TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-03-26 (23 회차)

강사: Innova Lee(이상훈)

gcccompil3r@gmail.com

학생: 정유경

ucong@naver.com

1. Child Process 의 status - 1.c

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

void term_status(int status)
{
        if(WIFEXITED(status)) // 정상종료
                printf("(exit)status: 0x%x₩n", WEXITSTATUS(status));
        else if(WTERMSIG(status)) // 시그널에 의한 비정상 종료
                printf("(signal)status: 0x%x, %s₩n",status & 0x7f, WCOREDUMP(status) ? "core
dumped": "");

}

int main(void)
{
        pid_t pid;
        int status;
        if(pid=fork()>0)
        {
                wait(&status);
                term_status(status);
        }
        else if(pid==0)
                abort(); // SIGNAL 6 가 status 로 들어간다.
        else
        {
                perror("fork() ");
                exit(-1);
        }



        return 0;
}
```

status	8bit	8bit
정상종료	프로세스 반환값 (exit 함수 입력인자)	0
비정상종료	0	종료시킨 시그널 번호

맨 앞은 코어덤프 관련 비트

*. 자식프로세스의 상태 값(종료 상태값을 확인할 수 있는 몇 가지 매크로 함수)

WIFEXITED(status)

- 자식 프로세스가 정상 종료되었다면 Non-Zero 값 리턴. (정상종료시 참)

WEXITSTATUS(status)

- 자식 프로세스에서 exit()를 호출할 때 인자를 설정하거나, 또는 어떤 return 값을 설정하고, 해당 자식 프로세스가 종료될 경우 return 값의 최하위 8 비트를 가져옴. (정상종료시 하위 8 비트 값)
- 정상종료 (즉, WIFEXITED(status)값이 Non-Zero)일때만 사용가능.

WTERMSIG(status)

- 자식 프로세스를 종료하도록 한 시그널의 번호를 반환. (시그널에 의해 종료되었을 때 시그널 번호)
- WIFSIGNALED(status)값이 Non-Zero 일 경우에만 사용가능.

WIFSIGNALED(status)

- 자식 프로세스가 어떤 시그널로 인해 종료되었다면 TRUE 반환. (시그널에 의해 종료되면 참)

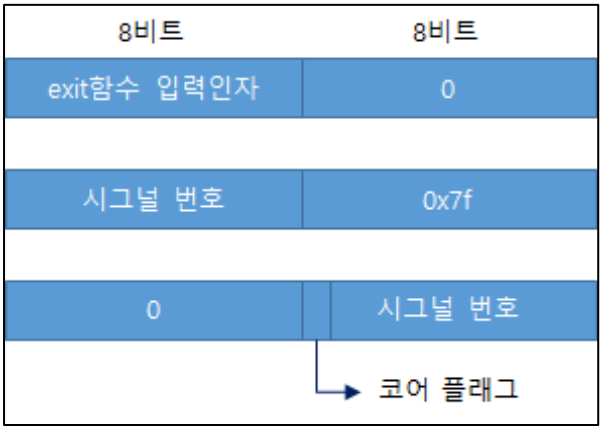
WCOREDUMP(status)

- 코어 파일 발생 여부

*. 0x7f 인 이유 맨 앞은 코어덤프

프로그램 비정상 종료 시 이에 대한 메모리 덤프 (코어 덤프)파일을 만들 것(1)인지 말 것(0)인지 결정

코어 덤프(core dump), 메모리 덤프(memory dump), 또는 시스템 덤프(system dump)[1]는 컴퓨터 프로그램이 특정 시점에 작업 중이던 메모리 상태를 기록한 것으로, 보통 프로그램이 비정상적으로 종료했을 때 만들어진다. 코어 덤프는 프로그램 오류 진단과 디버깅에 쓰인다.



종료 상태값은 16 비트 값으로 구성한다. 하위 8 비트 값이 0 이면 상위 8 비트 값은 종료할 때 exit 함수에 전달한 인자값(main 함수 return 값)이다. 하위 8 비트 값이 0x7f 일 때는 자식 프로세스가 SIGTSTP 이나 SIGSTOP 시그널에 의해 임시 중단 상태이며 상위 8 비트에 시그널 번호가 온다. 비정상 종료일 때는 상위 8 비트 값은 0 이지만 하위 8 비트 중에 7 비트는 시그널 번호가 오고 맨 첫번째 비트는 코어 플래그가 온다. 코어 플래그 값은 코어 파일 발생 여부 정보를 의미한다.

2. signal() ,sleep(), usleep() System Call 활용 - signal.c

불이 언제 날지 알 수 없으니 행동지침만 등록해놓자 signal(SIGCHLD, my_sig);
즉, SIHCHLD 들어올 때까지 my_sig 는 동작하지 않는다. 이를 "비동기처리"라고 한다.
시그널을 통해서 상황에 따라 제어가 가능하다.

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


void term_status(int status)
{ // Return 2; exit(2)처럼 각각의 반환값에 의미를 부여할 수 있다
        if(WIFEXITED(status)) // return 0; 이므로 0 이 출력된다.
                printf("(exit)status: 0x%x₩n", WEXITSTATUS(status));
        else if(WTERMSIG(status))
                printf("(signal)status: 0x%x, %s₩n", status & 0x7f, WCOREDUMP(status) ? "core dumped" :
"");
}

void my_sig(int signo)
{
        int status;
        wait(&status);
        term_status(status);
}

int main(void)
{
        pid_t pid;
        int i;
        signal(SIGCHLD, my_sig); // 동작매뉴얼: "SIGCHLD 시그널이 들어오면 my_sig 를 동작시켜라"
        if((pid=fork())>0)
                for(i=0;i<10000;i++)
                {
                        usleep(50000); // 0.05 초 간격으로 100 번실행시(5 초) SIGCHLD 가 발생
                        printf("%d₩n",i+1);
```

```
        }
    else if(pid==0)
        sleep(5); // 5 초간 sleep 후 정상종료, 부모에게 SIGCHLD 전송한다.
    else
    {
        perror("fork() ");
        exit(-1);
    }
    return 0;
}
```

*.signal() 시스템 콜

#include <signal.h> void (*signal(int signum, void (*handler) (int))) (int);

- signum : 제어하고자 하는 시그널 번호

- handler : 시그널이 발생했을 때, 실행할 함수의 주소 즉, 함수포인터

(함수포인터가 인자로 전달되므로, 시그널은 비동기적인 사건을 전달하는데, 유용하게 사용할 수 있다.)

3. Wait 의 논블록킹 함수 wait_pid (프로세스의 종료를 기다린다)....???

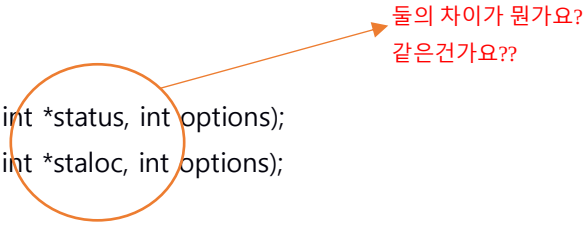
wait

#include <sys/types.h>

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);

pid_t waitpid(pid_t pid, int *staloc, int options);



*. waitpid 와 wait 은 일반적으로 생성된 자식 프로세스가 종료하거나 멈추도록 기다리거나, 그러한 상황인지를 판단하는데 사용할 수 있는 함수들이다.

*.리턴값은 에러라면 -1, WNOHANG 이 사용되고 어떤 자식도 이용되지 않았다면 0, 그 이외의 정상적인 경우는 보고된 상황을 가진 자식 프로세스의 PID 이다.

*. wait 함수는 waitpid 의 pid_t 인자가 -1 일때와 동일하게 동작한다. wait(&status)는 wait(-1, &status, 0)와 완전히 동일하다.

*. waitpid 의 첫번째 인자로 -1 을 전달하면 자식 프로세스중 어떤것을 의미한다(any)

. wait 시스템콜: for 문 전에 시그널 핸들러를 등록해놓고 그 핸들러에서 호출을 했는데 for 문 뒤에 wait(&stat)을 놓으면, for 문 내에서 fork 한 자식 프로세스가 수행되고 있는 상황에서 wait 시스템콜은 죽어있는 자식 프로세스가 없으면 **블럭되서 자식이 죽을때 까지 기다린다**.

그렇게 되면 자식 프로세스가 종료될 때까지 다른 요청을 받지 못하고 무시하게 된다.

블록킹 방식의 wait 시스템콜을 좀더 효과적으로 처리하려면 **비동기식, 논 블록킹 방식의 waitpid** 가 필요하다. (콜센터처럼) SIGCHLD 가 여러 개 들어올 때 처리 순번을 부여하여 예약된 순서대로 처리하는 방식을 사용한다. 따라서 서버가 뻗지않는다.

***. non-blocking 시스템콜인 waitpid()를 사용하는 이유**

wait 함수처럼 블록 상태가 되는게 아니라 종료된 프로세스가 없으면 그냥 넘어가서 다음 코드를 수행한다.
while 루프로 감싸서 계속 체크한다 물론 cpu 점유율이 높아질수있으므로 sleep 함수로 일정시간 동안 멈췄다가 체크하게 할수있다

while(waitpid(-1, &status,WNOHANG)>0)

***. waitpid**

[1] 첫번째 인자, 프로세스 id 이 값에 따라 대기할 프로세스가 달라진다.
양수는 해당 pid 값을 갖는 자식 프로세스를 기다린다.
0 일 때는 자신과 같은 그룹에 있는 프로세스의 자식 프로세스????의 종료를 기다린다.
(프로세스 그룹 ID 가 호출 프로세스의 ID 와 같은 자식프로세스를 기다린다.)
-1 일 때는 모든 하위 프로세스 -1 즉, 임의의 자식프로세스를 기다린다.
-1 보다 작은 값일 때는 프로세스 그룹 ID 가 pid 의 절대값과 같은??? 자식 프로세스의 종료를 기다린다.
[2] 두번째 인자, status 는 프로세스의 상태 저장하고 이를 가져오기 위해서 사용한다.
[3] 세번째 인자, options 의 값은 0 이거나 다음값들과의 OR 이다.
WNOHANG 일 때는 어떠한 자식도 종료되지 않았더라도 즉시 리턴
(이미 종료한 프로세스의 종료 상태값을 확인하기 위한 목적)
WUNTRACED 일 때는 종료된 프로세스 뿐만 아니라 멈춘 프로세스로부터도 상태정보를 얻어온다.

< execve() 계열의 실행 System Call >

3. printf 가 출력되지 않는다 - 둔갑술이라서- execve.c

```
#include <unistd.h>

int main()
{
        execlp("ps","ps","-e","-f",0); // ps -ef : 현재 구동중인 프로세스리스트 출력
        // 프로그램이름, argv[0] ,argv[1],argv[2]
        printf("after₩n");
        return 0;
}
```

*. fork()는 분신술 execve() 는 둔갑술 → 메모리 레이아웃을 ps 로 바꾸게 된다.
가상 메모리 레이아웃만 바뀌기 때문에 printf 는 없다!

4. fork() 와 execve() 를 함께 사용하기 - forkexec.c

fork 하고 둔갑시켜서 자식은 ps -ef 로 부모는 after 를 출력하도록 하기

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

int main(void)
{
        int status;
        pid_t pid;
        if((pid=fork())>0)
        {
                wait(&status);
                printf("prompt >");
        }

        else if(pid==0)
                execlp("ps","ps","-e","-f",0);  // 파일을 인자로 받는다
                        return 0;
}
```

```
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
```
file에 지정한 파일을 실행하며 arg0~argn만 인자로 전달한다. 파일은 이 함수를 호출한 프로세스의 검색 경로(환경 변수 PATH에 정의된 경로)에서 찾는다. arg0~argn은 포인터로 지정한다. execl 함수의 마지막 인자는 NULL 포인터로 지정해야 한다.

5.내가 만든 실행파일도 실행 시킬 수 있구나 - newpgm.c & execve4.c

```
   ^
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ vi execve4.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ vi newpgm.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ gcc -o newpgm newpgm.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ gcc execve4.c
execve4.c: In function ‘main’:
execve4.c:10:3: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declarat
ion]
   wait(&status);
   ^
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ ./a.out
argv[0] = [newpgm]
argv[1] = [one]
argv[2] = [two]
prompt > yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ vi execve4.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$ vi newpgm.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0326$
```

newpgm.c

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

```
int main(int argc, char **argv)
{
        int i;
        for(i=0;argv[i];i++)
                printf("argv[%d] = [%s]₩n",i,argv[i]);
        return 0;
}
```

execve4.c

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

int main()
{
        int status;
        pid_t pid;
        if((pid=fork())>0)
        {
                wait(&status);
                printf("prompt > ");
        }
        else if(pid==0)
                execl("./newpgm", "newpgm","one","two",(char*)0);
        return 0;
}
```

6.execve5.c – newpgm.c 를 고쳐서 env[] 동작하도록 해보자

수정된 newpgm.c

```
#include <stdio.h>

int main(int argc, char **argv, char**env)
{
        int i;
        for(i=0;argv[i];i++)
                printf("argv[%d] = [%s]₩n",i,argv[i]);
        for(i=0;env[i];i++)
                printf("env[%d] = [%s]₩n",i,env[i]);

        return 0;
```

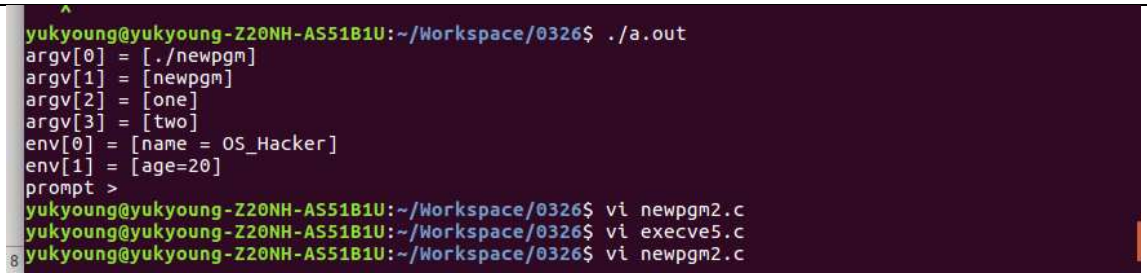
```
}
```

execve5.c

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

int main()
{
        int status;
        pid_t pid;
        char *argv[]={"./newpgm","newpgm","one","two",0};
        char *env[]={"name = OS_Hacker", "age=20",0};

        if((pid=fork())>0)
        {
                wait(&status);
                printf("prompt > ₩n");
        }
        else if(pid==0)
        {
                execve("./newpgm",argv,env);
        }
        return 0;
}
```



7. system 함수를 써보자

```
#include <stdio.h>

int main()
{

//      system("date");
        system("ls");
        printf("after₩n");
        return 0;
```

```
}
```

8. 7 과 동일한 작업 하도록, my_system 함수를 만들자 execve7.c

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

int my_system(char*cmd)
{
        pid_t pid;
        int status;
        char *argv[]={"sh","-c",cmd,0};
        char *envp[]={0};
        if((pid=fork())>0)
                wait(&status);
        else if(pid==0)
                execve("/bin/sh",argv,envp);
}


int main()
{
        my_system("date");
        printf("after₩n");
        return 0;
}
```

9. 일반적인 프로세스를 종료시켜 보자 - temp.c

Ps-ef | grep a.out 으로 실행중인 프로세스를 찾아서 kill -9 PID 해준다.

```
#include <stdio.h>
int main()
{

          while(1){
                sleep(1);
                system("date");
          }
```

```
//        system("ls");
          printf("after₩n");
          return 0;
}
```

그러나, 서버의 경우 터미널을 끄던말던 살아있어야 한다.
따라서 데몬 프로세스로 만들어 터미널을 끄던 말던 살아있도록 만든다. -> 10 번
예를 들면, 차량 영상처리, 서비스프로그램. FTP, 토렌트, 웹서버(구글, 네이버 등의 포털)이 데몬 프로세스로 되어있다.

*. 프로세스는 터미널과 생명을 같이 한다. 따라서 터미널이 꺼지면 프로세스도 종료된다.
일반 프로세스는 TTY, PTS 라는 가상터미널의 세션 id??를 가지고 있다.
(ps -ef | grep [실행파일명] 으로 살펴보면 데몬은 '?'로 표시된다)
Setsid 할 경우 터미널에 소속되지 않게 되면서 데몬이 된다.
즉, 터미널을 닫아도 종료되지 않는 프로세스가 되는 것이다.

(★)10. 데몬프로세스를 만들어 보자 - daemon.c

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

int daemon_init(void)
{
          int i;
          if(fork()>0)
                    exit(0);
          setsid(); //새로운 세션을 만든다. 자식프로세스를 현재 세션과 상관없이 동작시키기 위함
          chdir("/"); //루트로 옮긴다
          umask(0);
// i<64 기본적으로 리눅스는 64 개 파일을 열어놓는데 (0~63) 모든 연관을 다 끊는다.
          for(i=0;i<64;i++)
                    close(i);
          signal(SIGCHLD, SIG_IGN); // SIGCHLD 는 무시한다
          return 0;
}

int main(void)
{
          daemon_init();
```

```
        for(;;);
        return 0;
}
```

*. 데몬과 일반 프로세스와의 차이점
Daemon 들은 첫째 TTY(터미널 장치)를 가지고 있지 않다. 둘째 PPID(parent id)가 1(init)로 세팅되어 있으며 SID(session id)역시 자신의 아이디와 같다.

*. #include <unistd.h> pid_t setsid(void);
*. 반환값: 정상적으로 완료하면, setsid() 함수는, 새로운 프로세스 그룹의 프로세스 그룹 ID 의 값을 돌려준다. 이것은 호출한 프로세스의 프로세스 ID 와 같다.
*. setsid 함수는 호출하는 프로세스가 그룹의 리더가 아닐때 새로운 세션을 생성하여 다음과 같은 세 가지 일을 한다.
호출한 프로세스는 새로운 세션의 리더가 된다.
호출한 프로세스는 새로운 그룹의 리더가 된다.
프로세스는 컨트롤 터미널을 잃어 버리게 된다(언제나 얻을 수만은 없다)
※ session 이란 ? 하나 또는 이상의 프로세스 그룹들의 집합이다.

*. 새로운 디렉토리를 찾아서
chdir 을 이용하여 현재 워킹 디렉토리를 루트 디렉토리(/)로 변경한다. 부모로 물려 받은 워킹 디렉토리는 파일 시스템에 마운트 되어 있는 것일 수도 있고, 시스템이 정지 할때 까지 살아 있는 Daemon 의 특성 때문에 파일 시스템이 언마운트 되지 않을 수도 있다.

11. 모든 시그널을 무시하더라도, init 의 Kill 시그널은 막을 수 없다. - last.c

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

int main()
{
        signal(SIGINT, SIG_IGN); //    컨트롤 c
        signal(SIGQUIT, SIG_IGN); //     컨트롤 ₩
        signal(SIGKILL, SIG_IGN); // 킬 까지 막았는데 kill -9 는 못막는다
        pause();
        return 0;
}
```

Ps -ef | grep a.out 하여 데몬의 PID 를 확인 후, Kill -9 PID 와 같은 방식으로 종료시킨다.