

Xilinx Zynq FPGA, TI DSP, MCU
기반의 프로그래밍 및 회로 설계
전문가 과정

<리눅스 시스템 프로그래밍>
2018.03.23 – 22 일차

강사 – 이상훈
gcccompil3r@gmail.com

학생 – 안상재
sangjae2015@naver.com

소스코드 분석

1) ls-r 명령어 구현 : 현재 디렉토리부터 최하위 디렉토리 까지 재귀함수를 통해 순회하면서 존재하는 모든 파일을 출력함.

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

void recursive_dir(char *dname); // 함수포인터라면 main안에서도 가능

int main(int argc, char *argv[])
{
    recursive_dir("."); // "." 을 받았기 때문에 main으로 인자를 받을 필요없음
    return 0;
}

void recursive_dir(char *dname)
{
    struct dirent *p;    // 디렉토리 포인터에 잇는 리스트들 (각종 파일들)
    struct stat buf;     // 파일 상태
    DIR *dp;
    chdir(dname);        // 디렉토리 포인터를 현재위치로 바꾸어주는 시스템 콜.
    dp = opendir(".");
    printf("wt%s : \n", dname);
    while(p = readdir(dp))    // 리스트가 모두 순회를 할 때까지
        printf("%s\n", p->d_name);

    rewinddir(dp);        // 되감기 (포인터를 맨 앞으로 이동시킴)
    while(p = readdir(dp))
    {
        stat(p->d_name, &buf);    // st.mode를 얻어옴.

        if(S_ISDIR(buf.st_mode))
            if(strcmp(p->d_name, ".") && strcmp(p->d_name, "..")) // ".", ".." 재깸
                recursive_dir(p->d_name);
    }
    chdir("..");    // 상위로 이동 (처음에 들어갈때보다 한단계 더 상위로 끝남)
    closedir(dp);
}
```

*** 프로세스**

- 프로세스는 main함수를 의미한다. (프로세스 1개는 main함수 1개를 의미)
- 개발 과정 중에 프로세스를 나누어서 진행하는 이유 : 각각의 프로세스마다 특정 역할을 부여해 체계적인 개발을 진행하기 위해.
- 여러 개의 프로세스들은 Virtual Memory (가상 메모리)를 공유하지 않는다. (각각의 프로세스는 다른 프로세스의 페이지징에 대한 권한이 없음), 다른 프로세스에게 데이터를 넘겨주려면 파이프(메시지 큐, 공유 메모리)가 있어야 함.
- C.O.W(Copy On Write) : fork()에 의해 자식 프로세스가 생성이 되면, 부모 프로세스의 데이터를 쓰기 작업이 필요할 때만 복사를 한다.

*** fork 시스템 콜**

- fork 시스템 콜은 자식 프로세스를 만들고, 자식 프로세스는 fork() 가 있는 줄부터 실행이 됨. 부모는 fork() 실행함으로써 자식 프로세스의 pid 값을 반환 받고, 자식 프로세스는 fork()를 실행함으로써 0을 반환 받는다. (자식 프로세스는 생성 즉시 곧바로 프로세스를 생성할 수 없음.)
- fork 시스템 콜에 의해 프로세스가 2개가 되고(부모,자식) 각각의 프로세스는 독립적으로 수행이 된다. (부모, 자식 프로세스의 실행 순서는 프로세스 스케줄링에 따라 다름.)
- 자식 프로세스가 종료되면, 반드시 부모 프로세스가 처리해 주어야 함.(자식의 시신은 부모가 처리해 주어야 함.) / 자식 프로세스가 종료되는 순간 부모 프로세스가 sleep() 시스템 콜에 의해 처리해 주지 못한다면 자식 프로세스는 좀비 프로세스가 된다.

2) 자식 프로세스는 printf("before\n"); 을 수행하지 않음.

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

int main(void)
{
    int i;
    printf("before\n");
    fork();    // 자식 프로세스를 만듦.
    printf("after\n");
    return 0;
}
```

3) fork()를 하면 부모 프로세스는 자식 프로세스의 pid값을 반환 받기 때문에 부모 프로세스의 pid는 0보다 크고, 자식 프로세스는 생성된 즉시 새로운 프로세스를 생성할 수 없기 때문에 fork()로부터 0을 반환 받는다.

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

int main(void)
{
    pid_t pid;    // int도 상관없음.
    pid = fork(); // fork는 자식의 pid값을 반환, 부모는 자식이 있기니까 부모의 pid>0,
                // 자식의 pid=0

    if(pid > 0)
        printf("parent\n");
    else if(pid == 0)
        printf("child\n");
    else
    {
        perror("fork() "); // perror는 어떤 에러가 나왔는지 출력
        exit(-1);
    }
    return 0;
}
```

4) getpid() 함수는 함수를 호출한 프로세스의 ID를 리턴한다. (getppid()는 부모 프로세스의 ID를 리턴함.)

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

int main(void)
{
    pid_t pid;
    printf("%d\n",pid);
    pid = fork(); // 자식프로세스는 이 줄부터 실행됨. (fork()로부터 0을 반환받음)
    if(pid > 0)
        printf("parent : pid = %d, cpid = %d\n", getpid(), pid);
    else if(pid == 0)
        printf("child : pid = %d, cpid = %d\n",getpid(),pid);
    else
    {
        perror("fork() ");
        exit(-1);
    }
}
```

```
}  
  
return 0;  
}
```

5) 프로세스가 2개 이상이 되면 운영체제가 멀티 태스킹을 수행하여, 프로세스 스케줄링 기법에 의해 여러 프로세스를 번갈아 가며 수행하게 된다.

```
#include <stdio.h>  
#include <unistd.h>  
#include <errno.h>  
#include <stdlib.h>  
int main(void)  
{  
    pid_t pid;  
    int i;  
    pid = fork();  
  
    if(pid > 0)    // 부모 프로세스 수행  
    {  
        while(1)  
        {  
            for(i=0;i<26;i++)  
            {  
                printf("%c ",i+'A');    // 'A' ~ 'Z' 의 문자를 출력함.  
                fflush(stdout);    // 버퍼를 비움.  
            }  
        }  
    }  
    else if(pid == 0)    // 자식 프로세스 수행  
    {  
        while(1)  
        {  
            for(i=0;i<26;i++)  
            {  
                printf("%d ",i+'A');    // 'A' ~ 'Z' 의 십진수를 출력함.  
                fflush(stdout);  
            }  
        }  
    }  
    else  
    {  
        perror("fork()");  
        exit(-1);  
    }  
    printf("\n");  
    return 0;  
}
```

5-1) 결과 화면 분석 : 부모 프로세스는 문자를 출력하고 자식 프로세스는 숫자를 출력하는데, 결과화면에는 문자와 숫자가 번갈아 가며 출력이 되는 것을 확인할 수 있다. 즉, 운영체제는 멀티 태스킹을 한다는 것이 증명된다.

```
65 J 66 K 67 L 68 M 69 N 70 O 71 P 72 Q 73 R 74 S 75 T 76 U 77 V 78 W 79 X 80 Y 81 Z 82 A 83 B 84 C 85
80 Y 81 Z 82 A 83 B 84 C 85 D 86 E 87 F 88 G 89 H 90 I 65 J 66 K 67 L 68 M 69 N 70 O 71 P 72 Q 73 R 7
M 69 N 70 O 71 P 72 Q 73 R 74 S 75 T 76 U 77 V 78 W 79 X 80 Y 81 Z 82 A 83 B 84 C 85 D 86 E 87 F 88 G
B 84 C 85 D 86 E 87 F 88 G 89 H 90 I 65 J 66 K 67 L 68 M 69 N 70 71 72 73 74 75 76 77 78 79 80 81 82
Z 84 A 85 B 86 C 87 D 88 E 89 F 90 G 65 H 66 I 67 J 68 K 69 L 70 M 71 N 72 O 73 P 74 Q 75 R 76 S 77 T
2 O 73 P 74 Q 75 R 76 S 77 T 78 U 79 V 80 W 81 X 82 Y 83 Z 84 A 85 B 86 C 87 D 88 E 89 F 90 G 65 H 66
87 D 88 E 89 F 90 G 65 H 66 I 67 J 68 K 69 L 70 M 71 N 72 O 73 P 74 Q 75 R 76 S 77 T 78 U 79 V 80 W 81
76 S 77 T 78 U 79 V 80 W 81 X 82 Y 83 Z 84 A 85 B 86 C 87 D 88 E 89 F 90 G 65 H 66 I 67 J 68 K 69 L 7
G 65 H 66 I 67 J 68 K 69 L 70 M 71 N 72 O 73 P 74 Q 75 R 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
89 90 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 65 66 67 68 69 70
```

6) 다음과 같은 두 프로세스 A,B를 실행시키면 B 프로세스에서는 세그멘테이션 오류가 발생한다. 여러 개의 프로세스간에는 서로의 프로세스상의 가상메모리를 공유하지 못한다. (부모 프로세스와 자식 프로세스의 관계라 할지라도)

```
/* A 프로세스 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
int main(void)
{
    int a = 10;
    printf("&a = %p\n", &a);
    sleep(1000);
    return 0;
}

/* B 프로세스 */

#include <stdio.h>
int main(void)
{
    int *p = 0x7ffdddea2eb4; // A 프로세스상의 변수 a의 주소
    printf("&a : %#p\n", *p);
    return 0;
}
```

7) 자식 프로세스가 생성되면, 부모 프로세스를 그대로 복사해 오지만, 데이터는 필요할 때(COW : write작업을 수행할 때 복사) 복사를 한다.

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

int global = 100;
int main(void)
{
    int local = 10;
    pid_t pid;
    int i;
    pid = fork();
    if(pid > 0)    // 부모 프로세스 실행
    {
        global++;
        printf("global:%d, local:%d\n", global, local);
    }
    else if(pid == 0)    // 자식 프로세스 실행
    {
        global++;
        local++;
        printf("global:%d, local:%d\n", global, local);
    }
    else
    {
        perror("fork() ");
        exit(-1);
    }
    printf("\n");
    return 0;
}
```

8) mkfifo (mkfifo 파일명) 명령어를 통해 파이프 파일을 만들고, 부모 프로세스는 키보드의 입력을 받고 자식 프로세스는 파이프로부터 입력을 받는다. 운영체제가 멀티 태스킹을 하기 때문에 부모, 자식 프로세스에서 실시간으로 입력을 받고 결국 nonblocking이 된다.

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

int main(void)
{
    int fd, ret;
    char buf[1024];
```



```

pid_t pid;
fd = open("myfifo", O_RDWR);
if((pid = fork()) > 0)
{
    for(;;)
    {
        ret = read(0,buf,sizeof(buf)); // parent프로세스는 키보드의 입력만
기다림(blocking)
        buf[ret] = 0;
        printf("keyboard : %s\n", buf);
    }
}
else if(pid == 0)
{
    for(;;)
    {
        ret = read(fd, buf, sizeof(buf)); // 자식 프로세스는 파이프에서 입력을
기다림
        buf[ret] = 0;
        printf("myfifo : %s\n", buf);
    }
}
else
{
    perror("fork() ");
    exit(-1);
}
close(fd);
return 0;
}

```

9) 자식 프로세스가 종료되었는데 부모가 처리해주지 못하면 자식 프로세스는 좀비 프로세스가 된다.

```

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

int main(void)
{
    pid_t pid;
    if((pid = fork()) > 0)
        sleep(10); // 부모 프로세스는 실행이 되고 10초간 중지됨.
    else if(pid == 0)
        ; // 자식 프로세스는 수행이 되는 순간 종료가 됨.
    else

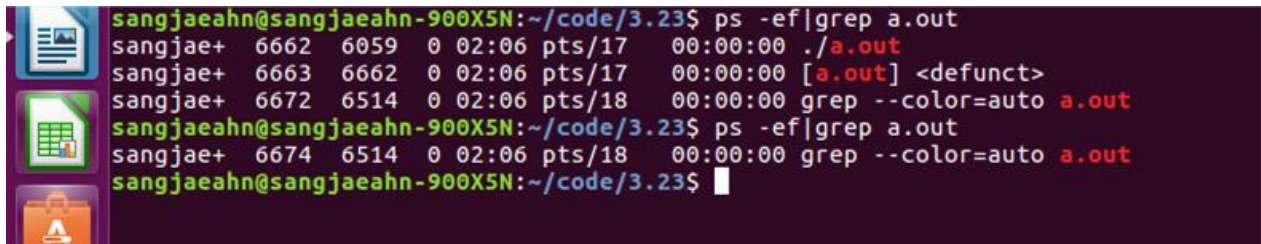
```

```

{
    perror("fork()");
    exit(-1);
}
return 0;
}

```

9-1) 결과화면 분석 : 좀비 프로세스는 <defunct>로 표시된다. 부모 프로세스가 10초동안 잠들고, 깨어나면 좀비 프로세스가 처리되고 <defunct>는 없어진다.



```

sangjaahn@sangjaahn-900X5N:~/code/3.23$ ps -ef | grep a.out
sangjae+ 6662 6059 0 02:06 pts/17 00:00:00 ./a.out
sangjae+ 6663 6662 0 02:06 pts/17 00:00:00 [a.out] <defunct>
sangjae+ 6672 6514 0 02:06 pts/18 00:00:00 grep --color=auto a.out
sangjaahn@sangjaahn-900X5N:~/code/3.23$ ps -ef | grep a.out
sangjae+ 6674 6514 0 02:06 pts/18 00:00:00 grep --color=auto a.out
sangjaahn@sangjaahn-900X5N:~/code/3.23$

```

10) wait() 시스템 콜은 자식 프로세스가 종료될 때까지 기다리고 어떻게 종료 되었는지 확인하는 시스템 콜이다.

```

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    int status;

    if((pid = fork()) > 0)
    {
        wait(&status); // 자식 프로세스가 종료되고 남긴 값을 status 변수로 받음.
        printf("status : 0x%x\n", WEXITSTATUS(status)); // 자식 프로세스의 전달값을
        반환함.
    }
    else if(pid == 0)
    {
        exit(7); // 자식 프로세스 종료하고 인자로 7을 넘김.
    }
    else
    {
        perror("fork() ");
        exit(-1);
    }
}

```

```

    }

    return 0;
}

```

11) abort() 시스템콜은 프로세스를 강제로 종료시키는 signal 이고 wait()에 6을 전달한다.

```

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    int status;

    if((pid = fork()) > 0)
    {
        wait(&status);
        printf("status : %d\n", status & 0x7F);
    }
    else if(pid == 0)
    {
        abort();    // signal (프로세스를 죽임) 강제로 프로세스 종료
    }
    else
    {
        perror("fork() ");
        exit(-1);
    }

    return 0;
}

```

11-1) kill -l 명령어 : signal 목록을 보여줌. Abort() 는 6번임.

```

sangjaeahn@sangjaeahn-900X5N:~/code/3.23$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX

sangjaeahn@sangjaeahn-900X5N:~/code/3.23$ vi fork11.c

```