TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
23 일차 (2018. 03. 26)

목차

학습 내용 복습

- waitpid()
  > NON Blocking
- core dump
- signal()
- exec 계열의 함수
- 데몬 프로세스
- 학습 예제

waitpid()

#include <sys/wait.h>를 포함해줘야 한다. **waitpid(-1, &status, 0);**와 같이 사용된다. 시스템 콜로 호출 프로세스의 실행을 일시 중지한다. 그러나 wait()와 달리 인수로 주어진 pid 번호의 자식 프로세스가 종료되거나, 시그널 함수를 호출하는 신호가 전달될 때까지 waitpid 호출한 영역에서 일시중지 된다. 즉, 특정 자식 프로세스를 기다리기 위해 사용된다. pid 인수로 지정된 자식의 상태가 변경 될 때까지 기본적으로 waitpid () 는 종료 된 자식 프로세스을 기다리지만, 아래에 설명 된 options 인수를 통해 수정할 수 있다.

또한, wait()와의 차이점은 wait()는 blocking 함수이기 때문에 자식 프로세스를 보고 있으면 다른 자식 프로세스는 볼 수가 없다. 반면, waitpid 는 non-blocking 함수이기 때문에 여러 자식의 상태도 모두 확인할 수 있다. 예로 waitpid 는 콜센터가 굉장히 바쁠 때 연락처를 적어두고 다음에 연락을 주듯, 여러 프로세스들이 대기를 하고 번호를 부여해서 순차적으로 처리를 하는 방식을 사용하여 모든 자식 프로세스의 상태를 확인해준다.

>waitpid(**-1**, &status, 0)에서 사용하는 pid 인자 값의 의미

< -1 : pid 의 절대값과 동일한 프로세스 그룹 ID 의 모든 자식 프로세스의 종료를 기다린다.

-1 : 모든 자식프로세스의 종료를 기다린다.

0 : 현재 프로세스의 프로세스 그룹 ID 를 가지는 모든 자식 프로세스의 종료를 기다린다.

> 0 : pid 값에 해당하는 프로세스 ID 를 가진 자식 프로세스의 종료를 기다린다.

>waitpid(-1, &status, **0**)

WNOHANG : 종료되지 않은 어떤 자식 프로세스도 즉시 리턴한다.

WUNTRACED : pid 에 해당하는 자식 프로세스가 멈출 상태일 경우 그 상태를 리턴한다.
프로세스 종료 뿐만 아니라 프로세스의 멈춤 상태도 찾아낸다.

WIFEXITED(status) : 자식이 정상적으로 종료되었다면 참이다. 0 이 아닌 값을 리턴한다.
WEXITSTATUS(status) : 자식 프로세스가 정상 종료되었을 때 반환한 값으로 자식의 종료 상태를 알려준다.
WIFSIGNALED(status) : 자식 프로세스가 시그널에 의해 종료되었다면 참이다. (비정상 종료가 참)
WTERMSIG(status) : 자식 프로세스를 비정상 종료하도록 한 시그널의 번호를 반환한다.
WIFSTOPPED(status) : 자식 프로세스가 중단되었다면(멈추어 있다면) 참이다.
WSTOPSIG(status) : 자식 프로세스를 멈춤상태로 만든 시그널 번호를 반환한다.
WCOREDUMP(status) : 코어덤프 파일이 있으면 참이다.

> NON Blocking

처리할 일이 많으면 예약해놓고 순차적으로 처리해준다.

core dump

어떤 프로그램의 메모리 사용 상태를 특정 시점에서 기록한 것이로, 주로 파일로 만들어진다. 즉, 프로그램이 Signal 을 받아서 비정상 종료되면 어떻게 종료되었는지 알려주는 파일을 만들 수 있다.

signal()

#incldue<signal.h>가 필요하다. 시그널을 시스템 콜로 매뉴얼과 비슷하게 생각하면 된다. 어떤상황에서 어떻게 해야할지 지침을 정하는 메뉴얼이다. 어떤 상황에서 어떤 동작을 시킬 것인가. 시그널은 행동지침을 등록해 놓은 것이다. 즉, 어떤상황이 발생해도 그에 맞는 상황을 정의할 수 있다. 비동기 처리를 해준다.

signal (int signum, 함수포인터) 로 사용된다. 설정 옵션으로는 정할 수 있는데
SIG_DFL : 기존방법 SIG_IGN : 시그널 무시 함수이름 : 시그널이 발생하면 지정된 함수를 호출
과 같다.

exec 계열의 함수

#include <unistd.h>가 필요하다. exec 에 l, v, e, p 등의 알파벳이 붙여서 사용한다.
l, v : argv 인자를 넘겨줄 때 사용한다. l 일 경우는 char *로 하나씩 v 일 경우에는 char *[]로 배열로 한번에 넘겨준다.
e : 환경변수를 넘겨줄 때 사용한다. e 는 v 와 같이 char *[]로 배열로 넘겨준다.
p : p 가 있는 경우에는 환경변수 PATH 를 참조하기 때문에 절대 경로를 입력하지 않아도 된다.

> execlp 함수는 첫번째 인자의 파일로 메모리 레이아웃을 바꾸고, 해당 파일경로의 프로세스로 갈아탄다. 프로세스를 새로 만드는 것이 아니다. Fork 가 분신술이라면 이 함수는 둔갑술이다.

#include<unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0); path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. 관례적으로 arg0에는 실행 파일명을 지정한다. execl함수의 마지막 인자로는 인자의 끝을 의미하는 NULL 포인터((char*)0)를 지정해야 한다. path에 지정하는 경로명은 절대 경로나 상대 경로 모두 사용할 수 있다.
int execv(const char *path, char *const argv[]); path에 지정한 경로명에 있는 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]); path에 지정한 경로명의 파일을 실행하며 arg0~argn과 envp를 인자로 전달한다. envp에는 새로운 환경변수를 설정할 수 있다. arg0~argn을 포인터로 지정하므로, 마지막 값은 NULL 포인터로 지정해야 한다. Envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
int execve(const char *path, char *const argv[], char *const envp[]); path에 지정한 경로명의 파일을 실행하며 argv, envp를 인자로 전달한다. argv와 envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
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.
int execvp(const char *file, char *const argv[]); file에 지정한 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

데몬 프로세스

데몬 프로세스에서 중요한 것은 생명력이다. 데몬 프로세스는 잘 죽지 않는다. 데몬 프로세스가 데몬인 이유 중 하나는 생명력 때문이다. 일반 프로세스는 터미널 등을 끄면 죽게 되나 데몬 프로세스는 아니다. 그렇기 때문에 서버 서비스에서 많이 사용되고 있다. 서버 같은 경우에는 어떠한 경우에도 살아있어야 하기 때문이다. 데몬 프로세스가 되기 위한 조건은
1. 부모 프로세스가 종료되어야 한다.
2. 데몬 프로세스와 관련된 모든 파일이 닫혀있어야 한다.

3. 터미널을 종료해도 데몬 프로세스는 살아있어야 한다.
는 것이다. 데몬 프로세스의 pts 가 물음표로 잡혀있는 이유이다. 소속이 없기 때문이다. 그렇다면 데몬 프로세스는 어떻게 죽여야 하는가? 신의 시그널인 kill -9 [데몬 프로세스의 pid]로 죽인다. 데몬 프로세스는 어떠한 명령어로도 제거되지 않으며, 오로지 kill -9 에 의해서만 제거가 된다.

pts – 섹션 아이디
파일 -디스크의 추상화
프로세스 -cpu 의 추상화

학습 예제

```
#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)        // 정상종료인지 비정상인지 보려는 함수
{    if(WIFEXITED(status))            // 자식 프로세스가 정상적으로 종료 되었다면 참이다.
      printf("(exit)status : 0x%x₩n", WEXITSTATUS(status));
// 자식 프로세스가 정상 종료 되었을 때 반환한 값을 출력한다.
    else if(WTERMSIG(status))
 // 비정상 종료일 경우, 자식 프로세스를 종료하도록 한 시그널의 번호를 반환한다.
      printf("(signal)status : 0x%x, %s₩n", status &0x7f, WCOREDUMP(status) ? "core dumped" : "");
 // 이 프로세스가 시그널로 죽게 되면 어느 메모리 상태에서 어떻게 죽었는지 기록하는 파일로 이를 기록할지 말지 정하는 것이 core dump 비트이다. 1 이 세팅 되어 있으면 (WCOREDUMP()가 1 이면)core dumped 가 있다는 것이다.
}

int main(void)
{  pid_t pid;
   int status;
   if((pid = fork()) > 0 )
   {     wait(&status);
     term_status(status);                }
   else if( pid == 0)
     abort();              //시그널 6 번, 비정상 종료
   else
   {      perror("fork()");
          exit(-1);               }
   return 0;                                    }
```

결과
(signal)status : 0x6, core dumped

```
#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)  //위의 예제와 같은 작동을 하는 함수
{   if(WIFEXITED(status))
      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);       //status 로 SIGCHLD 가 전달된다.
    term_status(status);
// 위 코드를 이용해서 정상인지 비정상인지 확인한다. 정상종료로 숫자 0 이 나오는데 리턴 0 을 했으므로
}

int main(void)
{  pid_t pid;
    int i;
  signal(SIGCHLD, my_sig);       // SIGCHLD 가 오면 my_sig 함수를 호출하라는 뜻이다.
  if((pid = fork()) > 0 )
    for(i = 0; i < 10000; i++)
  {    usleep(50000);               //마이크로 세컨드 단위.
       printf("%d₩n", i + 1);                          }
  else if( pid == 0)
    sleep(5);
//초단위. 5 초 뒤 자식 프로세스 죽어서 부모 프로세스로 간다. 그러나 부모는 계속 돌고 있다. 어떻게 될까?
 시그널을 설정해놨으므로 my_sig 호출한다.
  else
  {     perror("fork()");
         exit(-1);                        }
  return 0; // 해당 프로세스는 정상 종료 되었음을 나타낸다.
}
```

위 예제는 비동기 처리이다. 어떤 상황에 대해서도 대처가 가능하다는 것을 나타낸다.
결과는 쭉 숫자가 1 부터 써지다가
99
(exit)status : 0x0
100
가 나타나면서 계속 숫자가 써진다. 이는 부모 프로세스가 마이크로 세컨드 단위로 자고 일어나서 출력을 하는데 이는 0.005 초이다. 반면 자식프로세스는 5 초 뒤에 죽기 때문에 100 번 정도에 my_sig 함수가 호출 된 것이다.

Execve 학습 예제

```
#include <unistd.h>
int main(void)
{   execlp("ps", "ps", "-e", "-f", 0); // 0 은 NULL 문자이다.
    return 0;                                   }
```

결과는 ps -e -r 와 같다

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{   execlp("ps", "ps", "-e", "-f", 0);
    printf("after₩n");
    return 0;                                   }
```

결과는 after 가 출력이 안 된다. 왜 일까? Fork 가 분신술이라면 execlp 은 둔갑술이다. 메모리의 형태가 바뀐다. 메모리의 레이아웃을 ps 로 바꾼다. 현재 프로세스를 ps 로 대체해서 execlp 아래의 printf 명령어가 실행되지 않는다. 따라서 after 를 출력하려면 아래의 예제와 같이 만들어준다.

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

 return 0;                                          }
```

결과는

```
root       6337      2  0 14:42 ?        00:00:00 [kworker/u16:6]
root       6345      2  0 14:44 ?        00:00:00 [kworker/0:0]
sue1000+   6351   4443  0 14:44 pts/0    00:00:00 ./a.out
sue1000+   6352   6351  0 14:44 pts/0    00:00:00 ps -e -f
prompt >
```

-newpgm 프로세스 코드

```
#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;                                                    }
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(void)
{   int status;
   pid_t pid;
   if((pid = fork()) >0) // 부모 프로세스
   {     wait(&status);     // 끝나면 자식 프로세스가 죽고 밑에 글을 출력하고 끝난다.
          printf("prompt >");                    }
   else if(pid == 0) // 자식 프로세스를 둔갑시킨다
     execl("./newpgm","newpgm","one","two", (char *)0);
//이제 우리가 만든 프로그램을 동시다발적으로 돌려보겠다는 뜻이다.
  return 0;                                                  }
```

결과는

```
rgv[0] = [newpgm]
argv[1] = [one]
argv[2] = [two]
prompt >
```

문제 예제

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(void)
{   int status;
   pid_t pid;
```

```
   char *argv[] = {"./newpgm", "newpgm", "one", "two", 0};
   char *env = {"name = OS_Hacker", "age = 20", 0};
   if((pid = fork()) >0) // 부모 프로세스
   {      wait(&status);
          printf("prompt >₩n");                }
   else if(pid == 0) // 자식 프로세스를 둔갑시킴
     execve("./newpgm", argv, env);
  return 0;                                    }          를 정상작동하게 만들어라.
```

envp 가 시스템 환경변수이다. 답은 **newpgm.c 를 변경**한다.

```
#include <stdio.h>
int main(int argc, char **argv,  char **envp)
{   int i;
          for (i=0; argv[i]; i++)
              printf("argv[%d] = [%s]₩n",i,argv[i]);
          for (i=0; envp[i]; i++)
               printf("envp[%d] = [%s]₩n", i, envp[i]);
  return 0;                                                    }   로 newpgm 을 변경한다.
```

```
#include <stdio.h>
int main(void)
{   system("date");  //내부적으로 fork 를 한 다음에 execve 를 하는 것이다.
   printf("after₩n");
   return 0;                }
```

결과 system("date")는 날짜, 요일, 시간을 출력한다.

2018. 03. 26. (월) 16:14:07 KST

after

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int my_system(char *cmd)
{   pid_t pid;
     int status;
    char *argv[] = {"sh", "-c", cmd, 0}; // "sh"는 쉘, "-c" 해당 커맨드 실행 cmd 실행
    char *envp[] = {0};
    if((pid = fork()) > 0)
       wait(&status);
    else if(pid == 0)
       execve("/bin/sh", argv, envp); } //쉘을 만들어서 자식 프로세스가 실행하게 된다. 즉, "/bin/sh" 경로로
                                          들어가서 argv, envp 를 인자로 전달한다.
```

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

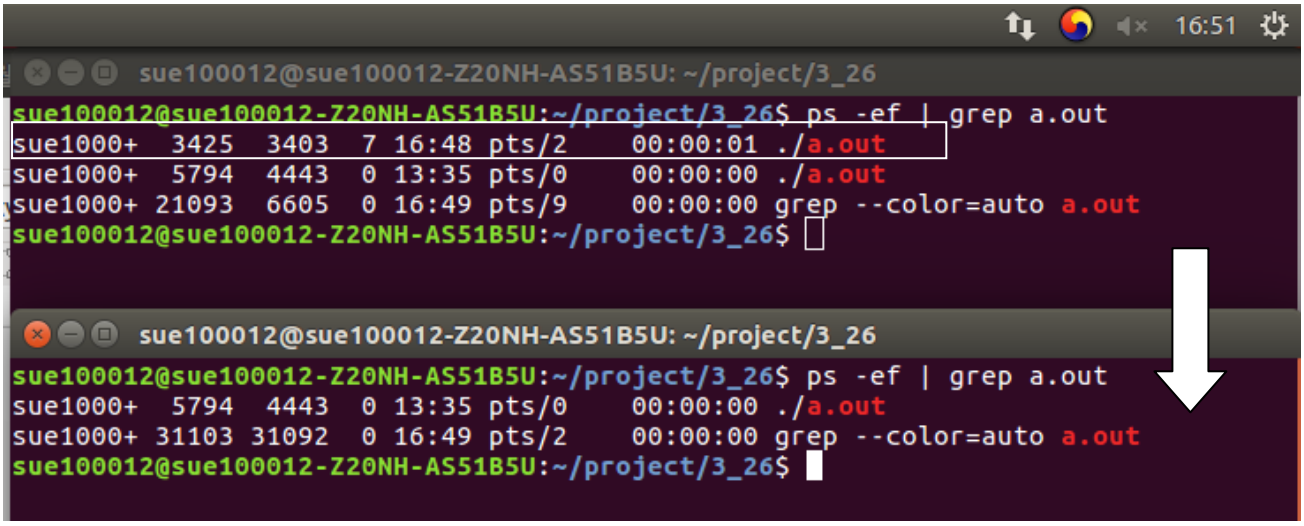
결과

```
Mon Mar 26 16:20:43 KST 2018
after
```

데몬 프로세스 학습 예제

```
#include <stdio.h>
int main(void)
{   for(;;)
   {   system("date");
       printf("after₩n");              }
   return 0;                                  }
```

기본 예제에 for(;;) 삽입하여 무한루프로 돌리는 코드이다. ps –ef | grep a.out 명령어로 프로세스가 돌아가던 터미널 창을 끄면 pts 사라지는 거 확인할 수 있다. 이는 프로세스가 터미널과 생명을 같이하기 때문이다.



데몬 프로세스는 기본적으로 pts 가 물음표로 잡혀 있다.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>
int daemon_init(void) // 데몬 프로세스를 만들고 있다.
{   int i;
   if(fork() > 0 ) // 부모 프로세스
```

```
    exit(0);       // 부모 프로세스 죽임
   setsid();       // 새로운 세션을 만드는 함수이다. 데몬 프로세스 만들어 주는 것이다.
                      소속을 없애 주는 것이다. 터미널을 꺼도 죽지 않게 하기 위함이다. 이처럼 부모 프로세스를
                      죽이고 setsid()를 하면 pts 가 ?가 된다. 소속을 잃었기 때문이다.
   chdir("/");     //데몬이 구동하다 보면 어떤 파일이든지 접근해야 할 수 있다. 그래서 디렉토리 루트로 이동
   umask(0);     //권한 설정. 루트에 있는 모든 것을 사용할 수 있게 해준다.
   for(i=0;i<64;i++)
    close(i);    //데몬은 자식이므로 부모의 것들을 상속받는다. 그러나 데몬이 되면서 연을 끊고
                      다 close 하는 것이다. 리눅스는 기본적을 64 개가 열려 있어 범위가 64 가  된 것이다.
   signal(SIGCHLD, SIG_IGN); // 데몬이 혹시나 자식프로세스를 만들 수도 있기 때문에 만든 것이다. 지금
                                    이건 자식프로세스가 죽어서 시그널 보내도 무시하라고 만든  것이다.
   Return 0;                  }

int main(void)
{    daemon_init();
   for(;;); // 계속 돌고 있는 것이다. 데몬 프로세스가 종료되지 않도록 해준다. for(;;){ }안에 계속 돌아야 하는
         내용을 쓰면 된다. 서비스 포털들이 데몬을 쓰고 있는 이유이다.
   return 0;                  }
```

데몬을 끄는 방법도 있다.

```
#include <signal.h>
#include <stdio.h>
int main(void)
{    signal(SIGINT, SIG_IGN);
     signal(SIGQUIT, SIG_IGN);
      signal(SIGKILL, SIG_IGN); //데몬 프로세스인데 시그널로 죽이는 것도 막아놓았다.
      pause();
     return 0;      }
```

데몬 프로세스를 죽일 수 있는 방법은?

위의 예제는 시그널을 싹 다 막아놓았다고 가정해 본 것이다. 이렇게 다 막아놓으면 정말 데몬 프로세스를 죽일 방법이 없을까? 그렇지 않다. 데몬보다 더 쎈 것이 있는데, 그 것은 신이다. 유일하게 신이 날리는 시그널이 있는데, SIGKILL 이다. 이것은 위처럼 막아놓아도 막아지지 않는다. 즉, 데몬 프로세스는 어떠한 명령어로도 제거되지 않으며, 오로지 kill -9 에 의해서만 제거가 된다. kill -9 [데몬 프로세스의 pid]를 실행하면 해당 pid 의 데몬 프로세스가 제거된다. Pkill -9 a.out 을 실행하면 모든 데몬 프로세스가 제거된다.