운영 체제

System call

강 의	운영체제
담 당 교 수	이 강 우 교수님
학 과	컴퓨터 공학과
학 번	2017112091
이 름	박 지 호
제 출 일 자	2019 . 04. 30

Process & Job Manipulation (or Control)

: 프로세스 생성, 종료와 같이 프로세스를 처리한다

1. load, execute, end, abort

- execve() : execute program
- exit () : create a file or device

int execve(const char *filename, char *const argv[], char *const envp[]);

: execute program

<<Parameter>>

pathname: 실행할 프로그램. 이로 인해 현재 실행중인 프로세스에서 실행 중인 프로그램이 새로 초기화 된 스택, 힙 및 데이터 세그먼트가 있는 새로운 프로그램이 실행된다.

argv: 새 프로그램에 전달되는 인수 문자열의 배열. argv[0]은 실행중인 파일과 관련된 파일 이름을 포함해야 한다.

envp: key = value형식의 환경변수 문자열 배열리스트로 마지막은 NULL이어야 한다.

<<반환값>>

정상적으로 처리되면 execve(2) 다음 로직은 실행할 수 없다.

리턴 값이 -1이면, binary 교체가 실패하였으며, 상세한 오류 내용은 errno전역변수에 설정된다.

<<Sample Code>>

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

          int
          main(int argc, char *argv[])
          {
              char *newargv[] = { NULL, "hello", "world", NULL };
              char *newenviron[] = { NULL };

              if (argc != 2) {
                  fprintf(stderr, "Usage: %s <file-to-exec>\n",
argv[0]);
                  exit(EXIT_FAILURE);
```

```
            }

            newargv[0] = argv[1];

            execve(argv[1], newargv, newenviron);
            perror("execve");   /* execve() returns only on
error */
            exit(EXIT_FAILURE);
        }
```

void _exit(int *status*);

: terminate the calling process

실행중인 현재 프로세스를 종료한다.

<<Parameter>>

status: 부모 프로세스로 넘겨줄 값을 설정한다. Status가 0이면 보통 정상 종료로 인식한다. 부모 프로세스는 wait(2)또는 waitpid(2)등으로 status 값을 받을 수 있다.

<<반환값>>

이 함수는 프로그램을 종료 시키므로 그 다음 로직을 실행하지 않기때문에 return값을 받을 수 없다.

<<Sample Code>>

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>


int main()
{
   int pid;
   int status;
   int spid;
   pid = fork();

   if (pid == 0)
   {
      sleep(5);
      printf("I will be back %d\n", getpid());
      return 1;
   }

   else if(pid > 0)
   {
      printf("Im parent %d\n", getpid());
```

```
        printf("Press any key and wait\n");
        getchar();
        // 자식프로세스를 wait 한다.
        // 자식프로세스의 종료상태는 status 를 통해 받아온다.
        spid = wait(&status);
        printf("자식프로세스 wait 성공 \n");
        // 자식프로세스의 PID, 리턴값, 종료상태(정상종료혹은 비정상종료)를
        // 얻어온다.
        printf("PID        : %d\n", spid);
        printf("Exit Value  : %d\n", WEXITSTATUS(status));
        printf("Exit Stat   : %d\n", WIFEXITED(status));
    }
    else
    {
        perror("fork error :");
    }
}
```

2. create & kill processes

- fork() : create a child process
- kill() : send signal to a process

pid_t fork(void);

: 현재 실행중인 프로세스와 동일한 기능을 하는 새로운 프로세스를 생성한다.

<<Parameter>>

없음

<<반환값>>

+값 : fork()에서 1이상을 return받는 프로세스는 부모 프로세스이며, Return값은 자식프로세스 id입니다.

0 : fork()에서 0을 Return받은 프로세스는 child프로세스 자신이다.

-1 : 오류가 발생하여 자식 프로세스가 생성되지 않았다는 뜻이다. 상세 오류내용은 errno에 설정된다.

<<Sample Code>>

```
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char **argv) {
```

```
    int pid;
    pid = fork();
    if (pid > 0){
        printf("부모 프로세스 %d : %d₩n", getpid(), pid);
        pause();
    } else if (pid == 0) {
        printf("자식 프로세스 %d₩n", getpid());
        pause();
    } else if (pid == -1) {
        perror("fork error : ");
        exit(0);
    }
}
```

int kill(pid_t *pid*, int *sig*);

: 특정 프로세스나 프로세스 그룹에 시그널을 보내기 위해서 사용한다.

<<Parameter>>

pid : pid가 양수이면, sig 시그널을 pid로 보낸다. Pid가 0이면 현재 프로세스가 속한 프로세스 그룹의 모든 프로세스에 sig시그널을 보낸다. -1이면 1번 프로세스를 제외한 모든 프로세스에서 sig 시그널을 보낸다. -1보다 작으면 pid프로세스가 포함된 모든 그룹 프로세스에게 시그널을 보낸다. Sig가 0이면 어떤 시그널도 보내지 않지만, 에러 검사는 할 수 있다.

<<반환값>>

성공할 경우 0 을 반환하고 실패했을 경우 -1 을 반환하여, 적당한 errno 값을 설정한다.

<<Sample Code>>

```
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int pid;
    int sig_num;

    // 아규먼트로 pid 번호와
    // 전송할 signal 번호를 받아들여서
```

```
    // 이를 해당 pid 로 보낸다.
    pid = atoi(argv[1]);
    sig_num = atoi(argv[2]);

    kill(pid, sig_num);
}
```

3. get & set process attributes

- getpid() : get process identification
- getppiid() : set/ get process group

pid_t getpid(void);

: 현재 프로세스의 프로세스 id 를 얻는 함수이다. 프로세스 id 는 프로세스가 생성된 순서대로 번호를 순차적으로 할당한다. 먼저 실행된 process 가 종료되었다고 해서 비어있는 프로세스 Id 를 사용하지 않는다. 그리고 할당할 수 있는, 최대 프로세스 id 에 도달하면 다시 처음부터 빈 프로세스 id 를 찾아서 할당해 나간다.

<<Parameter>>

없음

<<반환값>>

현재 프로세스의 Id 를 +값으로 반환한다.

<<Sample Code>>

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("PID = %d\n", getpid());
    printf("부모 PID = %d\n", getpid());
    return 0;
}
```

int setpgid(pid_t *pid*, pid_t *pgid*);

: set process group

<<Parameter>>

pid : 0이면 호출하는 프로세스의 프로세스 id를 반환한다.

Pgid : 0이면 pid에 의해 지정된 프로세스의 Pgid가 프로세스 Id와 동일하게 된

다.

<<반환값>>

성공하면 0 을 반환하고, 실패한 경우 -1 을 반환하고 적당한 errno 을 설정한다.

<<Sample Code>>

```
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
    pid_t pid;
    int p1[2], p2[2];
    char c='?';
    if (pipe(p1) != 0)
        perror("pipe() #1 error");
    else if (pipe(p2) != 0)
        perror("pipe() #2 error");
    else
        if ((pid = fork()) == 0) {
            printf("child's process group id is %d₩n", (int) getpgrp());
            write(p2[1], &c, 1);
            puts("child is waiting for parent to complete task");
            read(p1[0], &c, 1);
            printf("child's process group id is now %d₩n", (int) getpgrp());
```

```
            exit(0);
        }
        else {
            printf("parent's process group id is %d₩n", (int) getpgrp());
            read(p2[0], &c, 1);
            printf("parent is performing setpgid() on pid %d₩n", (int) pid);
            if (setpgid(pid, 0) != 0)
                perror("setpgid() error");
            write(p1[1], &c, 1);
            printf("parent's process group id is now %d₩n", (int) getpgrp());
            sleep(5);
        }
}
```

4. wait for a certain time

- waitid() : wait for process to change state
- epoll_wait() : wait for an I/O event on an epoll file descriptor

pid_t wait(int *_wstatus_);

: 프로세스 종료를 기다린다 주로 fork()를 이용해서 자식 프로세스를 생성했을 때 사용한다. wait를 쓰면 자식 프로세스가 종료할 때 까지 해당 영역에서 부모 프로세스가 sleep()모드로 기다리게 된다. 이는 자식 프로세스와 부모프로세스의 동기화를 위한 목적으로 부모 프로세스가 자식 프로세스보다 먼저 종료되어서 자식 프로세스가 고아 프로세스가 되는 것을 방지하기 위한 목적이다.

<<Parameter>>

status : 자식 프로세스의 상태를 받아온다. 자식 프로세스의 상태 값은 자식 프로세스의 종료 값 256(FF)

<<반환값>>

종료된 자식의 프로세스 Id 는 에러일 경우 -1, 그렇지 않을 경우 0 을 반환한다.

<<Sample Code>>

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
      pid_t cpid, w;
      int status;

    cpid = fork();
      if (cpid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
      }

    if (cpid == 0) {                  /* Code executed by child */
            printf("Child PID is %ld₩n", (long) getpid());
            if (argc == 1)
                  pause();                              /* Wait for signals */
            _exit(atoi(argv[1]));

    } else {                              /* Code executed by parent */
            do {
                  w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
                  if (w == -1) {
                        perror("waitpid");
                        exit(EXIT_FAILURE);
                  }

                if (WIFEXITED(status)) {
                        printf("exited, status=%d₩n", WEXITSTATUS(status));
                  } else if (WIFSIGNALED(status)) {
                        printf("killed by signal %d₩n", WTERMSIG(status));
                  } else if (WIFSTOPPED(status)) {
                        printf("stopped by signal %d₩n", WSTOPSIG(status));
```

```
            } else if (WIFCONTINUED(status)) {
                printf("continued₩n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        exit(EXIT_SUCCESS);
    }
```

int epoll_wait(int *epfd*, struct epoll_event **events*,int *maxevents*, int *timeout*);

: 파일에서 이벤트의 발생을 기다린다.

<<Parameter>>

timeout시간동안 epoll 지정자 epfd에 등록된 파일로 부터 입출력 이벤트가 발생하는지 검사한다. 만약 이벤트가 발생했다면 이벤트가 발생한 파일의 epoll 이벤트 구조체events를 되돌려준다. 다음은 정의된 events구조체이다. timeout는 밀리세컨드 단위이며, -1일 경우 영원히 기다리고, 0일 때는 바로 리턴, 0보다 클때는 timeout밀리초 만큼 이벤트의 발생을 기다린다. 이벤트가 발생하면 epoll_wait(2)는 리턴하며, 이벤트가 발생한 파일의 갯수를 리턴한다. 만약 timeout 시간내에 이벤트가 발생하지 않았다면 0을 리턴한다.

<<반환값>>

성공했을 경우 이벤트가 발생한 파일 지정자의 개수를 return 한다. Time out 시간 동안 이벤트가 발생하지 않았을 경우 0 을, 에러가 발생했을 경우 -1 을 반환한다.

<<Sample Code>>

```
typedef union epoll_data {
                void      *ptr;
                int        fd;
                uint32_t u32;
                uint64_t u64;
            } epoll_data_t;

            struct epoll_event {
                uint32_t      events;     /* Epoll events */
                epoll_data_t data;        /* User data variable */
            };

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

```
#include <sys/types.h>
#include <sys/epoll.h>
#include <sysdep-cancel.h>
int
epoll_wait (int epfd, struct epoll_event *events, int maxevents, int timeout)
{
#ifdef __NR_epoll_wait
  return SYSCALL_CANCEL (epoll_wait, epfd, events, maxevents, timeout);
#else
  return epoll_pwait (epfd, events, maxevents, timeout, NULL);
#endif
}
```

5. wait & signal events

- signal() : ANSI C signal handling
- signalfd() : create a file descriptor for accepting signals

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

: 각 신호에는 전류 처분이 있으며, 이는 프로세스가 신호를 전달할 때 어떻게 동작하는 지를 결정한다.

<<Parameter>>

SIG_IGN

시그널을 무시한다.

그러나 SIGKILL(:12), SIGSTOP(:12) 는 무시할수 없다.

SIG_DFL

시그널의 기본동작을 하도록 한다.

시그널:::핸들러(:12) 함수에 넘겨지는 정수 인자는 시그널의 번호이다.
시그널:::번호(:12)를 넘겨줌으로 인해서 여러개의 시그널에 대해서 하나의 시그널 핸들러를 사용할수 있도록 한다.

SIGKILL, SIGSTOP 시그널에 대해서는 핸들러를 지정할수 없다. 이들 시그널은 무시할수도 없고 핸들러를 지정할수도 없이 단지 기본동작으로만 작동한다.

<<반환값>>

signal 은 이전의 시그널 핸들러의 포인터를 반환하며, 에러시 SIG_ERR 을 반환한다.

<<Sample Code>>

```
#include <signal.h>

#include <unistd.h>

void sig_handler(int signo); // 비프음 발생 함수

int main()
{
    int i = 0;
    signal(SIGINT, (void *)sig_handler);

    while(1)
    {
        printf("%d\n", i);
        i++;
        sleep(1);
    }
    return 1;
}


void sig_handler(int signo)
{
    printf("SIGINT 발생\n");
}
```

int signalfd(int *fd*, const sigset_t **mask*, int *flags*);

: 호출자를 대상으로 하는 명령을 수신하는데 사용할 수 있는 파일 설명자를 생성한다.

<<Parameter>>

mask : 파일 설명자를 통해 발신자가 수신하고자 하는 신호 집합을 지정한다.

Fd : 인수가 -1이면 호출은 새 파일 설명자를 생성하고 마스크에 지정된 신호 세트를 해당 파일 설명자와 연결한다.

Flags : SFD_NONBLOCK | SFD_CLOEXEC

<<반환값>>

signalfd()가 성공하면 signalfd 파일 설명자를 반환한다. 이것은 새로운 파일 설명자(fd 가 -1 이면)이거나 fd 가 유효한 signalfd 파일 설명자라면 fd 이다. 오류 시 -1 이 반환되고 오류는 오류를 나타내도록 설정된다.

<<Sample Code>>

```
#include <sys/signalfd.h>
      #include <signal.h>
      #include <unistd.h>
      #include <stdlib.h>
      #include <stdio.h>

      #define handle_error(msg) \
         do { perror(msg); exit(EXIT_FAILURE); } while (0)

      int
      main(int argc, char *argv[])
      {
         sigset_t mask;
         int sfd;
         struct signalfd_siginfo fdsi;
         ssize_t s;

         sigemptyset(&mask);
         sigaddset(&mask, SIGINT);
         sigaddset(&mask, SIGQUIT);

         /* Block signals so that they aren't handled
            according to their default dispositions */

         if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
            handle_error("sigprocmask");

         sfd = signalfd(-1, &mask, 0);
         if (sfd == -1)
            handle_error("signalfd");

         for (;;) {
            s = read(sfd, &fdsi, sizeof(struct
signalfd_siginfo));
            if (s != sizeof(struct signalfd_siginfo))
               handle_error("read");

            if (fdsi.ssi_signo == SIGINT) {
               printf("Got SIGINT\n");
            } else if (fdsi.ssi_signo == SIGQUIT) {
               printf("Got SIGQUIT\n");
               exit(EXIT_SUCCESS);
            } else {
               printf("Read unexpected signal\n");
            }
```

```
            }
        }
```

6. allocate & free(deallocate) memory

- malloc()
- free()

void *malloc(size_t size);

: 동적 메모리 할당 및 해제. malloc() 함수는 크기 바이트를 할당하고 포인터를 할당된 메모리에 반환한다. 메모리가 초기화되지 않았다. 크기가 0이면 malloc()은 NULL 또는 나중에 free()로 성공적으로 전달할 수 있는 고유한 포인터 값을 반환한다.

<<Parameter>>

size : size만큼 메모리를 동적으로 할당하기 위해서 사용한다.

<<반환값>>

할당된 메모리를 가리키는 포인터를 반환한다. 실패하면 NULL 을 반환한다.

<<Sample Code>>

```
#include <string.h>
#include <stdlib.h>

struct name
{
    int     age;
    char    name[25];
};
int main()
{
    char *org_name;
    struct name *myname;
    int i;

    // org_name 에 char 이 25 만큼 들어갈수 있는
    // 메모리 공간을 할당한다.
    // malloc 는 할당된 메모리 영역의 포인터를 리턴해주므로
    // org_name 은 malloc 를 이용해 할당된 영역의 포인터를 가르키게 된다.
    org_name = (char *)malloc(sizeof(char)*25);

    // myname 역시 마찬가지로 struct name 이 2 개 만큼 들어갈수
    // 있는 메모리 공간을 할당한다.
    myname = (struct name *)malloc(sizeof(myname)*2);
```

```
    strcpy(org_name, "yundream");

    myname[0].age = 25;
    strcpy(myname[0].name, org_name);


    strcpy(org_name, "testname");
    myname[1].age = 28;
    strcpy(myname[1].name, org_name);

    for(i = 0; i < 2; i++)
    {
        printf("%d : %s\n", myname[i].age, myname[i].name);
    }
}
```

int free(void *ptr);

: free() 함수는 ptr가 가리키는 메모리 공간을 자유롭게 해주는데, 이는 malloc(), calloc() 또는 ralloc()에 대한 이전 호출에 의해 반환되었을 것이다. 그렇지 않은 경우 또는 프리(ptr)가 이전에 이미 호출된 경우 정의되지 않은 동작이 발생한다. ptr가 NULL이면 작업이 수행되지 않는다.

<<Parameter>>

ptr : ptr이 가르키는 메모리 영역을 해제한다.

<<반환 값>>

반환 값 없음

<<Sample Code>>

```
#include <stdio.h>
#include<stdlib.h>
main()
{
   int i,j,k, n ;
   int* Array;
   clrscr();
   printf("Enter the number of elements of Array : ");
   scanf("%d", &n );
   Array= (int*) calloc(n, sizeof(int));
   if( Array== (int*)NULL)
     {
         printf("Error. Out of memory.\n");
         exit (0);
      }
       printf("Address of allocated memory= %u\n" , Array);
       printf("Enter the values of %d array elements:", n);
        for (j =0; j<n; j++)
             scanf("%d",&Array[j]);
             printf("Address of Ist member= %u\n", Array);
             printf("Address of 2nd member= %u\n", Array+1);
             printf("Size of Array= %u\n", n* sizeof(int) );
```

```
        for ( i = 0 ; i<n; i++)
          printf("Array[%d] = %d\n", i, *(Array+i));
          free(Array);
          printf("After the action of free() the array elements are:\n");
          for (k =0;k<n; k++)
             printf("Array[%d] = %d\n", k, *(Array+i));
             return 0;
   }
```

File Manipulation

: 파일 생성, 파일 읽기, 파일 쓰기와 같은 파일 조작을 담당

7. create & delete file

- creat()
- unlink()

int creat(const char *pathname, mode_t mode);

: 새로운 파일을 생성하는 함수

<<Parameter>>

- pathname : 만들려는 파일의 이름

- mode : 새 파일의 사용 권한 모드

<<반환값>>

사용되지 않은 `fd` 를 반환하거나 `-1` 이 반환되면 생성 에러.

<<Sample Code>>

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
   char fn[]="creat.file", text[]="This is a test";
   int fd;
```

```
  if ((fd = creat(fn, S_IRUSR | S_IWUSR)) < 0)
    perror("creat() error");
  else {
    write(fd, text, strlen(text));
    close(fd);
    unlink(fn);
  }
return(fd);
}
```

int unlink(const char *pathname);

: 파일 시스템에 있는 파일의 이름을 제거하는 함수

<<Parameter>>

`pathname:` 제거할 파일의 이름

<<반환값>>

성공한 경우 0, 실패한 경우 -1 을 반환한다.
현재 프로세스의 Id 를 +값으로 반환한다.

<<Sample Code>>

```
#include <unistd.h>
int main(int argc, char **argv)
{
    if (access(argv[1], F_OK) !=0 )
    {
        printf("파일 %s 가 존재하지 않습니다₩n", argv[1]);
        exit(0);
    }
    unlink(argv[1]);
    printf("삭제 완료₩n");
    exit(0);
}
```

8. open & close file

Int open(const char *pathname, int flags, mode_t mode);

: 새로운 파일을 열고 그 파일의 Fd를 얻는 데 사용되는 함수

<<Parameter>>

pathname: 오픈할 파일의 절대 경로 혹은 상대 경로

flags : 파일을 열 방법을 지정하는 비트

mode : 열린 파일의 사용 권한을 결정하는 값

<<반환값>>

열린 파일의 Fd 를 반환, 실패한 경우 -1 을 반환한다.

<<Sample Code>>

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

int main()
{
    int filedesc = open("testfile.txt", O_WRONLY | O_APPEND);
    if(filedesc < 0)
        return 1;

    if(write(filedesc,"This will be output to testfile.txt\n", 36) != 36)
    {
        write(2,"There was an error writing to testfile.txt\n");    // strictly
not an error, it is allowable for fewer characters than requested to be
written.
        return 1;
    }

    return 0;
}
```

Int close(int fd);

: 열려있는 파일을 닫는 데 사용되는 함수

<<Parameter>>

fd : 닫을 파일의 Fd

<<반환값>>

성공 한 경우 0 을, 실패한 경우 -1 을 반환한다.

<<Sample Code>>

```
#include <stdlib.h>
#include <fcntl.h>

int main()
```

```
{
    size_t filedesc = open("testfile.txt", O_WRONLY | O_CREAT);
    if(filedesc < 0)
        return 1;

    if(close(filedesc) < 0)
        return 1;

    return 0;
}
```

9. read, write & reposition

- read()
- write()

ssize_t read(int fd, void *buf, size_t count);

: 데이터를 버퍼로 읽는 데 사용되는 함수

<<Parameter>>

fd: socket, open등으로 열린 파일 기술자

buf: fd에 읽을 데이터가 있다면 buf에 담아서 가져온다

count: buf에서 한번에 가져올 데이터의 크기

<<반환값>>

성공할 경우 0이상의 값을 반환한다. 0이라면 파일의 끝을 의미하며, 0보다 큰 양수라면 읽어 들인 buf의 크기를 나타낸다. 에러가 발생할 경우 -1을 반환하며 Errno는 적당한 값으로 설정된다.

<<Sample Code>>

```
#include <unistd.h>

int main()
{
    char data[128];

    if(read(0, data, 128) < 0)
     write(2, "An error occurred in the read.\n", 31);

    exit(0);
}
```

ssize_t write(int fd, const void *buf, size_t count);

: 버퍼에 저장된 데이터를 출력하여 작성하는 함수

<<Parameter>>

fd: socket, open등으로 열린 파일 기술자

buf: 쓰기를 할 데이터 메모리 영역

count: 쓸 데이터의 크기

<<반환값>>

성공할 경우 쓰여진 바이트 만큼이 반환된다. 0이면 쓰여진 것이 없음을 나타내며, -1 일경우는 에러가 발생했을 경우이다. 에러가 발생했을 경우에는 errno 에 적당한 값이 설정된다.

<<Sample Code>>

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

int main(void)
{
    int filedesc = open("testfile.txt", O_WRONLY | O_APPEND);

    if (filedesc < 0) {
        return -1;
    }

    if (write(filedesc, "This will be output to testfile.txt\n", 36) != 36) {
        write(2, "There was an error writing to testfile.txt\n", 43);
        return -1;
    }

    return 0;
}
```

10. get & set file attributes

Int chmod(const char *pathname, mode_t mode);

: path에 의해서 참조된 파일의 권한을 변경하는 데 사용되는 함수

<<Parameter>>

pathname : 참조할 파일의 경로 혹은 이름

mode : 참조된 파일에 변경할 권한

<<반환값>>

성공한 경우 0, 실패한 경우 -1 을 반환한다.

<<Sample Code>>

```
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
   char fn[]="./temp.file";
   FILE *stream;
   struct stat info;

   if ((stream = fopen(fn, "w")) == NULL)
      perror("fopen() error");
   else {
      fclose(stream);
      stat(fn, &info);
      printf("original permissions were: %08x₩n", info.st_mode);
      if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
         perror("chmod() error");
      else {
         stat(fn, &info);
         printf("after chmod(), permissions are: %08x₩n", info.st_mode);
      }
      unlink(fn);
   }
}
```

Int fstat(int fd, struct stat *statbuf);

: fd를 기반으로 파일에 대한 정보를 결정하는 데 사용되는 함수

<<Parameter>>

Fd : 조회할 파일의 FD

Statbuf : 조회한 파일에 대한 정보가 저장되는 구조체

<<반환값>>

실패할 경우 음수를 반환한다.

<<Sample Code>>

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;

    int file=0;
        if((file=open(argv[1],O_RDONLY)) < -1)
            return 1;

    struct stat fileStat;
    if(fstat(file,&fileStat) < 0)
        return 1;

    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n",fileStat.st_size);
    printf("Number of Links: \t%d\n",fileStat.st_nlink);
    printf("File inode: \t\t%d\n",fileStat.st_ino);

    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");

    printf("The file %s a symbolic link\n\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is
not");

    return 0;
}
```

Device Manipulation

: 장치 버퍼에서 읽기, 쓰기 와 같은 장치 조작 담당

11. request & release device

- open() : open and possibly create a file
- creat() : create a file or device

int open(const char *_pathname_, int _flags_);

: open and possibly create a file

<<Parameter>>

pathname: 생성하고자 하는 파일 이름

flag: 파일을 어떠한 mode로 열 것인지 결정하기 위해 사용

읽기 전용, 쓰기 전용, 읽기/쓰기 모드로 열 수 있다.

이들 모드 선택을 위해서 O_RDONLY, O_WRONLY, O_RDWR 이 존재한다

<<반환값>>

에러가 발생하면 -1을 반환하며, 성공했을 경우에는 새로운 파일 지시자를 반환한다. 에러가 발생한 경우에는 적당한 errno값이 설정된다.

<<Sample Code>>

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
// /usr/my.txt 파일을 읽기 전용으로 열고자 할때
fd = open("/usr/my.txt", O_RDONLY);
...
//파일을 찾아오지 못한 경우 에러메시지를 출력한다.
if ( fd == -1){
  fprintf(stderr, “my.txt파일을 open하는 도중 오류 발생 : %s\n”, strerror(errno));
  return -1;
}
```

int creat(const char *_pathname_, mode_t _mode_);

: create a file or device

<<Parameter>>

pathname: 생성하고자 하는 파일 이름

mode: 파일에 대한 access권한을 설정한다.

읽기 전용, 쓰기 전용, 읽기/쓰기 모드로 열 수 있다.

이들 모드 선택을 위해서 O_RDONLY, O_WRONLY, O_RDWR 이 존재한다

<<반환값>>

에러가 발생하면 -1을 반환하며, 성공했을 경우에는 새로운 파일 지시자를 반환한다. 에러가 발생한 경우에는 적당한 errno값이 설정된다.

<<Sample Code>>

```
#include <stdio.h>
#include <fcntl.h>
#define FILEMODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char *argv[])
{
   int fd;
   if(argc < 2){
      printf("useage: %s filepath₩n", argv[0]);
      return 1;
   }
   fd = creat(argv[1], FILEMODE);
   if(fd < 0){
      printf("file open faild.₩n");
```

```
        return 1;
    }
    else{
        printf("%s is opened by %s for write!₩n", argv[1], argv[0]);
    }
    close(fd);
    return 0;
}
```

12. read, write & reposition

- read() : read from a file descriptor
- write() : write to a file descriptor

ssize_t read(int *fd*, void **buf*, size_t *count*);

: open, create, socket등으로 열린 파일로부터 데이터를 읽어 들인다. 파일을 읽으면 size만큼 다음 읽을 위치가 이동된다.

읽을 위치가 파일의 끝에 도달하면 더 이상 읽을 데이터가 없으므로 0을 return한다.

<<Parameter>>

fd: socket, open등으로 열린 파일 기술자

buf: fd에 읽을 데이터가 있다면 buf에 담아서 가져온다

count: buf에서 한번에 가져올 데이터의 크기

<<반환값>>

성공할 경우 0이상의 값을 반환한다. 0이라면 파일의 끝을 의미하며, 0보다 큰 양수라면 읽어 들인 buf의 크기를 나타낸다. 에러가 발생할 경우 -1을 반환하며 Errno는 적당한 값으로 설정된다.

<<Sample Code>>

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#define STDIN 1
int main() {
	char buf[80];
	memset(buf, 0x00, 80);
	if (read(STDIN, buf, 80) < 0) {
		perror("read erro : ");
		exit(0);
	}
	printf("%s", buf);
}
//키보드로 부터 읽어들인 문자열을 화면에 출력한다.
```

ssize_t write(int *fd*, const void **buf*, size_t *count*);

: open, create등으로 생성한 파일로 데이터 쓰기 또는 전송한다.

파일에 쓰기를 하면 파일의 쓰기 또는 읽기 위치가 쓴 size만큼 뒤로 이동한다. -1이 return되지 않는 이상 count만큼 write된다.

<<Parameter>>

fd: socket, open등으로 열린 파일 기술자

buf: 쓰기를 할 데이터 메모리 영역

count: 쓸 데이터의 크기

<<반환값>>

성공할 경우 쓰여진 바이트 만큼이 반환된다. 0이면 쓰여진 것이 없음을 나타내며, -1 일경우는 에러가 발생했을 경우이다. 에러가 발생했을 경우에는 errno 에 적당한 값이 설정된다.

<<Sample Code>>

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
struct data
{
    int         age;
    char        name[25];
};
int main()
{
    int fd;
    int n;
    struct data mydata, readdata;
    fd = open("data.txt", O_CREAT|O_RDWR);
    if (fd == -1){
        perror("open error : ");
    }
    mydata.age = 25;
    strcpy(mydata.name, "hello");
    n = write(fd, (void *)&mydata, sizeof(mydata));
    close(fd);
}
```

이 예제는 data.txt 라는 파일을 일기/쓰기 모드로 open 한 다음에 data 구조체를 파일에 적는 프로그램이다.

13. get & set device attributes

- ioctl() : control device
- ioperm() : set port input/output permissions

int ioctl(int *fd*, unsigned long *request*, ...);

: 특수 파일의 기본 장치 매개 변수를 조작한다.

<<Parameter>>

fd: socket, open등으로 열린 파일 기술자

request: ioctl에 의해 불리는 함수의 인덱스

ioctl로 불리는 함수는 switch문과 같은 것을 이용해 request로 전달된 값을 비교해 해당 함수를 다시 호출해 주게 된다.

<<반환값>>

성공할 경우 0이 반환된다. 에러가 발생하면 -1 을 반환하고, errno에 적당한 값을 설정한다.

<<Sample Code>>

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <fcntl.h>

int main()
{
	int fd;
	char buf[256];
	fd = open("/dev/hdd_info", O_RDWR);
	if (fd < 0)
	{
		printf("Device open error.\n");
		return -1;
```

```
        }
        ioctl(fd, 0, buf);
        printf("buf : %s\n", buf);
        close(fd);
        return 0;
}
```

open 후 ioctl의 0번 함수를 호출해 hddinfo.c의 read_serial()을 불러 하드디스크의 시리얼 번호를 읽어온다. 하드디스크의 시리얼 번호는 커널 부팅할 때 이미 얻어진 하드디스크에 대한 정보를 갖고 있는 구조체에서 복사한다.

int ioperm(unsigned long *from*, unsigned long *num*, int *turn_on*);

: 포트의 입출력 허가권을 설정한다. 프로세스를 위해 포트 주소 from을 시작으로 num바이트를 turn_on값으로 포트 접근 허가 비트를 설정한다.

<<Parameter>>

from: 포트를 읽거나 쓴다.

num: num byte

turn_on: 허가 비트. 0이 아닌 경우 허가, 0이면 불가

<<반환값>>

성공할 경우 0이 반환된다. 에러가 발생하면 -1 을 반환하고, errno에 적당한 값을 설정한다.

<<Sample Code>>

```
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378 /* lp1 */

int main()
{
  /* Get access to the ports */
```

```
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* Set the data signals (D0-7) of the port to all low (0) */
  outb(0, BASEPORT);

  /* Sleep for a while (100 ms) */
  usleep(100000);

  /* Read from the status port (BASE+1) and display the result */
  printf("status: %d\n", inb(BASEPORT + 1));

  /* We don't need the ports anymore */
  if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}
```

14. logically attach & detach devices

- mmap() : map files or devices into memory
- munmap() : unmap files or devices into memory

void *mmap(void *_addr_, size_t _length_, int _prot_, int _flags_,

int _fd_, off_t _offset_);

: 파일이나 장치를 메모리에 대응시키거나 푼다.Mmap은 지정된 영역이 대응된 실제 시작 위치를 반환한다.

<<Parameter>>

fd: 지정 파일이나 객체의 상태를 받아옴

length: length byte

port: 메모리 보호모드의 상태. PROT_EXEC,PROT_READ…

flags: 대응된 객체의 타입, 대응 옵션, 대응된 페이지 복사본에 대한 수정의 공유 여부 설정.

offset: mapping할 때 length의 시작점을 지정

<<반환값>>

성공할 경우 대응된 영역의 포인터를 반환한다. 실패하면 -1을 반환하고, errno는 적당한 값으로 설정한다.

<<Sample Code>>

```
#include <stdio.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 argc, char **argv){
   int fd;
   char *file = NULL;
   struct stat sb;
   char buf[80] ={0x00,};
   int flag = PROT_WRITE | PROT_READ;
   if (argc < 2){
      fprintf(stderr, "Usage: input\n");
      exit(1);
   }
   if ((fd = open(argv[1], O_RDWR|O_CREAT)) < 0){
      perror("File Open Error");
      exit(1);
   }
   if (fstat(fd, &sb) < 0){
      perror("fstat error");
      exit(1);
   }
   file = (char *)malloc(40);
if ((file =
      (char *) mmap(0, 40, flag, MAP_SHARED, fd, 0)) == -1){
```

```
        perror("mmap error");
        exit(1);
    }
    printf("%s\n", file);
    memset(file, 0x00, 40);
    mnumap(file);
    close(fd);
}
```

mmap 를 이용해서 열린 파일을 메모리에 대응시킨다.
file 은 대응된 주소를 가리키고, file 을 이용해서 필요한 작업을 하면 된다.

int munmap(void *addr, size_t length);

: 지정된 주소 범위에 대한 매핑을 삭제한다. 프로세스가 종료되면 영역이 자동으로 매핑 해제된다.

<<Parameter>>

addr: the address that must be a multiple of the page size.

length: length byte

<<반환값>>

성공할 경우 0을 반환한다. 실패하면 -1을 반환하고, errno는 적당한 값으로 설정한다.

<<Sample Code>>

```
int main (int argc, char *argv[]) {
int fdin, fdout; char *src, *dst; struct stat statbuf;
if (argc != 3) {
printf("usage: a.out \n"); return -1;
}
/* open the input file */
if ((fdin = open (argv[1], O_RDONLY)) < 0) {
printf ("can't open %s for reading", argv[1]); return -2;
}
/* open/create the output file */
if ((fdout = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR)) < 0) {
printf ("can't create %s for writing", argv[2]); return -2;
```

```
}
/* find size of input file */
if (fstat (fdin,&statbuf) < 0) {
printf ("fstat error"); return -2;
}
/* go to the location corresponding to the last byte */
if (lseek (fdout, statbuf.st_size - 1, SEEK_SET) == -1) {
printf ("lseek error"); return -2;
}
/* write a dummy byte at the last location */
if (write (fdout, "", 1) != 1) {
printf ("write error"); return -2;
}
/* mmap the input file */
if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED, fdin, 0)) == (caddr_t)
-1) {
printf ("mmap error for input"); return -2;
}
/* mmap the output file */
if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fdout,
0)) == (caddr_t) -1) {
printf ("mmap error for output"); return -2;
}
/* this copies the input file to the output file */
memcpy (dst, src, statbuf.st_size);
munmap(src, statbuf.st_size);
munmap(dst, statbuf.st_size);
return 0;
}
```

Information Maintenance

: 정보와 운영체제와 사용자 프로그램 간의 전송을 처리

15. get & set time & date

- time() : get time in seconds
- gettimeofday() : get time

time_t time(time_t *tloc);

: 초 단위로 시간을 가져온다. 이 함수가 돌려주는 시간은 1970년 1월 1일 00:00:00 부터 지금까지의 시간을 초단위로 환산한 것이다.

<<Parameter>>

tloc : tloc에 들어가는 값이 NULL이 아니라면 반환값은 t가 가르키는 메모리에 저장된다.

<<반환값>>

성공하면 현재까지의 흐른 시간을 초단위로 반환한다. 에러가 발생할 경우 -1 을 반환한다.

<<Sample Code>>

```
#include <time.h>

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

void swaptime(time_t, char *);
int main()
{
    int     i;
    time_t  the_time;
    char buffer[255];

    // 현재 TIME 을 구한다.
    time(&the_time);

    printf("현재 시간은 %d 초\n", the_time);

    // 보기좋은 지역시간대로 바꾼다.
    swaptime(the_time, buffer);
    printf("%s\n", buffer);
}

void swaptime(time_t org_time, char *time_str)
{
    struct tm *tm_ptr;
    tm_ptr = localtime(&org_time);


    sprintf(time_str, "%d/%d/%d %d:%d:%d",
```

```
                    tm_ptr->tm_year+1900,
                    tm_ptr->tm_mon+1,
                    tm_ptr->tm_mday,
                    tm_ptr->tm_hour,
                    tm_ptr->tm_min,
                    tm_ptr->tm_sec);
}
```

int gettimeofday(struct timeval *tv, struct timezone *tz);

: 현재 시간을 가져오고 시스템의 시간 값을 설정한다. Time()과 비슷하지만 마이크로 초 단위까지의 시간 까지 되돌려준다.

<<Parameter>>

tv : 현재 시스템 시간을 저장하기 위한 구조체

```
//tv 구조체 정의
struct timeval
{
    long tv_sec;       // 초
    long tv_usec;      // 마이크로초
}
```

tz : 타임존을 설정하기 위해서 사용

```
//tz
struct timezone
{
    int tz_minuteswest:  // 그리니치 서측분차
    int tz_dsttime       // DST 보정타입(일광 절약시간)
}
```

<<반환값>>

성공한다면 0 을 반환하고, 실패한다면 -1 을 반환한다.

<<Sample Code>>

```
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main()
{
    struct timeval mytime;

    // 현재 시간을 얻어온다.
    gettimeofday(&mytime, NULL);
```

```
    printf("%ld:%ld\n", mytime.tv_sec, mytime.tv_usec);

    // 시간을 1 시간 뒤로 되돌려서 설정한다.
    mytime.tv_sec -= 3600;
    settimeofday(&mytime, NULL);
    return 0;
}
```

16. get & set system data

- stime() : set time
- gettrlimit() : get resource limits

int stime(const time_t *t);

: 시스템의 시간과 날짜를 설정한다. 이 함수는 단지 root유저만이 사용할 수 있다.

<<Parameter>>

t : 설정할 시간

<<반환값>>

성공할 경우 0 을 실패했을 경우에는 -1 을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

```
#include <time.h>

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    // 사용법 : ./stime "2002 2 16 9 9 9"
    //                  년  월 일 시 분 초
    char buff[6][5];
    struct tm tm_ptr;
    time_t m_time;

    sscanf(argv[1], "%s %s %s %s %s %s", buff[0], buff[1],
                                    buff[2], buff[3],
                                    buff[4], buff[5]);

    tm_ptr.tm_year  = atoi(buff[0]) - 1900;
    tm_ptr.tm_mon   = atoi(buff[1]) - 1;
    tm_ptr.tm_mday  = atoi(buff[2]);
```

```
    tm_ptr.tm_hour  = atoi(buff[3]);
    tm_ptr.tm_min   = atoi(buff[4]);
    tm_ptr.tm_sec   = atoi(buff[5]);
    tm_ptr.tm_isdst = 0;

    m_time = mktime(&tm_ptr);
    stime(&m_time);

}
```

int getrlimit(int *resource*, struct rlimit **rlim*);

: 시스템 자원(resource)의 값을 얻어오고 값을 설정한다.

<<Parameter>>

resource : 자원의 종류를 지정

rlim : 리소스의 크기

```
struct rlimit
{
    rlim_t rlim_cur;   /* soft limit */
    rlim_t rlim_max;   /* Hard limit */
};
```

<<반환값>>

성공할경우 0 을 실패했을경우에는 -1 을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    struct rlimit rlim;

    // 생성가능한 프로세스의 갯수를 출력한다. (현재 : 최대)
    getrlimit(RLIMIT_NPROC, &rlim);
    printf("PROC MAX : %lu : %lu\n", rlim.rlim_cur,
rlim.rlim_max);

    // 오픈가능한 파일의 갯수를 출력한다.
    getrlimit(RLIMIT_NOFILE, &rlim);
    printf("FILE MAX : %lu : %lu\n", rlim.rlim_cur,
rlim.rlim_max);
```

```
    // 사용가능한 CPU 자원을 출력한다.
    getrlimit(RLIMIT_CPU, &rlim);

    // 만약 무한대로 사용가능하다면 UNLIMIT 를 출력하도록한다.
    // CPU 자원은 최대한 사용가능하도록 되어있음으로 UNLIMIT 를
출력할것이다.
    if(rlim.rlim_cur == RLIM_INFINITY)
    {
        printf("UNLIMIT\n");
    }
}
```

17. get & set process, file & device attributes

- chown() : change ownership of a file
- utime() : change file last access and modification times

int chown(const char *_pathname_, uid_t _owner_, gid_t _group_);

: 파일에 대한 소유권을 바꾸기 위해서 사용된다. 유하산 류의 함수로 fchown(), lchown() 함수들이 있다.

<<Parameter>>

pathname : 파일이나 디바이스의 경로

owner / group : -1일 경우 ID가 바뀌지 않는다.

<<반환값>>

성공할 경우 0 을 실패했을 경우에는 -1 을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

/home/test/my.txt 파일의 유저권한을 root 그룹권한을 post 로 변경하기 원할때

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <sys/types.h>

int main()
{
    struct passwd *u_info;
    struct group  *g_info;

    u_info = getpwnam("root");
```

```
    g_info = getgrnam("post");

    chown("/home/test/my.txt", u_info->pw_uid, g_info->gr_gid);
}
```

int utime(const char *filename, const struct utimbuf *times);

: 파일에 대한 접근/수정 시간을 변경

<<Parameter>>

filename : 변경할 파일 이름

times : 변경할 시간

<<반환값>>

성공할 경우 0을 실패했을 경우에는 -1을 반환하며, 적당한 errno 값을 설정한다

<<Sample Code>>

```
#include <sys/types.h>
#include <utime.h>
#include <sys/time.h>
#include <stdio.h>
#include <string.h>

int main()
{
    struct utimbuf ubuf;
    ubuf.actime = time((time_t *)0);
    ubuf.modtime = time((time_t *)0);

    // 접근,수정 시간을 현재 시간으로 변경한다.
    utime("sizeof.c", NULL);

    // NULL 대신 actime,modtime 을 세팅해서
    // 직접 값을 지정해줄수도 있다.
    utime("sizeof.c", &ubuf);
}
```

Communications

: 프로세스 간 통신 관리, 통신 연결을 만들고 삭제하는 작업을 처리

18. create & delete communication connection

- socket() : create an endpoint for communication
- connect() : initiate a connection on a socket

int socket(int *domain*, int *type*, int *protocol*);

: 통신을 위한 endpoint를 생성하기 위해서 사용한다.

<<Parameter>>

domain : 통신 영역을 지정하기 위해서 사용한다.

<<반환값>>

성공 할 경우 정수의 파일 지정 번호를 반환하고, 실패 할 경우 -1 을 반환한다.

<<Sample Code>>

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    int server_sockfd, client_sockfd;
    int state, client_len;

    struct sockaddr_in clientaddr, serveraddr, myaddr;


    // internet 기반의 스트림 소켓을 만들도록 한다.
    // server_sockfd 는 endpoint 소켓(듣기 소켓) 으로 사용된다.
    if ((server_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket error : ");
        exit(0);
    }
    bzero(&serveraddr, sizeof(serveraddr));
```

```
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(12345);

    // 듣기 소켓에 소켓특성(주소, 포트)를 묶어준다.
    state = bind(server_sockfd , (struct sockaddr *)&serveraddr,
                sizeof(serveraddr));
    if (state == -1)
    {
        perror("bind error : ");
        exit(0);
    }

    state = listen(server_sockfd, 5);
    if (state == -1)
    {
        perror("listen error : ");
        exit(0);
    }

    // 만약 듣기 소켓으로 클라이언트 연결이 들어왔다면,
    // 새로운 클라이언트와의 통신을 위한 소켓 지정번호를 할당한다.
    client_sockfd = accept(server_sockfd, (struct sockaddr *)&clientaddr,
                                    &client_len);
    if (client_sockfd == -1)
    {
        perror("Accept error : ");
        exit(0);
    }
    // 여기에서 부터는 클라이언트와의 통신
    // .....
    // .....

    close(client_sockfd);
    return 1;
}
```

int connect(int *sockfd*, const struct sockaddr **addr*, socklen_t *addrlen*);

: 소켓으로 연결을 시작한다.

<<Parameter>>

sockfd : 연결되지 않은 소켓 기술자

addr : 연결 정보를 담고있는 sockaddr구조체의 포인터

addrlen : sockaddr구조체 포인터가 가리키는 데이터의 크기

<<반환값>>

연결 또는 바인딩이 성공하면 0이 반환된다. 실패할 경우 -1을 반환한다.

<<Sample Code>>

```
#include <sys/stat.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
   struct sockaddr_in serveraddr;
   int server_sockfd;
   int client_len;
   char buf[80];
   char rbuf[80];


   if ((server_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      perror("error :");
      exit(0);
   }

   server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
   serveraddr.sin_family = AF_INET;
   serveraddr.sin_addr.s_addr = inet_addr("218.234.19.87");
   serveraddr.sin_port = htons(atoi(argv[1]));

   client_len = sizeof(serveraddr);

   if (connect(server_sockfd, (struct sockaddr *)&serveraddr,
client_len) < 0)
   {
      perror("connect error :");
      exit(0);
   }

   memset(buf, 0x00, 80);
   read(0, buf, 80);
   if (write(server_sockfd, buf, 80) <= 0)
   {
      perror("write error : ");
      exit(0);
```

```
    }
    memset(buf, 0x00, 80);
    if (read(server_sockfd, buf, 80) <= 0)
    {
        perror("read error : ");
        exit(0);
    }
    close(server_sockfd);
    printf("read : %s", buf);
}
```

19. send & receive messages

- recvfrom() : receive a message from a socket
- recvmsg() : receive a message from a socket

ssize_t recvfrom(int *sockfd*, void **buf*, size_t *len*, int *flags*, struct sockaddr *src_addr*, socklen_t *addrlen*);

: 소켓으로부터 메시지를 읽어들인다.

<<Parameter>>

sockfd : 소켓지정자

buf : 데이터를 읽어들이는 버퍼

len : 읽어 들일 데이터의 최대 크기 지정

flags : 선택 비트

from : 메시지의 원주소를 나타냄

fromlen : 주소 구조체의 크기

<<반환값>>

성공할 경우 0 을 실패했을 경우에는 -1 을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

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

#include <sys/stat.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
```

```
int main(int argc, char **argv)
{
    int server_sockfd, client_sockfd;
    int client_len, n;
    char buf[80];
    struct sockaddr_in clientaddr, serveraddr;

    client_len = sizeof(clientaddr);

    if ((server_sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket error : ");
        exit(0);
    }
    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(atoi(argv[1]));

    bind (server_sockfd, (struct sockaddr *)&serveraddr,
sizeof(serveraddr));
    listen(server_sockfd, 5);

    while(1)
    {
        memset(buf, 0x00, 80);
        client_sockfd = accept(server_sockfd, (struct sockaddr
*)&clientaddr,
                           &client_len);

        if ((n = recvfrom(client_sockfd, buf, 80, 0, NULL,
&client_len)) <= 0)
        {
            close(client_sockfd);
            continue;
        }
        sendto(client_sockfd,(void *)buf, 80, 0,
                      NULL, client_len);
        close(client_sockfd);
    }
}
```

ssize_t recvmsg(int *sockfd*, struct msghdr **msg*, int *flags*);

: 소켓으로부터 메시지를 받는다.

<<Parameter>>

sockfd : 소켓지정자

msg : 받을 메시지

flags : 선택 비트

<<반환값>>

성공할 경우 0을 실패했을 경우에는 -1을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

```
int fd;                  // = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
char buf[MAX_BUF];
struct msghdr msg;       // basic msg header
struct iovec iov;       // i/o vercor (about buffer information)
cmsg_fd cmsg;            // control message

// register buffer
iov.iov_base = buf;
iov.iov_len   = MAX_BUF;

// configure msg
msg.msg_name           = NULL; // optional address (= sockaddr_in)
msg.msg_namelen       = 0;      // address size
msg.msg_iov            = &iov; // iov that registered buffer
msg.msg_control       = &cmsg;
msg.msg_controllen = sizeof(cmsg);
msg.msg_flags         = 0;

// configure control msg
struct cmsghdr* cptr(CMSG_FIRSTHDR(&msg));
cptr->cmsg_len     = CMSG_LEN(sizeof(int));
cptr->cmsg_level = SOL_SOCKET;
cptr->cmsg_type    = SCM_RIGHTS;

memcpy(CMSG_DATA(cptr), &fd, sizeof(int));
if(sendmsg(sockfd, &msg, 0) < 0) {
      perror("[-] Failed sendmsg()");
      exit(-1);
}
```

20. transfer status information

- semget() : get a System V semaphore set identifier
- semctl() : System V semaphore control operations

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

: 세마포어 설정을 확인한다.

<<Parameter>>

key : 키 값과 일치하는 세마포어 설정자를 반환

nsems : 세마포어를 생성할 것인지 아니면 이미 만들어진 세마포어에 접근할 것인지를 결정하기 위해서 사용한다.

semflg : 세마포어 생성 특성을 결정하기 위해서 사용한다.

<<반환값>>

성공할 경우 세모포어 설정 확인자(양의 정수)이며, 그렇지 않을 경우 -1 이 반환되고 errno 가 설정된다.

<<Sample Code>>

```
#include <errno.h>

int main()
{
    int semid;
    int status;

    if ((semid = semget(12345, 1, IPC_CREAT|0666)) == -1)
    {
        perror("semget error ");
        return 1;
    }

    printf("success semid is %d : %d %d\n", status, semid,
errno);
}
```

int semctl(int *semid*, int *semnum*, int *cmd*, ...);

: 시스템 V 세마포어 제어 작업을 한다

<<Parameter>>

semid : semget함수가 return한 System V semaphore set ID

semnum : semaphore set의 배열의 처리할 Index

cmd : 처리할 명령어 값

<<반환값>>

성공할 경우 -1 이 아닌 값을 반환하고, 실패한 경우 -1 을 반환한다.

<<Sample Code>>

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

#define SEMKEY 2345

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

static int   semid;
int main(int argc, char **argv)
{
	FILE* fp;
	char buf[11];
	char count[11];

	union semun sem_union;

	// open 과 close 를 위한 sembuf 구조체를 정의한다.
	struct sembuf mysem_open   = {0, -1, SEM_UNDO}; // 세마포어 얻기
	struct sembuf mysem_close = {0, 1, SEM_UNDO};   // 세마포어 돌려주기
	int sem_num;

	memset(buf, 0x00, 11);
	memset(count, 0x00, 11);

	// 아규먼트가 있으면 생성자
	// 그렇지 않으면 소비자이다.
	if (argc > 1)
		sem_num = 1;
	else
```

```
        sem_num = 0;

    // 세마포설정을 한다.
    semid = semget((key_t)234, sem_num, 0660|IPC_CREAT);
    if (semid == -1)
    {
        perror("semget error ");
        exit(0);
    }

// 세마포어 초기화
        sem_union.val = 1;
        if ( -1 == semctl( semid, 0, SETVAL, sem_union))
        {
                printf( "semctl()-SETVAL 실행 오류₩n");
                return -1;
        }
    // counter.txt 파일을 열기 위해서 세마포어검사를한다.
    if(semop(semid, &mysem_open, 1) == -1)
    {
        perror("semop error ");
        exit(0);
    }

    if ((fp = fopen("counter.txt", "r+")) == NULL)
    {
        perror("fopen error ");
        exit(0);
    }
    // 파일의 내용을 읽은후 파일을 처음으로 되돌린다.
    fgets(buf, 11, fp);
    rewind(fp);

    // 개행문자를 제거한다.
    buf[strlen(buf) - 1] = 0x00;

    sprintf(count, "%d₩n", atoi(buf) + 1);
    printf("%s", count);
    // 10 초를 잠들고 난후 count 를 파일에 쓴다.
    sleep(10);
    fputs(count,fp);
```

```
    fclose(fp);
    // 모든 작업을 마쳤다면 세마포어 자원을 되돌려준다
    semop(semid, &mysem_close, 1);
    return 1;
}
```

21. attach or detach remote devices

- pipe() : create pipe
- shmget() : allocates a System V shared memory segment

int pipe(int *pipefd*[2]);

: 파이프를 생성한다. pipe 를 이용하면 2개의 파일 지시자를 생성할 수 있다. 2개가 생성되는 이유는 읽기 전용과 쓰기 전용의 파이프를 생성하기 위함이다.

<<Parameter>>

pipefd : 읽기 전용인지 쓰기 전용인지

<<반환값>>

성공할 경우 0 을 실패했을 경우에는 -1 을 반환하며, 적당한 errno 값을 설정한다.

<<Sample Code>>

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

int main()
{
 int n, fd[2];
 char buf[255];
 int pid;

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

 // 파이프를 생성한다.
 if ((pid = fork()) < 0)
 {
 perror("fork error : ");
 exit(0);
 }
```

```
// 만약 자식프로세스라면 파이프에 자신의 PID(:12) 정보를 쓴다.
else if (pid == 0)
{
close(fd[0]);
while(1)
{
memset(buf, 0x00, 255);
sprintf(buf, "Hello : %d\n", getpid());
write(fd[1], buf, strlen(buf));
sleep(1);
}
}

// 만약 부모프로세스(:12)라면 파이프(:12)에서 데이타를 읽어들인다.
else
{
close(fd[1]);
while(1)
{
memset(buf, 0x00, 255);
n = read(fd[0], buf, 255);
fprintf(stderr, "%s", buf);
}
}
}
```

위 프로그램은 파이프를 생성한 후 만들어진 파이프를 통해서 자식과 부모가 서로 통신하는 예제이다. fork 하기 전에 pipe 를 만들면 된다. fork() 는 특성상 열린 파일 지시자를 자식에게 상속하기 때문이다.

int shmget(key_t *key*, size_t *size*, int *shmflg*);

: 공유 메모리 영역을 할당한다.

<<Parameter>>

key : 접근 번호

size : 생성될 공유 메모리 공간의 크기

shmflg : 공간 할당의 모드 비트

<<반환값>>

성공하면 **shmid** 를 반환, 실패하면 -1 을 반환한다.

<<Sample Code>>

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

```
#include <unistd.h>


int main()
{
	int shmid;
	int pid;

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


	// 공유메모리 공간을 만든다.
	shmid = shmget((key_t)1234, sizeof(int), 0666|IPC_CREAT);

	if (shmid == -1)
	{
		perror("shmget failed : ");
		exit(0);
	}

	// 공유메모리를 사용하기 위해 프로세스메모리에 붙인다.
	shared_memory = shmat(shmid, (void *)0, 0);
	if (shared_memory == (void *)-1)
	{
		perror("shmat failed : ");
		exit(0);
	}

	cal_num = (int *)shared_memory;
	pid = fork();
	if (pid == 0)
	{
		shmid = shmget((key_t)1234, sizeof(int), 0);
		if (shmid == -1)
		{
			perror("shmget failed : ");
			exit(0);
		}
		shared_memory = shmat(shmid, (void *)0, 0666|IPC_CREAT);
		if (shared_memory == (void *)-1)
		{
```

```
            perror("shmat failed : ");
            exit(0);
        }
        cal_num = (int *)shared_memory;
        *cal_num = 1;

        while(1)
        {
            *cal_num = *cal_num + 1;
            printf("child %d₩n", *cal_num);
            sleep(1);
        }
    }

    // 부모 프로세스로 공유메모리의 내용을 보여준다.
    else if(pid > 0)
    {
        while(1)
        {
            sleep(1);
            printf("%d₩n", *cal_num);
        }
    }
}
```