

TI DSP, MCU 및 Xilinx Zynq FPGA

프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – 문한나

mhn97@naver.com

내용정리

dup()는 파일의 디스크립터를 복제하는 함수이다

예제1)

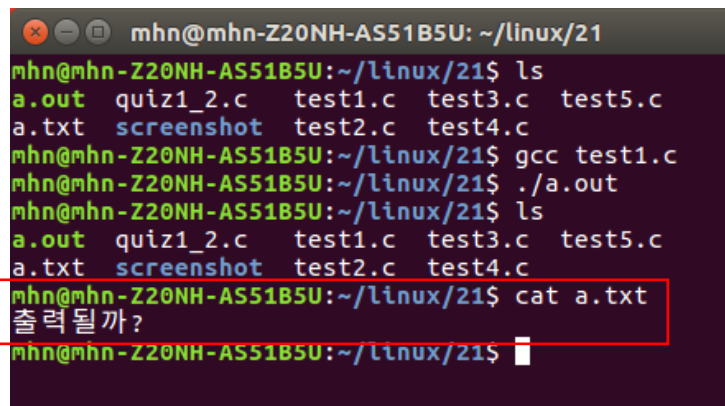
```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(void){

    int fd;
    fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    close(1); //표준출력을 닫았으므로 출력이 안된다
    dup(fd); //dup 함수로 1번을 fd(파일)로 대체한다
    printf("출력될까? \n"); //따라서 출력을 하면 파일에 입력된다

    return 0;

}
```



```
mhn@mhn-Z20NH-AS51B5U: ~/linux/21
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ls
a.out  quiz1_2.c  test1.c  test3.c  test5.c
a.txt  screenshot test2.c  test4.c
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ gcc test1.c
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ./a.out
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ls
a.out  quiz1_2.c  test1.c  test3.c  test5.c
a.txt  screenshot test2.c  test4.c
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ cat a.txt
출력될까?
mhn@mhn-Z20NH-AS51B5U:~/linux/21$
```

예제2)

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

int main(void){

    int fd;
    char buff[1024];
    fd = open("a.txt",O_RDONLY);
    close(0); //표준입력을 닫았다
    dup(fd); //dup 함수로 입력을 fd(파일)로 대체한다
    gets(buff); //gets 는 입력함수로 파일의 내용이 버퍼에 저장된다
    printf("출력될까? \n");
    printf("%s",buff); //그래서 버퍼를 찍으면 복사된 파일 내용이 찍히게 된다.

    return 0;

}
```

```
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ gcc test2.c
test2.c: In function 'main':
test2.c:12:2: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
    gets(buff); //파일에서 입력을 받음
    ^
/tmp/ccuqN82N.o: In function `main':
test2.c:(.text+0x5b): warning: the `gets' function is dangerous and should not be used.
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ./a.out
출력될까?
출력될까? mhn@mhn-Z20NH-AS51B5U:~/linux/21$
```

```

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

int main(int argc, char *argv[]){

    int i;
    char ch = 'a';

    int fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    //lseek()로 511바이트만큼 이동한 후 그 이동된 곳으로 포인터를 변경한다
    lseek(fd, 512-1, SEEK_SET);
    write(fd, &ch, 1); //표준입력으로 'a'를 fd에 담는다
    close(fd); //파일을 닫는다.
    return 0;

}

```

따라서 512바이트번째에 a가 담겨있다.

이 값은 xxd로 확인이 가능하다

```

mhn@mhn-Z20NH-AS51B5U: ~/linux/21
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000150: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000160: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000170: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000180: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000190: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001f0: 0000 0000 0000 0000 0000 0000 0061 .....a
mhn@mhn-Z20NH-AS51B5U: ~/linux/21$

```

명령어 **ps -ef**는 지금 돌아가고 있는 프로세스들을 보여준다

```
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ps -ef
UID          PID    PPID  C  STIME TTY          TIME CMD
root           1        0  0  08:51 ?        00:00:01 /sbin/init splash
root           2        0  0  08:51 ?        00:00:00 [kthreadd]
root           4        2  0  08:51 ?        00:00:00 [kworker/0:0H]
root           6        2  0  08:51 ?        00:00:00 [mm_percpu_wq]
root           7        2  0  08:51 ?        00:00:00 [ksoftirqd/0]
root           8        2  0  08:51 ?        00:00:01 [rcu_sched]
root           9        2  0  08:51 ?        00:00:00 [rcu_bh]
root          10        2  0  08:51 ?        00:00:00 [migration/0]
root          11        2  0  08:51 ?        00:00:00 [watchdog/0]
root          12        2  0  08:51 ?        00:00:00 [cpuhp/0]
root          13        2  0  08:51 ?        00:00:00 [cpuhp/1]
root          14        2  0  08:51 ?        00:00:00 [watchdog/1]
root          15        2  0  08:51 ?        00:00:00 [migration/1]
root          16        2  0  08:51 ?        00:00:00 [ksoftirqd/1]
root          18        2  0  08:51 ?        00:00:00 [kworker/1:0H]
root          19        2  0  08:51 ?        00:00:00 [cpuhp/2]
root          20        2  0  08:51 ?        00:00:00 [watchdog/2]
root          21        2  0  08:51 ?        00:00:00 [migration/2]
root          22        2  0  08:51 ?        00:00:00 [ksoftirqd/2]
root          24        2  0  08:51 ?        00:00:00 [kworker/2:0H]
root          25        2  0  08:51 ?        00:00:00 [cpuhp/3]
root          26        2  0  08:51 ?        00:00:00 [watchdog/3]
root          27        2  0  08:51 ?        00:00:00 [migration/3]
root          28        2  0  08:51 ?        00:00:00 [ksoftirqd/3]
root          30        2  0  08:51 ?        00:00:00 [kworker/3:0H]
root          31        2  0  08:51 ?        00:00:00 [kdevtmpfs]
root          32        2  0  08:51 ?        00:00:00 [netns]
```

명령어 **ps -ef | grep bash** 는 우리가 실행시킨 터미널을 보여준다
(실행시킨 터미널(2)+찾는 프로세스(1))

```
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ps -ef | grep bash
mhn      3489   3482  0  09:00 pts/4    00:00:00 bash
mhn      5142   3482  0  11:58 pts/12    00:00:00 bash
mhn      6378   5142  0  13:33 pts/12    00:00:00 grep --color=auto bash
```

명령어 **ps -ef | grep bash | grep -v grep** 은 실제 돌아가고 있는 터미널만 보여준다
(찾는 프로세스는 제외한다)

```
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ps -ef | grep bash | grep -v grep
mhn      3489   3482  0  09:00 pts/4    00:00:00 bash
mhn      5142   3482  0  11:58 pts/12    00:00:00 bash
```

명령어 **ps -ef | grep bash | grep -v grep | awk '{print \$2}'** 는 PID 를 보여준다
PID 란? 프로세스의 고유식별 번호이다.

```
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ps -ef | grep bash | grep -v grep | awk '{print $2}'
3489
5142
```

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

```
int main(void){
```

```
    int fd,ret;
```

```
    char buf[1024];
```

```
    mkfifo("myfifo");
```

```
    fd = open("myfifo",O_RDWR);
```

```
    for(;;){
```

```
        ret = read(0,buf,sizeof(buf)); //버퍼만큼 읽는다
```

```
        //이 작업을 해주는 이유는 buf의 ret-1위치에 엔터값까지 들어있기 때문이다
```

```
        //입력을 할 때 엔터까지 같이 저장되므로 이것 이전부터 출력을 하기 위함이다
```

```
        buf[ret-1]=0;
```

```
        printf("keyboard input : [%s]\n",buf);
```

```
        read(fd,buf,sizeof(buf));
```

```
        buf[ret-1]=0;
```

```
        printf("Pipe input : [%s]\n",buf);
```

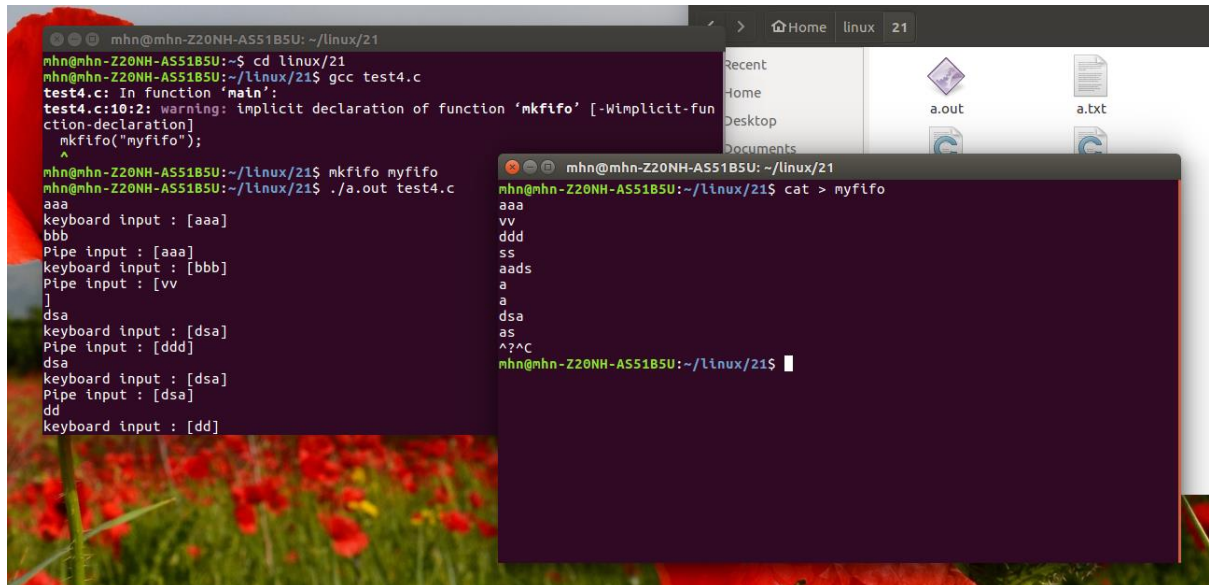
```
    }
```

```
    return 0;
```

```
}
```

read는 block함수로 먼저 입력을 할 때까지 제어권을 넘겨주지 않는다.

어떤 작업이 완료될 때까지 기다려야 할 때 유리하다



```
mhn@mhn-Z20NH-AS51B5U: ~/linux/21
mhn@mhn-Z20NH-AS51B5U:~$ cd linux/21
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ gcc test4.c
test4.c: In function 'main':
test4.c:10:2: warning: implicit declaration of function 'mkfifo' [-Wimplicit-fun
ction-declaration]
   mkfifo("myfifo");
   ^
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ mkfifo myfifo
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ./a.out test4.c
aaa
keyboard input : [aaa]
bbb
Pipe input : [aaa]
keyboard input : [bbb]
Pipe input : [vv
]
dsa
keyboard input : [dsa]
Pipe input : [ddd]
dsa
keyboard input : [dsa]
Pipe input : [dsa]
dd
keyboard input : [dd]
```

Recent
Home
Desktop
Documents

a.out a.txt

```
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ cat > myfifo
aaa
vv
ddd
ss
aads
a
a
dsa
as
^?^C
mhn@mhn-Z20NH-AS51B5U:~/linux/21$
```

read()함수는 읽어온 바이트 수를 반환하는데 이 예제에서는 ret에 저장되어 있다.

만약 버퍼에 있는 엔터값을 0으로 초기화 하지 않는다면

```
mhn@mhn-Z20NH-AS51B5U: ~/linux/21
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 int main(void){
7
8     int fd,ret;
9     char buf[1024];
10    mkfifo("myfifo");
11    fd = open("myfifo",O_RDWR);
12    for(;;){
13
14        ret = read(0,buf,sizeof(buf));
15        // buf[ret-1]=0;
16        printf("keyboard input : [%s]\n",buf);
17        read(fd,buf,sizeof(buf));
18        // buf[ret-1]=0;
19        printf("Pipe input : [%s]\n",buf);
20
21    }
22
23    return 0;
24
25 }
```

%s에서 같이 출력이 될 것이다

```
(fd);
    return 0;
}

mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ls
a.out quiz1_2.c t1.txt test1.c test3.c test
a.txt screenshot test test2.c test4.c ttt.
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ gcc test4.c
test4.c: In function 'main':
test4.c:10:2: warning: implicit declaration of fu
mkfifo("myfifo");
^
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ vi test4.c
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ mkfifo myfifo
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ./a.out
s
keyboard input : [s]
a
s
Pipe input : [d]
keyboard input : [a]
Pipe input : [s]
keyboard input : [s]
^C
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ vi test4.c
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ gcc test4.c
test4.c: In function 'main':
test4.c:10:2: warning: implicit declaration of function 'mkfifo' [-Wimplicit-function
mkfifo("myfifo");
^
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ mkfifo myfifo
mhn@mhn-Z20NH-AS51B5U:~/linux/21$ ./a.out
aaa
keyboard input : [aaa]
^]
```



```

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

int main(void){

    int fd,ret;
    char buf[1024];
    fd = open("myfifo",O_RDWR);
    fcntl(0,F_SETFL,O_NONBLOCK); // SETFL은 파일권한 세팅이다
    fcntl(fd,F_SETFL,O_NONBLOCK); //fd를 NONBLOCK으로 세팅한다

    for(;;){

        if((ret = read(0,buf,sizeof(buf)))>0){

            buf[ret-1]=0;
            printf("Keyboard input : [%s]\n",buf);
        }
        if((ret = read(fd,buf,sizeof(buf)))>0){
            buf[ret-1]=0;
            printf("pipe input : [%s]\n",buf);
        }
    }

    close(fd);

    return 0;

}

```

read는 block함수로 먼저 입력을 할 때까지 제어권을 넘겨주지 않는다.

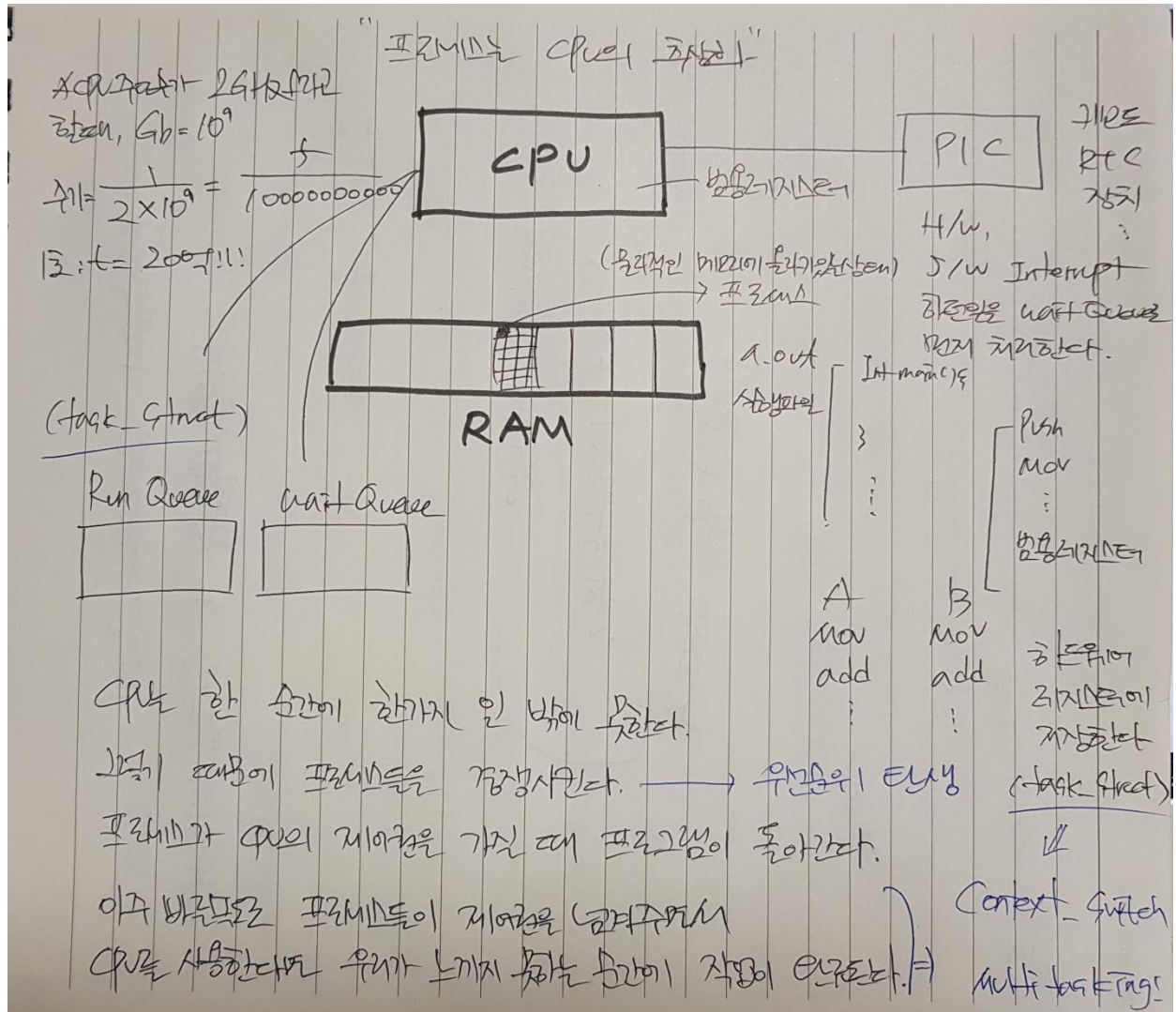
하지만 이 예제에서는 NONBLOCK으로 세팅이 되어있어서 계속 읽을 것을 찾다가 먼저 들어온 입력이 있으면 순서대로 처리한다

NONBLOCK은 많은 사람들이 빠른 통신을 할 때 유리하다

```
crw-rw---- 1 root tty 7, 132 3월 21 16:41 vcsa4
crw-rw---- 1 root tty 7, 133 3월 21 16:41 vcsa5
crw-rw---- 1 root tty 7, 134 3월 21 16:41 vcsa6
drwxr-xr-x 2 root root 60 3월 21 16:41 vfio
crw----- 1 root root 10, 63 3월 21 16:41 vga_arbiter
crw----- 1 root root 10, 137 3월 21 16:41 vhost-net
crw----- 1 root root 10, 238 3월 21 16:41 vhost-vsock
crw-rw-rw- 1 root root 1, 5 3월 21 16:41 zero
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ mkfifo myfifo
mkfifo: cannot create fifo 'myfifo': File exists
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ls -al
total 48
drwxrwxr-x 2 mhn mhn 4096 3월 21 16:51 .
drwxrwxr-x 5 mhn mhn 4096 3월 21 10:04 ..
-rwxrwxr-x 1 mhn mhn 8760 3월 21 16:43 a.out
-rw-r--r-- 1 mhn mhn 22 3월 21 13:22 a.txt
-rw-r--r-- 1 mhn mhn 512 3월 21 14:24 mbr.txt
prw-rw-r-- 1 mhn mhn 0 3월 21 16:46 myfifo
-rw-rw-r-- 1 mhn mhn 3360 3월 21 13:05 quiz1_2.c
-rw-rw-r-- 1 mhn mhn 225 3월 21 13:52 test1.c
-rw-rw-r-- 1 mhn mhn 297 3월 21 14:06 test2.c
-rw-rw-r-- 1 mhn mhn 251 3월 21 14:24 test3.c
-rw-rw-r-- 1 mhn mhn 375 3월 21 16:40 test4.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ls
a.out a.txt mbr.txt myfifo quiz1_2.c test1.c test2.c test3.c test4.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test4.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ gcc test5.c
test5.c: In function 'main':
test5.c:14:20: error: 'buf' undeclared (first use in this function)
    if((ret = read(0,buf,sizeof(buf)))>0){
                   ^
test5.c:14:20: note: each undeclared identifier is reported only once for each function it appears in
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ vi test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ gcc test5.c
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ mkfifo myfifo
mhn@mhn-Z20NH-AS51BSU:~/linux/21$ ./a.out test5.c
pipe input : [a]
a
Keyboard input : [a]
```

프로세스들이 경쟁을 하며 CPU의 제어권을 가질 때 프로그램이 돌아간다.

이로써 프로세스는 CPU의 추상화라고 할 수 있다.



CPU는 한 순간에 한가지 일 밖에 못한다. 그렇기 때문에 프로세스들을 경쟁시키며, 이 이유로 우선순위가 생긴다.

여기에서 나온 중요한 개념이 **Context Switching**인데, 이것은 인터럽트 요청에 의해 다음 명령이 실행되어야 할 때 우선순위를 판단하여 하드웨어 레지스터에 실행할 명령을 저장하는 것이다.

아주 빠른 속도로 여러 프로세스들이 제어권을 넘겨주면서 CPU를 사용한다면 우리가 느끼지 못하는 순간에 모든 작업이 완료될 것이다. 이를 **Multi tasking**이라고 하며 이 덕분에 우리는 한번에 여러가지 일을 동시에 처리 할 수 있게 된다.