
        pid_t pid = getpid();
        if ( (semid = initsem (skey)) < 0) exit (0);
        printf ("process %d before critical section₩n", pid);

        p (semid);
        printf ("process %d in critical section₩n", pid);
        /* 실제로 뭔가 흥미로운 일을 한다 */
        sleep (10);
        printf ("process %d leaving critical section₩n", pid);

        v (semid);
        printf ("process %d exiting₩n", pid);
        exit (0);
}

main () {
        key_t semkey = 0x200;
        for (i = 0; i < 3; i++)
                if (fork () == 0) handlesem (semkey);
}
```

```
process 4794 before critical section
process 4794 in critical section
process 4795 before critical section
process 4796 before critical section
process 4794 leaving critical section
process 4794 exiting
process 4795 in critical section
process 4795 leaving critical section
process 4795 exiting
process 4796 in critical section
process 4796 leaving critical section
process 4796 exiting
```

Shared Memory

- Concept - shared memory
- shmget
- shmat & shmdt
- shmctl

Concept – Shared Memory

- 공유 메모리
  - 여러 프로세스에서 동시에 접근 가능한 영역
  - 여러 IPC중 가장 빠른 실행속도 (데이터 복사 불필요)
  - 한번에 하나의 프로세스만 접근하도록 보장 필요
  - ***shmget*** 호출로 공유 메모리 영역 생성
  - 프로세스는 ***shmat***을 사용하여 자신을 그 메모리에 부착
  - 그 메모리가 더 이상 필요 없게 되면 ***shmdt***로 자신을 떼어낸다.
- 공유 메모리 생성 및 관리
  - 커널에 의해 커널 영역 메모리 할당
  - 커널에 의해 관리되어지므로 운영체제를 리부팅 하거나 직접 공유메모리 공간을 삭제시키지 않는 한 유지됨
  - 공유메모리 상태 자료구조 (shmid_ds)를 통하여 공유 메모리 관리

Concept – Shared Memory

- 공유 메모리 상태 자료구조 (shmid_ds)

```
struct shmid_ds
{
struct      ipc_perm shm_perm; // 퍼미션
int         shm_segsz;         // 메모리 공간의 크기
time_t      shm_atime;         // 마지막 attach 시간
time_t      shm_dtime;         // 마지막 detach 시간
time_t      shm_ctime;         // 마지막 변경 시간
unsigned short shm_cpid;       // 생성프로세스의 pid
unsigned short shm_lpid;       // shmop()를 마지막으로 호출한 프로세스의 pid
short       shm_nattch;        // 현재 부착 횟수
};
```

shmget 함수

```
#include <sys/shm.h>
int shmget (key_t key, size_t size, int permflags);
returns: shared memory ID if OK, -1 on error
```

- k
- size : 필요한 공유메모리 영역의 최소 크기를 바이트 단위로 지정
- permflags: IPC_CREAT, IPC_EXCL
- 예제

```
main() {
    key_t key;
    int shmid;
    key = ftok ("myfile", 1);
    shmid = shmget (key, 1024, IPC_CREAT|0644);
    if(shmid == -1) {
        perror ("shmget");
        exit (1);
    }
 }
```

shmat 함수와 shmdt 함수

```
#include <sys/shm.h>
int shmat (int shmid, const void *daddr, int shmflags);
returns: starting address of mapped region if OK, -1 on error
```

- shmid : shmget 함수에서 받은 공유메모리 식별자
- daddr : 메모리가 붙을 주소 (0인경 우 커널에서 지정)
- shmflags
  - SHM_RDONLY : 읽기전용 (이 flag가 없으면 읽기/쓰기 가능)
  - SHM_RND: daddr 값을 반올림하여 메모리 페이지의 경계에 맞춤

```
#include <sys/shm.h>
int shmdt (void *addr);
returns:0 if OK, -1 on error
```

- shmdt를 성공적으로 수행되면 커널은 shmid_ds의 내용 갱신
  - shm_dtime : 가장 최근 dettach된 시간
  - shm_lpid : 호출한 프로세스의 pid
  - shm_nattch: 현재 공유메모리를 사용하는 프로세스의 수

shmctl 함수

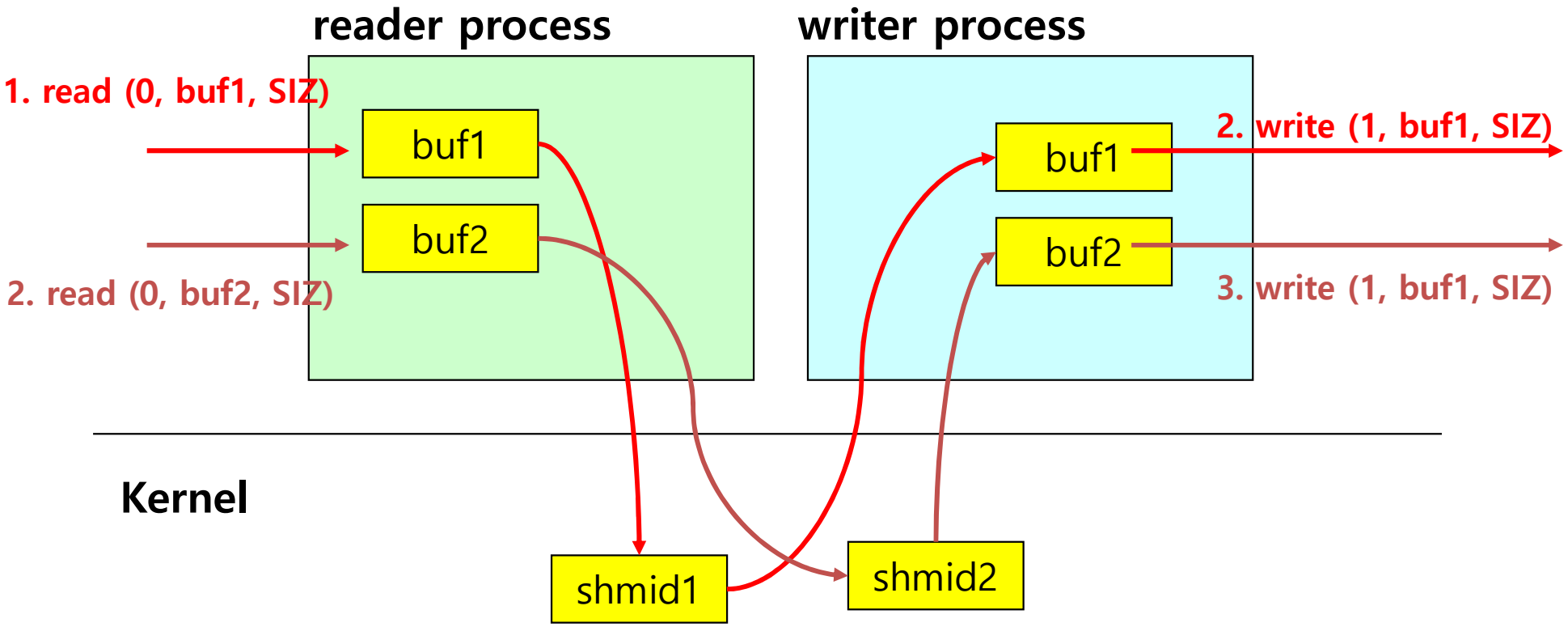
```
#include <sys/shm.h>
int shmctl( int shmid, int command, struct shmid_ds *shm_stat );
returns: 0 if OK, -1 on error
```

- **shmid : 유효한 공유 메모리 식별자**
- **command**

IPC_STAT	공유메모리 공간에 관한 정보를 가져오기 위해서 사용된다. 정보는 buf 에 저장된다.
IPC_SET	공유메모리 공간에 대한 사용자권한 변경을 위해서 사용된다. 사용자 권한 변경을 위해서는 슈퍼유저 혹은 사용자권한을 가지고 있어야 한다.
IPC_RMID	공유메모리 공간을 삭제하기 위해서 사용된다. 이 명령을 사용한다고 해서 곧바로 실행되는 것은 아니며, 더 이상 공유메모리 공간을 사용하는 프로세스가 없을 때, 즉 shm_nattch가 0일 때까지 기다렸다가 삭제된다. 즉 해당 공유메모리 공간에 대해서 삭제표시를 하는 것으로 생각하면 된다.
SHM_LOCK	공유메모리 세그먼트를 잠근다. 이 커맨드는 슈퍼유저에 의해서만 실행된다.
SHM_UNLOCK	공유메모리 세그먼트 잠금을 해지한다. 이 커맨드는 슈퍼유저에 의해서만 실행된다.

- **shm_stat : 해당 공유메모리의 상태 자료구조**

Shared Memory 예제



$ shmcp < big > /tmp/big

Shared Memory 예제

```
#define SHARED_KEY1 (key_t) 0x10 /* 공유 메모리 키 */
#define SHARED_KEY2 (key_t) 0x15 /*공유 메모리 키 */
#define SEM_KEY (key_t) 0x20 /* 세마포 키 */
#define IFLAGS (IPC_CREAT )
#define ERR ((struct databuf *) -1)

#define SIZE 2048

struct sembuf p1 = {0, -1, SEM_UNDO }, p2 = {1, -1, SEM_UNDO };
struct sembuf v1 = {0, 1, SEM_UNDO }, v2 = {1, 1, SEM_UNDO };

struct databuf {              /* data 와 read count 저장 */
        int d_nread;
        char d_buf[SIZE];
};

static int shm_id1, shm_id2, sem_id;
```

```
/* 초기화 루틴 들*/
void getseg (struct databuf **p1, struct databuf **p2)  {
      if ((shm_id1 = shmget (SHARED_KEY1, sizeof (struct databuf), 0600 | IFLAGS)) == -1) {
            perror("error shmget₩n");
            exit(1);
      }

      if ((shm_id2 = shmget (SHARED_KEY2, sizeof (struct databuf), 0600 | IFLAGS)) == -1){
            perror("error shmget₩n");
            exit(1);
      }

      if ((*p1 = (struct databuf *) shmat (shm_id1, 0, 0)) == ERR){
            perror("error shmget₩n");
            exit(1);
      }

      if ((*p2 = (struct databuf *) shmat (shm_id2, 0, 0)) == ERR){

            perror("error shmget₩n");
            exit(1);
      }
}
```

```
int getsem (void)
{
        semun x;
        x.val = 0;
        int id=-1;

        if ((id = semget (SEM_KEY, 2, 0600 | IFLAGS) ) == -1)       exit(1);
        if (semctl ( id, 0, SETVAL, x) == -1)                       exit(1);
        if (semctl ( id, 1, SETVAL, x) == -1)                       exit(1) ;

        return (id);
}


void remobj (void)
{
        if (shmctl (shm_id1, IPC_RMID, 0) == -1)                    exit(1);
        if (shmctl (shm_id2, IPC_RMID, 0) == -1)                    exit(1);
        if (semctl (sem_id, 0, IPC_RMID, 0) == -1)                  exit(1) ;
}
```

```
main () {
        pid_t pid;
        struct databuf *buf1, *buf2;

        sem_id = getsem();                          // 세마포 생성 및 초기화

        getseg (&buf1, &buf2);                      // 공유 메모리 생성 및 부착

        switch (pid = fork ()) {
                case -1:
                    perror("fork");
                    break;

                case 0:
                    writer (sem_id, buf1, buf2);    // 자식 프로세스, 공유메모리 → 표준출력
                    remobj ();
                    break;

                default:                            // 부모 프로세스 ,표준입력 → 공유메모리
                    reader (sem_id, buf1, buf2);
                    break;
        }
}
```

```
// reader – 파일 읽기 처리 (표준입력 → 공유 메모리)

void reader (int semid, struct databuf *buf1, struct databuf
*buf2)
{
   for (;;)
   {
      buf1 ->d_nread = read (0, buf1 -> d_buf, SIZE);

      semop (semid, &v1, 1);
      semop (semid, &p2, 1);

      if (buf1 -> d_nread <= 0)
         return;

      buf2 -> d_nread = read (0, buf2 -> d_buf, SIZE);

      semop (semid, &v1, 1);
      semop (semid, &p2, 1);

      if (buf2 -> d_nread <= 0)
         return;
   }
}
```

```
// writer – 파일 쓰기 처리 (공유 메모리 → 표준출력)

void writer (int semid, struct databuf *buf1, struct databuf
*buf2)
{
   for (;;)
   {

      semop (semid, &p1, 1);
      semop (semid, &v2, 1);

      if (buf1 -> d_nread <= 0)
         return;

      write (1, buf1 -> d_buf, buf1 -> d_nread);

      semop (semid, &p1, 1);
      semop (semid, &v2, 1);

      if (buf2 -> d_nread <= 0)
         return;

      write (1, buf2 -> d_buf, buf2 -> d_nread);
   }
}
```