|  |  |
| --- | --- |
| **Week 1. IPC**  **(Inter-Process Communication)** | |
| **실습목표 :** | UNIX상에서의 다양한 IPC기능의 관련함수들을 습득한다. | |

[Introduction 2](#_Toc86663796)

[1. Identifier 와 key 2](#_Toc86663797)

[2. IPC 생성 (get 함수) 2](#_Toc86663798)

[3. 허가구조 (Permission Structure) 3](#_Toc86663799)

[Message queue 4](#_Toc86663800)

[1. concept 4](#_Toc86663801)

[2. msgget 5](#_Toc86663802)

[3. msgctl 5](#_Toc86663803)

[4. msgsnd 8](#_Toc86663804)

[5. msgrcv 8](#_Toc86663805)

[6. 연습과제 11](#_Toc86663806)

[Semaphore 15](#_Toc86663807)

[1. concept 15](#_Toc86663808)

[2. semget 18](#_Toc86663809)

[3. semctl 20](#_Toc86663810)

[4. semop 21](#_Toc86663811)

[5. 연습과제 22](#_Toc86663812)

[Shared Memory 25](#_Toc86663813)

[1. concept 25](#_Toc86663814)

[2. shmget 26](#_Toc86663815)

[3. Shared memory 연산 27](#_Toc86663816)

[4. shmctl 28](#_Toc86663817)

[5. 연습과제 28](#_Toc86663818)

|  |
| --- |
| Introduction |

하나의 프로세스는 다른 프로세스에게 운영체제에서 제공해주는 (시스템 콜) 기능을 통해 메시지를 주고받을 수 있는데 이러한 기능을 통상 IPC (Interprocess Communication)라 한다. 개념적으로 IPC 는 프로그래머가 하나의 운영체계에서 동시에 수행될 개별 프로그램을 생성하고 다룰 수 있도록 해주는 프로그래밍 인터페이스 집합이다. 단일 사용자의 요구가 운영체계에서 여러 프로세스를 수행시키는 결과를 가져올 수 있기 때문에, 이러한 사용자를 위해 프로세스간 통신이 필요하게 되는데 IPC 인터페이스가 이를 가능케 한다. IPC는 프로세스간의 통신을 가능하게 해주고 프로세스간의 행위에 동기화를 제공해주는 기법을 제공해준다. 즉 하나의 일을 여러 단위 프로세스들이 협력 처리 지원하는 기법을 말한다.

예를 들어 멀티 프로세스 환경에서 공동으로 작업을 수행하려면 자료를 공유해야 하는 경우가 있다. fork() 함수로 자식 프로세스를 생성할 때 부모 프로세스는 자신이 개방하고 있던 파일에 대한 정보를 자식 프로세스에게 넘겨주지 않는다. 이런 경우 IPC를 통해 공유해야 할 파일에 대한 파일 디스크립터를 전달해준다.

전형적으로 내부 프로세스간 통신을 가능하게 해주는 기법으로 파이프, FIFO 그리고 시그널이 있다. 또한 System V 계열의 UNIX에서는 하나의 일을 여러 단위 프로세스들이 협력하여 처리하도록 하는 다중 IPC인 메시지 큐(message queue), 공유 메모리(shared memory), 세마포어(semaphore)등의 세 가지 IPC기법을 제공한다. 우리는 여기서 System V에서 제공하는 세가지 IPC 기법에 대해서 살펴보겠다.

## 1. Identifier 와 key

세 가지 IPC 설비에 대한 프로그래밍 인터페이스는 커널내의 구현상의 유사성을 반영하여 가능한 유사하게 만들어졌다. 각 IPC의 가장 중요한 공통성은 IPC 설비 *key*와 *identifier*이다. 키란 파일 이름이 파일을 식별하는 것과 같이 UNIX시스템에서 IPC객체를 식별하는 숫자이다. 즉 키는 여러 프로세스 사이에서 자원이 공유되도록 해준다. 실제로 새 IPC를 생성할 때 이 키 값을 명시해주어야 한다. 키의 실제 자료유형은 구현에 좌우되는 key\_t 유형으로 시스템 헤더파일 <sys/types.h>에 정의되어 있다. 이 키 값은 IPC를 생성하면서 커널에 의해 *identifier*로 변환된다. 이 *identifier*는 각 IPC 자료형을 명시하는 값으로 음이 아닌 정수를 취하며 다른 IPC 루틴 호출에서 사용될 수 있다.

## 2. IPC 생성 (get 함수)

IPC를 생성하는 get 함수 (msgget, semget, shmget)는 모두 key와 flag라는 공통된 인자를 가지고 있다. 다음과 같은 설정 값을 가질 때 새로운 IPC구조가 생성된다.

1) key 값이 IPC\_PRIVATE 일 때

2) key 값이 현재 특정 IPC구조와 연결되어 있지 않으며 flag인수에 IPC\_CREAT가 명시되어 있을 때.

만약 새 IPC 구조를 생성하되 이미 존재하는 IPC를 참조하지 않으려면 flag값을 IPC\_CREAT와 IPC\_EXEL로 세팅한다. 이렇게 함으로써 이미 해당하는 키 값에 대한 IPC가 이비 존재한다면 EEXIST 에러 메시지를 출력한다.

## 3. 허가구조 (Permission Structure)

한 IPC 객체가 생성될 때 시스템은 IPC 허가구조를 만들어 해당 객체에 연관된 각종 관리정보를 수록한다. 허가구조는 메시지, 세마포, 공유 메모리 모두 공통적으로 존재하며, 이 구조는 허가와 사용자 정보를 유지한다. 허가 구조는 다음과 같은 형태로 정의 되며 ipc\_perm으로 식별된다.

|  |
| --- |
| 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 IPC를 생성할 때 초기화 되어진다. 나중에 msgctl, semctl, shmctl를 이용하여 uid, gid, mode 값을 변경할 수 있다. 이러한 값을 변경하기 위해서는 해당하는 IPC를 생성한 사람이거나 슈퍼유저이어야 한다.

|  |
| --- |
| **Message queue** |

## 1. concept

메시지 큐는 문자열이나 바이트 열로 구성되는 레코드 방식으로 프로세스간 메시지를 주고받을 수 있도록 지원한다. 메시지 큐는 하나 이상의 프로세스가 메시지를 쓸 수 있고, 이를 하나 이상의 프로세스가 읽을 수 있을 수 있도록 한다.



**그림 1. Message Queue**

메시지란 문자나 바이트의 열이다. 메시지는 프로세스간의 메시지 큐를 통해 전달되며, 메시지 큐는 msgget 함수에 의해 생성되고 접근된다. 일단 큐가 만들어지면, 적당한 큐 허가를 가진 프로세서가 msgsnd를 통해 새 메시지를 메시지 큐에 추가할 수 있다. 메시지는 msgrcv에 의해 큐로부터 읽을 수 있으며, 이때 메시지는 큐로부터 제거된다.

유닉스는 메시지 큐의 리스트를 msgque 벡터로 관리한다. msgque의 각 원소는 메시지 큐에 대한 모든 것을 기술하는 msqid\_ds 자료구조를 가리킨다. 이 자료구조는 현재 큐의 상태를 정의하는데 메시지 큐를 하나 생성하면 msqid\_ds 자료구조를 시스템 메모리에서 할당받아 이 벡터에 삽입한다. 각 msqid\_ds 자료구조는 ipc\_perm 자료구조와, 이 큐에 들어온 메시지에 대한 포인터들을 가지고 있다. 또한 큐에 마지막으로 쓴 시간 같은 큐 수정 시간도 유지한다.

|  |
| --- |
| struct msqid\_ds {      struct ipc\_perm msg\_perm; /\* defines the permissions and owner \*/      struct msg \*msg\_first; /\* ptr to first message on queue \*/      struct msg \*msg\_last; /\* ptr to first message on queue \*/      ulong       msg\_cbytes; /\* current num of bytes on queue \*/      ulong       msg\_qnum; /\* num of messages on queue \*/      ulong       msg\_qbytes; /\* max num of bytes on queue \*/      pid\_t        msg\_lspid; /\* pid of last msgsnd() \*/      pid\_t        msg\_lrpid; /\* pid of last msgrcv() \*/      time\_t      msg\_stime; /\* last-msgsnd() time \*/      time\_t      msg\_rtime; /\* last-msgrcv() time \*/      time\_t      msg\_ctime; /\* last-change time \*/  }; |

## 2. msgget

함수 msgget은 존재하고 있는 큐를 open 하거나 새로운 큐를 생성하는 역할을 하는데 다음과 같이 정의된다.

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  int msgget(key\_t  *key*, int *permflags*);  returns: message queue ID if OK, -1 on error |

*key* : 메시지 큐를 시스템에 식별시키는 단순한 숫자이다. 호출이 성공하여 새 큐를 생성하거나 기존의 큐에 접근하게 되면, msgget은 메시지 큐 식별자라 불리는 음이 아닌 정수를 리턴한다.

*permflags* : msgget이 수행해야 할 정확한 작업을 결정한다. 이와 관련이 있는 두 개의 상수(IPC\_CREAT, IPC\_EXCL)가 <sys/ipc.h> 파일에 정의되어 있다. 이들은 독립적으로 사용될 수 있고, 비트별로 OR 연산될 수 있다 :

|  |  |
| --- | --- |
| IPC\_CREAT | key에 해당하는 메시지 큐가 존재하지 않는 경우에 msgget이 이를 생성하도록 지시한다. 파일과의 비유를 하면 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 이것이 동시에 설정된 경우 호출은 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. |

메시지 큐가 생성되는 경우에는 파일 모드와 유사하게 permflags의 하위 9비트가 메시지 큐에 대한 허가를 만드는 데 사용된다. 이 값은 큐와 함께 생성되는 ipc\_perm 구조에 저장된다.

예)  mqid = msgget( (key\_t) 0100, 0644 | IPC\_CREAT | IPC\_EXCL );

  키 값이 100인 메시지 큐를 생성한다. 호출이 성공하는 경우 큐의 허가는 0644가 된다. 이것은 마치 파일의 허가와 같이 해석할 수 있다. 따라서 큐를 만든 프로세스는 메시지를 큐에 넣거나, 큐에서 읽어 올 수 있다. 반면에 큐의 생성자와 같은 프로세스 그룹에 속하는 구성원들이나 그 밖의 모든 프로세스들은큐에서 메시지를 읽을 수만 있다.

## 3. msgctl

msgctl을 통해 우리는 각각 한 프로세스가 메시지 큐의 상태정보를 얻을 수 있게 하고, 메시지 큐에 관련된 제한을 변경할 수 있으며, 시스템 상에서 큐를 통째로 제거할 수 있게 한다. msgctl는 다음과 같이 사용된다.

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #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 큐의 식별자.

*command* : 수행될 작업을 시스템에 알려준다. 사용 가능한 값은 3개가 있는데 이들 전부가 IPC 설비 전반(semaphore, shared memory)에 걸쳐 모두 적용된다. 이들 값은 sys/ipc.h 에 정의된 상수에 의해 식별된다.

|  |  |
| --- | --- |
| 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. |

*msq\_stat* : msqid\_ds struct의 주소를 저장하고 있다. 이 struct는 <sys/msg.h>에 정의 되어 있다.

(예제) msgctl함수를 사용하여 하나의 메시지 큐 IPC에 대한 상태정보를 변겅할 수 있는 프로그램을 작성하시오.

|  |
| --- |
| #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\_RMIDn");  system("ipcs -q");  msgctl(msgid, IPC\_RMID, (struct msqid\_ds \*)NULL);  printf("After IPC\_RMIDn");  system("ipcs -q");  } |

## 4. msgsnd

일단 큐가 생성되면, 다음과 같은 msgsnd 함수를 사용하여 데이터를 msgget으로부터 얻은  mqid가 가리키는 메시지 큐에 추가할 수 있다.

|  |
| --- |
| #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*:유효한 메시지 큐의 식별자.

*size* : 메시지의 최대 길이.

*message* : 메시지 큐에 추가되는 size 크기의 메시지를 담고 있는 mymsg를 가리키는 포인터. 메시지를 담고 있는 구조체는 통상 다음과 같은 구조와 같다.

struct mymsg{

long mtype;

char mtext[SOMEVALUE];

};

프로그래머는 mtype 필드를 사용하여 메시지를 분류할 수 있다. 각각의 가능한 값은 각각 다른 범주를 나타낸다. 여기서 양의 정수만이 의미를 가지며 음의 정수와 0은 무의미하다. mtext는 메시지의 내용을 담는다. 실제로 전달되는 메시지의 길이는 msgsnd의 인수 size에 의해 주어지며 이 값의 범위는 SOMEVALUE와 시스템이 결정한 최대 값 중에 작은 값을 취한다.

*flag* : 단하나의 의미 있는 값 IPC\_NOWAIT을 가진다.

IPC\_NOWAIT이 설정된 경우 : 메시지 큐가 full 상태일 경우 EAGAIN 에러 메시지를 리턴하게 하여 추후에 제시도 할 것을 의미한다.

(예제)

|  |
| --- |
| #include <stdio.h>  #include <sys/msg.h>  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|0644);  if(msgid == -1) {  perror("msgget");  exit(1);  }    mesg.mtype = 1;  strcpy(mesg.mtext, "Message Q Testn");    msgsnd(msgid, (void \*)&mesg, 80, 0);  } |

## 5. msgrcv

메시지는 msgrcv에 의해 회수할 수 있다. 큐의 허가에 저촉되지 않는 한, mqid가 가리키는 큐로부터 메시지를 읽는다. 한 메시지를 큐에서 읽으면 그 메시지는 큐에서 제거된다.

|  |
| --- |
| #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: 0 if OK, -1 on error |

*mqid*:유효한 메시지 큐의 식별자.

*message* : 받아들인 메시지를 저장한다.

*size* : 이 구조에 저장할 수 있는 최대 길이를 지정한다.

*msg\_type* : 인수는 실제로 어떤 메시지를 받아들일지를 결정한다. 선택을 하는 기준은 메시지의 mtype 필드이다.

msg\_type == 0  큐의 제일 첫 번째 메시지가 리턴 된다.

msg\_type >  0   메시지 큐에서 메시지 타입이 msg\_type와 같은 값을 가지는 첫 번째 메시지를 리턴 한다.

msg\_type <  0

*flags* : 제어 정보를 갖게 된다. 유효한 값으로 두 개의 값 IPC\_NOWAIT와 MSG\_NOERROR를 단독으로 또는 OR연산을 통해 사용할 수 있다.

(예제)

|  |
| --- |
| #include <stdio.h>  #include <sys/msg.h>  main() {  struct msgbuf \*inmesg;  key\_t key;  int msgid;  key = ftok("keyfile", 1);  msgid = msgget(key, 0);  inmesg = (struct msgbuf \*)malloc(sizeof(long) + 80);  msgrcv(msgid, inmesg, 80, 0, 0);  printf("Received Msg = %s", inmesg->mtext);  } |

## 6. 연습과제

메시지 큐를 사용하여 두 프로세스 간에 간단한 메시지를 주고받는 프로그램을 작성하시오.

(메시지 전달)

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <stdio.h>  #include <string.h>  #define MSGSZ     128  typedef struct msgbuf {           long    mtype;           char    mtext[MSGSZ];           } message\_buf;  main()  {      int msqid;      int msgflg = IPC\_CREAT | 0666;      key\_t key;      message\_buf sbuf;      size\_t buf\_length;      key = 1234;      fprintf(stderr, "\nmsgget: Calling msgget(%#lx, %#o)\n", key, msgflg);      if ((msqid = msgget(key, msgflg )) < 0) {          perror("msgget");          exit(1);      }      else       (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);          sbuf.mtype = 1;        (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);        (void) strcpy(sbuf.mtext, "Did you get this?");        (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);        buf\_length = strlen(sbuf.mtext) + 1 ;        if (msgsnd(msqid, &sbuf, buf\_length, IPC\_NOWAIT) < 0) {         printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf\_length);          perror("msgsnd");          exit(1);      }     else        printf("Message: \"%s\" Sent\n", sbuf.mtext);        exit(0);  } |

(메시지 받음)

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <stdio.h>  #define MSGSZ     128  /\*  \* Declare the message structure.  \*/  typedef struct msgbuf {      long    mtype;      char    mtext[MSGSZ];  } message\_buf;  main()  {      int msqid;      key\_t key;      message\_buf  rbuf;      /\*       \* Get the message queue id for the       \* "name" 1234, which was created by       \* the server.       \*/      key = 1234;      if ((msqid = msgget(key, 0666)) < 0) {          perror("msgget");          exit(1);      }        /\*       \* Receive an answer of message type 1.       \*/      if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {          perror("msgrcv");          exit(1);      }      /\*       \* Print the answer.       \*/      printf("Recieved: %s\n", rbuf.mtext);      exit(0);  } |

다음과 같은 메시지를 주고받을 수 있는 두 프로세스를 작성하시오

(a) 프로세스1. “Hello can you here me?"

(b) 프로세스2. “Copy that P1 how are you?"

(c) 프로세스1. “Doing great!!"

|  |
| --- |
| **Semaphore** |

## 1. concept

프로세스 동기화 문제에 대한 해결책으로 세마포(Semaphore)라는 개념을 E.W. Dijkstra에 의해 처음으로 도입되었다. 세마포(Semaphores)를 비록 IPC 설비의 하나로 분류하였으나, 다른 파이프, 메시지 큐, FIFO 등과는 다르다. 다른 IPC 설비들이 대부분 프로세스간 메시지 전송을 목적으로 하는데 반해서 세마포는 프로세스간 데이터를 동기화하고 보호하는데 목적이 있다.

프로세스간 메시지 전송을 하거나, 혹은 공유메모리를 통해서 특정 데이터를 공유하게 될 경우 발생하는 문제를 막기 위해서는 공유된 자원에 여러 개의 프로세스가 동시에 접근을 하면 안 되며, 단지 한번에 하나의 프로세스만 접근 가능하도록 만들어줘야 할 것이다. 이것은 쓰레드에서 메시지간 동기화를 위해서 mutex를 사용하는 것과 같은 이유이다.

하나의 데이터에 여러 개의 프로세스가 관여할 때 어떤 문제점이 발생할 수 있는지 간단한 예를 들어보자.

|  |
| --- |
| int count=10;  A 프로세스가 count를 읽는다. 10  B 프로세스가 count를 읽는다. 10  B 프로세스가 count를 1 증가 시킨다. 11  A 프로세스가 count를 1 증가 시킨다. 11 |

count는 공유 메모리이며 A와 B 프로세스가 여기에 대한 작업을 한다. A가 1을 증가 시키고 B가 1을 증가시키므로 최종 count 값은 12 가 되어야 할 것이다. 그러나 A 가 작업을 마치기 전에 B가 작업을 하게 됨으로 엉뚱한 결과를 보여주게 된다. 위의 문제를 해결하기 위해서는 count에 A가 접근할 때 B프로세스가 접근하지 못하도록 block시키고, A가 모든 작업을 마쳤을 때 B프로세스가 작업을 할 수 있도록 block을 해제 시키면 될 것이다. 세마포는 이러한 작업을 할 수 있도록 해준다. 즉 세마포는 여러 개의 프로세스에 의해서 공유되는 자원의 접근제어를 가능하게 해주는 도구이다.

원리는 매우 간단하다. 차단을 원하는 자원에 대해서 세마포를 생성하면 해당자원을 가리키는 세마포 값이 할당된다. 이 세마포 값에는 현재 세마포를 적용하고 있는 자원에 접근할 수 있는 프로세스의 숫자를 나타낸다. 이 값이 0이면 이 자원에 접근할 수 있는 프로세스의 숫자가 0이라는 뜻이며, 0보다 큰 정수면 해당 정수의 크기만큼의 프로세스가 자원에 접근할 수 있다는 뜻이 된다. 그러므로 우리는 접근제어를 해야 하는 자원에 접근하기 전에 세마포 값을 검사해서 값이 0이면 자원을 사용할 수 있을 때까지 기다리고, 0보다 크면(1이라고 가정하자) 자원에 접근해서 세마포 값을 1 감소 시켜서 (세마포 값은 0이 된다) 다른 프로세스가 자원에 접근할 수 없도록 하며, 자원의 사용이 끝나면 세마포 값을 다시 1증가시켜서 다른 프로세스가 자원을 사용할 수 있도록 만들어준다. 만약 세마포 값을 검사했는데 값이 0이라면 사용할 수 있을 때까지 (1이 될 때까지) 기다려야하는 block 상태에 있게 된다.

세마포 S는 세마포 연산을 허용하는 정수로 wait와 signal에 해당하는 세마포 연산 p와 v를 토대로 한다. p와 v 함수의 pseudocode는 다음과 같이 정의한다.

|  |
| --- |
| P(S) {  while(S<=0); //no-loop  S--;  } |

|  |
| --- |
| V(S) {  S++;  } |

위의 두 연산 p와 v를 사용해서 n개의 프로세스가 mutex라는 세마포를 사용해서 critical section을 처리하게 할 수 있다. 각 프로세스 아래 그림과 같이 구성되며 하나의 프로세스가 critical section에 접근 했을 때 p(mutex)에 의해 다른 프로세스의 접근을 block 하게 되며 이후 critical section의 처리가 끝나고 v(mutex)가 실행되면 다른 프로세스가 접근할 수 있게 된다. 여기서는 System V UNIX에서 세마포를 어떻게 호출하고 사용하는지 알아보자.



**그림 2. 세마포어 개념**

세마포는 그 특성상 원자화된 연산을 필요로 한다. 이러한 원자화된 연산은 유저레벨의 함수에서는 제공하기가 힘들기 때문에 세마포 정보는 메시지 큐와 같이 커널에서 전용 구조체를 이용해서 관리하게 된다. 다음은 커널에서 세마포 정보를 유지하기 위해서 관리하는 구조체인 semid\_ds 구조체의 모습이다.

|  |
| --- |
| struct semid\_ds  {  struct ipc\_perm sem\_perm;  time\_t sem\_otime;  time\_t sem\_ctime;  unsigned long int sem\_nsems;  struct sem \*sem\_base; /\* pointer to first semaphore in set \*/  }; |

sem\_perm 은 세마포에 대한 퍼미션으로 일반 파일퍼미션과 마찬가지의 기능을 제공한다. 즉 현재 세마포어 구조체에 접근할수 있는 사용자권한을 설정한다. sem\_nsems는 생성할 수 있는 세마포의 크기이다. sem\_otime은 마지막으로 세마포관련 작업을 한 시간이며, sem\_ctim은 마지막으로 구조체 정보가 바뀐 시간이다.

## 2. semget

세마포의 생성 혹은 기존에 만들어져 있는 세마포에 접근하기 위해서는 semget() 함수를 사용한다. semget 호출은 msgget과 유사하다. semget의 사용법은 다음과 같다.

|  |
| --- |
| #include <sys/sem.h>  int semget( key\_t key, int nsems, int permflags ); |

첫 번째 아규먼트 key는 세마포의 유일한 키 값을 부여하여 세마포를 생성하거나 접근할 수 있게 해준다. 키 값은 int형이다.

추가된 인수 nsems는 세마포 집합에 필요한 세마포의 개수를 나타낸다. 이는 UNIX에서 세마포 연산은 여러 세마포를 하나의 집합으로 다룬다는 점을 시사한다. 세마포 집합에 대한 인덱스는 0부터 nsems-1까지 있을 수 있다. (nsems는 집합내의 세마포 전체의 개수이다) 이 값은 최초 세마포를 생성하는 생성자의 경우에 크기가 필요하다(보통 1). 그 외에 기존의 세마포에 접근해서 사용하는 경우에는 세마포를 만들지 않고 단지 접근만 할뿐임으로 크기는 0이 된다.

semflg를 통해서 새로 생성되거나 기존의 세마포에 접근하는 것을 제어할 수 있다. 다음은 semflg의 아규먼트이다.

|  |  |
| --- | --- |
| 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. |

semflg를 통해서 세마포에 대한 퍼미션을 지정할 수도 있다. 퍼미션 지정은 보통의 파일에 대해서 유저/그룹/other에 대해서 지정하는 것과 같다.

만약 IPC\_CREAT만 사용할 경우 해당 key 값으로 존재하는 세마포가 없다면, 새로 생성하고, 이미 존재한다면 존재하는 세마포어의 id를 넘겨준다. IPC\_EXCL을 사용하면 key값으로 존재하는 세마포가 없을 경우 새로 생성되고, 이미 존재한다면 존재하는 id 값을 돌려주지 않고 실패 값 (-1)을 되돌려주고, errno를 설정한다.

semget 호출이 성공하면 세마포 집합 식별자 semid라는 메시지 큐 식별자와 유사한 역할을 하는 값이 돌아온다.

각 세마포는 다음과 같은 구조를 가지며 값들의 의미는 다음과 같다.

|  |
| --- |
| struct sem {  ushort  semval;  pid\_t   sempid;  ushort  semncnt;  ushort  semzcnt;  }; |

|  |  |
| --- | --- |
| semval | 세마포의 값으로 항상 양의 정수가 지정된다. 여기에 값을 새로 지정하려면 반드시 세마포 시스템 호출을 통해야 한다. 즉 세마포는 한 프로그램에서 일반 자료형의 변수와 같이 직접 접근할 수는 없다. |
| sempid | 세마포에 최근에 접근했던 프로세스의 프로세스 식별번호이다 |
| semncnt | 세마포의 값이 현재보다 더 큰 값을 갖기를 기다리는 프로세스의 수 |
| semzcnt | 세마포의 값이 0이 되기를 기다리는 프로세스의 수 |

(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);  } |

이제 위의 코드를 컴파일해서 실행시키고 나서 실제로 세마포 정보가 어떻게 바뀌였는지 확인해 보도록 하자. 커널에서 관리되는 IPC 정보를 알아보기 위해서는 ipcs()라는 도구를 이용하면 된다.

## 3. semctl

|  |
| --- |
| #include <sys/sem.h>  int semctl( int semid, int sem\_num, int command, union semun ctl\_arg ); |

위의 정의에서처럼 semctl 함수는 msgctl보다 복잡하다. semid는 semget 호출로부터 복귀된 유효한 세마포 식별자여야 한다. command는 msgctl에서와 같이 수행할 정확한 기능을 명시한다. 이 기능에는 세 가지 범주가 있다. 표준 IPC 기능, 단일 세마포에만 영향을 미치는 기능, 세마포 전체 집합에 영향을 미치는 기능이 그것이다. 가능한 모든 기능은 아래 표에 정리하였다.

sem\_num 인수는 semctl 옵션의 두 번째 그룹에 사용된다. 즉 단일 세마포 기능에서 특정 세마포를 식별해준다. 마지막 인수인 ctl\_arg는 아래와 같이 정의하는 union 이다.

|  |
| --- |
| union semun{  int val;  struct semid\_ds \*buf;  unsigned short \*array;  }; |

union의 각 구성원은 세 가지 유형의 semctl 기능 각각에 대해 지정되는 다른 유형의 값을 나타낸다. semctl의 중요한 용도중 하나는 세마포의 초기값을 지정한다는 것이다. semget은 프로세스에게 이러한 작업을 허용하지 않는다.

|  |  |
| --- | --- |
| **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 |

## 4. semop

|  |
| --- |
| #include <sys/sem.h>  int semop( int semid, struct sembuf \*op\_array, size\_t num\_ops ); |

semid는 세마포 집합 식별자로서, 아마도 이전에 semget 호출을 통해 얻어졌을 것이다. op\_array 인수는 sembuf 구조의 배열로서 sembuf 구조는 <sys/sem.h>에 정의되어있다. num\_ops 인수는 배열내의 sembuf 구조의 수이다. 각각의 sembuf 구조는 한 세마포에 대해 수행할 연산에 대한 명세를 저장하고 있다.

semop 함수가 세마포 집합에 대해 수행하는 일련의 연산들은 모두 원자화되어야 한다. 다시 말해, 이중 하나의 연산이라도 수행 할 수 없다면 전체연산이 수행되지 말아야 한다.

sembuf 구조를 보면 다음과 같은 구성원을 포함 한다.

|  |
| --- |
| struct sembuf{  unsigned short sem\_num;  short sem\_op;  short sem\_flg;  } |

sem\_num은 집합내의 세마포에 대한 인덱스를 저장한다. 예를 들어 집합 내에 세마포가 하나면 sem\_num의 값은 0이다.

sem\_op는 함수 semop이 수행해야하는 기능을 정수로서 나타낸다. 여기에는 세 가지 경우가 있다.

*case 1* : sem\_op가 음수일 때

이것은 앞서 소개한 세마포 명령 p()의 일반화된 형태이다. 기본 아이디어는 함수 semop는 먼저 세마포 sem\_num에 관련된 semval 값을 조사하고 semval 값이 충분히 크다면 즉시 감소시킨다. 아니면 semval 값이 충분히 커질 때까지 프로세스가 기다린다. 그러나 sem\_flgdp IPC\_NOWAIT 플래그가 지정되어 있으면 sem\_op은 즉시 -1을 되돌려주고 errno값을 EAGAIN으로 한다.

*case 2* : sem\_op가 양수일 때

이것은 v() 연산을 수행할 때 이다. 즉 sem\_op의 값을 해당 semval에 더해준다. 이 때 해당 세마포의 값이 증가하기를 기다리는 프로세스들을 깨운다.

*case 3* : sem\_op가 0일 때

이 경우는 semval은 변경되지 않고 sem\_op는 값이 0이 될 때까지 기다린다. semval이 아직 0이 아니고 sem\_flg에서 IPC\_NOWAIT가 지정된 경우 semop는 즉시 오류 값을 돌려주게 된다.

sem\_flg는 IPC\_NOWAIT와 SEM\_UNDO 2개의 설정할 수 있는 값을 가지고 있다. IPC\_NOWAIT 는 none block 모드 지정을 위해서 사용되며, SEM\_UNDO는 프로세스가 세마포를 돌려주지 않고 종료해버릴 경우 커널에서 알아서 세마포 값을 조정(증가) 할 수 있도록 만들어 준다.

## 5. 연습과제

(예제)

|  |
| --- |
| 1 #include <sys/types.h>  2 #include <sys/ipc.h>  3 #include <sys/sem.h>  4 #include <errno.h>  5  6 union semun {  7 int val;  8 struct semid\_ds \*buf;  9 unsigned short \*array;  10 };  11  12 int initsem(key\_t semkey) {  13 union semun semunarg;  14 int status = 0, semid;  15  16 semid = semget(semkey, 1, IPC\_CREAT | IPC\_EXCL | 0600);  17 if(semid == -1) {  18 if(errno == EEXIST)  19 semid = semget(semkey, 1, 0);  20 }  21 else {  22 semunarg.val = 1;  23 status = semctl(semid, 0, SETVAL, semunarg);  24 }  25 if(semid == -1 || status == -1) {  26 perror("initsem");  27 return (-1);  28 }  29 return semid;  30 }  31  32 int semlock(int semid) {  33 struct sembuf buf;  34  35 buf.sem\_num = 0;  36 buf.sem\_op = -1;  37 buf.sem\_flg = SEM\_UNDO;  38 if(semop(semid, &buf, 1) == -1) {  39 perror("semlock failed");  40 exit(1);  41 }  42 return (0);  43 }  44  45 int semunlock(int semid) {  46 struct sembuf buf;  47  48 buf.sem\_num = 0;  49 buf.sem\_op = 1;  50 buf.sem\_flg = SEM\_UNDO;  51 if(semop(semid, &buf, 1) == -1) {  52 perror("semunlock failed");  53 exit(1);  54 }  55 return (0);  56 }  57  58 void semhandle() {  59 int semid;  60 pid\_t pid = getpid();  61  62 printf("Process %d : Before Lockn", pid);  63 if((semid = initsem(IPC\_PRIVATE)) < 0)  64 exit(1);  65  66 semlock(semid);  67 printf("In Lock Moden");  68 semunlock(semid);  69 printf("Process %d : After Lockn", pid);  70 exit(0);  71 }  72  73 main() {  74 int a;  75 for(a=0; a<3; a++)  76 if(fork() == 0) semhandle();  77 } |

하나의 파일에 2개의 프로세스가 동시에 접근하는 프로세스를 만들어보자. account파일에 접근하여 money변수를 읽고 일정 금액을 인출하는 시스템을 구성하라. 한 프로세스가 작업을 수행 중일 때 다른 프로세스가 금액을 모두 인출하는 과정에 있는 시나리오를 만들고 인출하는 동시에 다른 프로세스가 접근하여 인출을 시도하는 프로그램을 작성하시오. 또한 세마포를 사용하지 않았을 때 어떠한 결과가 나오는지 비교 하시오.

|  |
| --- |
| **Shared Memory** |

## 1. concept

보통 프로세스에서 사용되는 메모리영역은 해당 프로세스만이 사용할 수 있다. 하지만 때때로 여러 개의 프로세스가 특정 메모리영역을 사용해야 할 때가 있다. System V IPC 설비중의 하나인 공유메모리 (Shared memory)는 이러한 일을 할 수 있도록 해준다.

모든 프로세스는 자신의 업무를 수행하기 위해서 필요한 자료를 저장하기 위한 메모리 공간을 가지게 된다. 이러한 메모리공간에는 CPU에 의해 수행되는 명령어들, 프로그램 시작 시 정의되고 초기화된 데이터, 프로그램 시작 시 정의되었지만 초기화되지 않은 데이터, 함수 호출에 필요한 정보, 동적할당이 이루어지는 데이터 등이 들어가게 된다.

프로세스는 시작 시 혹은 실행 중에 이러한 데이터를 저장하고 사용하기 위한 메모리 공간을 커널에 요구하여서 할당받아 사용하게 되는데, 이러한 메모리공간은 기본적으로 메모리를 요청한 프로세스만이 접근 가능하도록 되어있다. 하지만 가끔은 여러 개의 프로세스가 특정 메모리 공간을 동시에 접근해야할 필요성을 가질 때가 있을 것이다. 공유메모리는 이러한 작업을 위한 효율적인 방법을 제공한다.

공유메모리는 여러 IPC 중에서 가장 빠른 수행속도를 보여준다. 그 이유는 하나의 메모리를 공유해서 접근하게 되므로, 데이터 복사와 같은 불필요한 오버헤드가 발생하지 않기 때문이다. 그러나 하나의 프로세스가 메모리에 접근중일 때, 또 다른 프로세스가 메모리에 접근하는 일이 발생하면 자칫 데이터가 훼손될 수 있으므로, 한번에 하나의 프로세스가 메모리에 접근하고 있다는 것을 보증해줄 수 있어야 한다. 이러한 작업을 위해서 앞서 살펴본 세마포를 통해 제어할 수 있다.

메모리의 일부를 공유하려면 먼저 shmget 호출로 공유메모리영역을 생성한다. 메모리가 형성되면 프로세스는 shmat를 이용하여 자신을 그 메모리에 부착하고 자신의 목적에 맞게 사용할 수 있다. 그 메모리가 더 이상 필요하지 않게 되면 프로세스는 shmdt를 사용하여 자신을 떼어낼 수 있다.

우선 공유메모리가 어떻게 할당되고, 어떤 과정을 통해서 접근가능한지에 대해서 알아보자. 공유메모리의 생성요청은 최초 공유메모리 영역을 만드는 프로세스가 커널에 공유메모리 공간의 할당을 요청함으로써 이루어지며, 만들어진 공유메모리는 커널에 의해서 관리하게 된다. 따라서 한번 만들어진 공유메모리는 운영체제를 리부팅하거나, 직접 공유메모리 공간을 삭제시켜주지 않은 한, 공유메모리를 사용하는 모든 프로세스가 없어졌다고 하더라도, 지속적으로 유지되게 된다.

프로세스가 커널에게 공유메모리 공간을 요청하게 되면, 커널은 공유메모리 공간을 할당시켜주고 이들 공유메모리 공간을 관리하기 위한 내부자료구조를 통하여 공유메모리를 관리하게 된다. 이 자료구조는 shmid\_ds라는 구조체에 의해서 관리되며 <shm.h>에 정의되어 있다.

|  |
| --- |
| struct shmid\_ds  {  struct ipc\_perm shm\_perm; // 퍼미션  int shm\_segsz; // 메모리 공간의 크기  time\_t shm\_dtime; // 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; // 마지막으로 작동한 프로세스의 pid  short shm\_nattch; // 현재 접근한 프로세스의 수  }; |

## 2. shmget

공유메모리는 shmget 호출로 생성된다.

|  |
| --- |
| #include <sys/shm.h>  int semop( key\_t key, size\_t size, int permflags ); |

이 호출은 msgget이나 semget과 밀접하게 대응된다. 여기서 인수 size는 공유할 메모리의 영역의 필요한 최소 크기를 바이트 단위로 나타낸다. key는 영역을 식별하기 위한 키 값이다. permflags는 메모리 영역에 대한 허가로, msgget이나 semget의 경우와 같이 IPC\_CREAT와 IPC\_EXCL가 OR로 묶일 수 있다.

(예제)

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>    main() {  key\_t key;  int shmid;  key = ftok("/etc/passwd", 1);  shmid = shmget(key, 1024, IPC\_CREAT|0644);  if(shmid == -1) {  perror("shmget");  exit(1);  }  } |

## 3. Shared memory 연산

shmget에 의해 생성되는 메모리 영역은 프로세스의 논리적 자료공간이 아니라 물리적 메모리의 일부이다. 따라서 프로세스가 이를 사용하려면 shmat 호출을 사용하여 이 메모리 영역을 자신의 논리적 자료공간에 명시적으로 부착해야 한다.

**shmat**

|  |
| --- |
| #include <sys/shm.h>  int shmat( int shmid, const void \*daddr, int shmflags ); |

일단 공유메모리 공간을 생성했으면, 우리는 공유메모리에 접근할 수 있는 int 형의 식별자(shmid)를 얻게 된다. 우리는 이 식별자를 shmat를 이용해서 지금의 프로세스가 공유메모리를 사용가능하도록 덧붙임 작업을 해주어야 한다. shmat는 shmid가 지정하는 메모리 영역과 호출 프로세스의 유효한 주소를 연관짓는다. 이 주소는 shmat에 의해 복귀된 것이다.

첫 번째 아규먼트는 shmget을 이용해서 얻어낸 식별자 번호이며, 두 번째 아규먼트는 메모리가 붙을 주소를 명시하기 위해 사용하는데, 0을 사용할 경우 커널이 메모리가 붙을 주소를 명시하게 된다. 특별한 사항이 없다면 0을 사용하도록 한다.

세 번째 아규먼트를 이용해서, 우리는 해당 공유메모리에 대한 읽기전용, 읽기/쓰기가능 모드를 결정할 수 있는데, SHM\_RDONLY를 지정할 경우 읽기전용으로, 아무 값도 지정하지 않을 경우 읽기/쓰기가능 모드로 열리게 된다. shmflags 아규먼트의 또 다른 인수인 SHM\_RND는 daddr값이 0이 아닌 경우 이를 처리하는 법을 결정하는데, 만일 SHM\_RND값이 설정되어 있으면 daddr 값을 반올림하여 메모리의 페이지 경계에 맞추고 그렇지 않으면 daddr 값을 정확히 그대로 사용하게 된다.

**shmdt**

shmdt는 앞의 호출과는 반대로 공유메모리 영역을 프로세스의 논리적 주소공간으로부터 떼어내는 것이다.

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

프로세스가 더 이상 공유메모리를 사용할 필요가 없을 경우 프로세스와 공유메모리를 분리하기 위해 사용한다. 이 함수를 호출할 경우 단지 현재 프로세스와 공유메모리를 분리시킬 뿐, 공유메모리 내용을 삭제하지는 않는다. 공유메모리를 커널상에서 삭제하길 원한다면 shmctl 함수를 이용해야 한다.

shmdt 가 성공적으로 수행되면 커널은 shmid\_ds 의 내용을 갱신한다. 즉 shm\_dtime, shm\_lpid, shm\_nattch 등의 내용을 갱신하는데, shm\_dtime 는 가장 최근에 dettach (즉 shmdt를 사용한)된 시간, shm\_lpid는 호출한 프로세스의 pid, shm\_nattch는 현재 공유메모리를 사용하는 (shmat를 이용해서 공유메모리에 붙어있는) 프로세스의 수를 돌려준다. shmdt 를 사용하게 되면 shm\_nattch는 1 감소하게 될것이며, shm\_nattch가 0이면 더 이상 붙어있는 프로세스가 없다는 뜻이 될 것이다. shm\_nattch가 0이 되었을 때 만약 이 공유메모리가 shm\_ctl 등에 의해 삭제표시가 되어 있다면 이 공유메모리는 삭제된다.

## 4. shmctl

|  |
| --- |
| #include <sys/shm.h>  int shmctl( int shmid, int command, struct shmid\_ds \*shm\_stat ); |

shmctl은 공유메모리를 제어하기 위해서 사용한다. 즉 shmid\_ds를 직접 제어함으로써, 해당 공유메모리에 대한 소유자, 그룹 등의 허가권을 변경하거나, 공유메모리를 삭제 혹은, 공유메모리의 잠금을 설정하거나 해제하는 등의 작업을 한다.

command를 이용해서 원하는 제어를 할 수 있으며, command를 이용해 내릴 수 있는 명령에는 다음과 같은 것들이 있다.

|  |  |
| --- | --- |
| 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. |

## 5. 연습과제

(예제) 부모프로세스와 자식프로세스 사이에 공유메모리를 통해 데이터를 주고 받는 프로그램이다. 부모-자식프로세스간에 통신할 것이므로 10행에서 ‘IPC\_PRIVATE'을 키로 하여 공유메모리를 생성하였다.

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>    main() {  key\_t key;  int shmid, i;  char \*shmaddr;    shmid = shmget(IPC\_PRIVATE, 10, IPC\_CREAT|0644);  if(shmid == -1) {  perror("shmget");  exit(1);  }    switch(fork()) {  case -1:  perror("fork");  exit(1);  break;  case 0:  shmaddr = (char \*)shmat(shmid, (char \*)NULL, 0);  printf("Child Process =====n");  for(i=0; i<10; i++)  shmaddr[i] = 'a' + i;  exit(0);  break;  default:  wait(0);  shmaddr = (char \*)shmat(shmid, (char \*)NULL, 0);  printf("Parent Process =====n");  for(i=0; i<10; i++)  printf("%c ", shmaddr[i]);  printf("n");  shmdt((char \*)shmaddr);  shmctl(shmid, IPC\_RMID, (struct shmid\_ds \*)NULL);  break;  }  } |

(예제) Talker와 Listener라는 두 개의 독립적인 프로세스 사이에 공유메모리를 통하여 메시지를 주고 받는 프로그램이다. 두 개의 프로세스간에 시그널을 통하여 데이터를 읽어가도록 신호를 보낸다.

Listener 프로그램

|  |
| --- |
| #include <sys/types.h>  #include <sys/mman.h>  #include <signal.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <unistd.h>    void handler(int dummy) {  ;  }    main() {  key\_t key;  int shmid;  void \*shmaddr;  char buf[1024];  sigset\_t mask;    key = ftok("/etc/passwd", 1);  shmid = shmget(key, 1024, IPC\_CREAT|0666);    sigfillset(&mask);  sigdelset(&mask, SIGUSR1);  sigset(SIGUSR1, handler);  printf("Listener wait for Talkern");  sigsuspend(&mask);    printf("Listener Start =====n");  shmaddr = shmat(shmid, NULL, 0);  strcpy(buf, shmaddr);  printf("Listener received : %sn", buf);    strcpy(shmaddr, "Have a nice day.n");  sleep(10);  shmdt(shmaddr);  shmctl(shmid, IPC\_RMID, NULL);  } |

Talker 프로그램

|  |
| --- |
| #include <sys/types.h>  #include <signal.h>  #include <sys/mman.h>  #include <sys/ipc.h>  #include <sys/shm.h>    main(int argc, char \*\*argv) {  key\_t key;  int shmid;  void \*shmaddr;  char buf[1024];    key = ftok("/etc/passwd", 1);  shmid = shmget(key, 1024, 0);    shmaddr = shmat(shmid, NULL, 0);  strcpy(shmaddr, "Hello, I'm talkern");    kill(atoi(argv[1]), SIGUSR1);    sleep(1);  strcpy(buf, shmaddr);  printf("Listener said : %sn", buf);  sleep(3);  system("ipcs");  shmdt(shmaddr);  } |

위의 공유메모리 함수들을 사용하여 int 형의 공유메모리 공간을 할당한 다음 자식프로세스에서 1씩을 더하고 부모프로세스에서는 공유메모리 내용을 출력하는 일을 하는 간단한 프로그램을 작성하시오.

|  |
| --- |
| **참고문헌** |

UNIX Network Programming Volume 1,2 by W.Richard Stevens