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

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 장성환
redmk1025@gmail.com

wait 는 블록함수.

블록함수 이기 때문에 빈번한 자식프로세서가 오작동이 일어날 경우
종료가 바로바로 처리되지 않고 좀비 프로세서가 생성이 된다.

그게 많아진다는 것은 task_struct 가 많이 생성이 되어 메모리를 많이 차지하게 된다.

이것을 효과적으로 처리하기 위한 방법은 무엇인가 ?
뭔가 비동기식 처리 및 논 블록킹 방식으로 처리를 해야 한다.

sigchld 를 맞은 애들이 막 들어오면, 순번을 기록하고 할일 끝내고 순번대로 처리를 해줄게 라고 한다면
예약이 되었기 때문에 처리가 가능하다 (논 블록킹 방식)

waitpid() - 논 블록킹 wait 함수

demon process

데몬이란 녀석의 생명력이 중요하다.
일반 프로그램은 쉽게 죽지만 데몬은 잡초처럼 끈질기다.

예를 들어,
어떤 서비스를 할 때, (프로그램 구동하고 있을 때) 펌웨어를 껐더니 프로세스가 죽어버린다.

데몬 프로세서의 효과을 확인하기 위해서
이전의 system2.c 파일을 무한루프로 함수 호출하도록 변경하고
ps -ef |grep a.out 으로 pts 확인

이후 실행프로세스 터미널 닫고 새로운 터미널 열어서 ps 반복하면 pts 가 없다.
(원래는 터미널 꺼도 프로세서는 살아 있어야 하는데 껏다고 해서 죽어버렸다.)

따라서 견고한 SW 를 위하여
터미널을 꺼도 죽지않도록 해야한다. 이것을 데몬 프로세스
pts 넘버가 아니라 ? 가 뜨는데 이게 데몬 프로세스이다.

이것을 죽이는 방법은 단 하나
kill -9 [pid]

파일은 하드디스크의 추상화 프로세스는 cpu 의 추상화

* execve.c

```
#include <unistd.h>

int main(void){
   execlp("ps","ps","-e","-f",0);
// 프로그램이름, 인자, 플래그..., 0(여기가 끝이라는 뜻)
   printf("After\n");
//After 가 출력이 되지 않는 이유는 뭘까 ?
// exe 은 둔갑술이다. 형태가 바뀌는 것이다.
//메모리 레이아웃을 ps 로 바꾸었기 때문에
// printf 가 없어진 것이다. (ls 만든 사람이 그렇게 만듬)
// a.out -> ls 로 바뀌었기 때문이다.
// fork 는 분신술
// printf 도 동작시키려면 분신술로 1 개를 더 만든뒤에 분신을 둔갑 시키면 된다.
   return 0;
}
```

execlp() 함수

첫 인자는 실행파일명 (./a.out 등등)
그 다음 인자는 연속적인 문자열들이 오게 된다.
(argv 의 인자들을 생각하면 된다.)

해당 함수의 특징으로
실행 파일이 되어 버리면, 프로세서가 종료 되어 버린다.

따라서 printf 함수가 동작하지 않는다.

*execve2.c

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

int main(void){
   int status;
   pid_t pid;
   if((pid = fork()) > 0){
          wait(&status);
          printf("prompt >\n");
   }
   else if(pid == 0)
          execlp("ps", "ps", "-e", "-f", 0);
   return 0;

   /*
     인수 중복 이유
   프로그램 이름을 중복해서 입력했는데,
   이는 프로그램을 실행하면 첫번째 인수가 실행한 프로그램의 전체 이름이기 때문.

    */

   // exe file 을 동작시키며 나머지 동작도 완료하기 위해서는 fork 함수로 프로세서를 하나 더
   // 생성한 뒤에 생성 된 자식 프로세서로 실행을 해야 한다.
}
```

ps -e -f 는 터미널 창에서
현재 돌아가고 있는 프로세스들을 확인 할 수 있는 명령어이다.

execlp()함수에 해당 인자들을 넣어서 구현하였다.

execve.c 파일과 다른점은
printf()함수가 동작한다는 점이다.

즉, execlp 함수를 동작 시키고도 프로세스가 종료되지 않고 printf()함수를 동작 시킨다.

부모 프로세스는 자식 프로세스를 생성하여
execlp 함수를 구동 시키며 종료하게 하고,

wait 함수를 통하여 자식 프로세스가 종료 될때까지 대기 한후에
printf 함수를 구동시킨다.

설명 자식 프로세스 작업이 끝날 때 까지 대기하며, 자식 프로세스가 종료한 상태를 구합니다.
wait()함수를 실행하면 자식 프로세스가 종료될 때가지 대기하게 됩니다. 만일 자식 프로세스가 정상 종료하여, main()에서 return으로 값을 반환하거나, 또는 exit()로 값을 반환하며 정상 종료했다면 wait(int *status) 에서 status의 변수 값의 상위 2번째 바이트에 반환 값을 저장합니다.
또는 어떤 시그널에 의해 종료되었다면 최하위 바이트에 시그널 번호가 저장됩니다. 즉,

	8비트	8비트
정상 종료	프로세스 반환 값	0
비정상 종료	0	종료 시킨 시그널 번호

헤더 wait.h

형태 pid_t wait(**int** *status)

인수 **int** status 자식 프로세스 종료 상태

반환 **pid_t** 종료된 자식 프로세스 ID

*wait10.c

```
#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){
  if(WIFEXITED(status))// WIFEXITED(status) : 자식 프로세스가 정상적으로 종료시 true 리턴
        printf("(exit)status : 0x%x", WEXITSTATUS(status));
  // WEXITSTATUS(status) : 자식 프로세서 정상 종료될때, 반환한 값. 즉, status >> 8 과 같은 값이다.
  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);
/*
    while(waitpid(-1, &status, WNOHANG) > 0) // (WNOHANG) 자식이 죽은게 없으면 0 이 되어 빠져나온다.
    term_status(status);

    // 이것이 wait 의 논 블록킹 처리 방식
```

WIFEXITED(status)

자식 프로세스가 정상적으로 종료시 true 리턴

WEXITSTATUS(status)

자식 프로세스가 정상 종료시, 반환한 값이다.
즉 정상종료시 상위 2 비트의 값을 보기 위하여 쓴다.
status>>8 과 같은 값을 나타낸다.

status 는 자식 프로세서 종료 상태를 나타낸다.

비동기 처리의 방식으로 signal()함수를 이용한다.
SIGCHLD 는 자식 프로세스가 종료 상태가 되면 알림

설명

시그널 처리를 설정합니다. 즉,

- 어떤 시그널이 발생하면 기존 방법으로 처리할지
- 아니면 무시하게 할지
- 프로그램에서 직접 처리할지를 설정할 수 있습니다.

설정하는 옵션에는 위의 분류에서처럼 3가지가 있습니다.

유형	의미
SIG_DFL	기존 방법을 따른다.
SIG_IGN	시그널을 무시한다.
함수이름	시그널이 발생하면 지정된 함수를 호출한다.

헤더	#include <signal.h>
형태	**void** (*signal(**int** signum, **void** (*handler)(**int**)))(**int**);
인수	**int** signum 시그널 번호 **void** (*handler)(**int**) 시그널을 처리할 핸들러
반환	**void** *()(**int**);이전에 설정된 시그널 핸들러

```
   */
   term_status(status);
}

int main(void){
   pid_t pid;
   int i;
   signal(SIGCHLD, my_sig); // SIGCHLD 와 함수의 주소가 인자로 들어간다.
   // 돌발상황에 대처하기 위한 매뉴얼이 필요하다.
   // 시그널이라는 함수는 이러한 행동 지침을
// 등록해 놓는 것이다. (signal = manual)
   // 즉, SIGCHLD 라는 상황이 발생하면 my_sig 가 동작하도록 하는 것이다.
   // 자식이 죽으면 SIGCHLD 발생

   if((pid = fork()) > 0)
         for(i=0; i<10000; i++){
            usleep(50000); // 마이크로 초 단위 = 0.05 sec
            printf("%d\n", i+1);
         }
   else if(pid == 0)
         sleep(5); // 초단위
   // 자식이 만들어 지자마자 5 초 잔다.
   // 자식이 종료되면 시그널이 동작한다.
   else{
         perror("fork() ");
         exit(-1);
   }
   return 0;
}

// 중요! signal 을 통해서 상황에 따라서 그 상황을 제어 할 수 있다. (비동기 처리)
```

설명 wait()함수처럼 자식 프로세스가 종료될 때까지 대기합니다. 차이점은 wait() 함수가 자식 프로세스 중 어느 하나라도 종료되면 복귀되지만, 즉 대기에서 풀리지만 waitpid()는 특정 자식 프로세스가 종료될 때까지 대기 합니다.
또한 wait()는 자식 프로세스가 종료될 때까지 block되지만 waitpid()는 WNOHANG 옵션을 사용하면 block되지 않고 다른 작업을 진행할 수 있습니다.

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

첫번째 인수 pid_t pid는 감시 대상인 자식 프로세스의 ID입니다. 이 ID값에는 자식 프로세스 ID외에도 아래와 같은 값을 지정할 수 있습니다.

pid	설명
-1	여러 자식 중 하나라도 종료되면 복귀, wait()와 같은 처리
0	현재의 프로세스의 그룹 ID와 같은 그룹의 자식 프로세스가 종료되면 복귀
양수	pid에 해당하는 자식 프로세스가 종료되면 복귀

3번재 인수 options는 여러 상수 값이 있습니다만 WNOHANG와 0을 많이 사용합니다.

options	설명
WNOHANG	자식 프로세스가 종료되었는지 실행 중인지만 확인하고 바로 보귀. 즉, 부모프로세스는 block되지 않음
0	자식 프로세스가 종료될 때까지 block됨. 즉, wait()와 같은 처리

WNOHANG를 사용했을 경우 waitpid()는 자식 프로세스의 종료 상태를 확인하고 바로 복귀합니다. 만일 자식 프로세스가 활동 중이라면 0을 반환하고, 종료되면 자식 프로세스의 ID가 반환됩니다.

헤더 wait.h

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

인수

pid_t pid	감시할 자식 프로세스 ID
int *status	자식 프로세스의 종료 상태 정보
int options	대기를 위한 옵션

반환 **int**
정상: 종료된 자식 프로세스 ID
실패: -1
WNOHANG를 사용했고 자식 프로세스가 종료되지 않았다면: 0

waitpid() 함수는 논블록킹 함수이다. 따라서 wait 함수와 달리 부모 프로세스가 같이 동작할 수 있다. (적절한 옵션 구현 WNOHANG)

다만 wait 함수도 signal 비동기식 처리 방법을 통하여
논 블록킹과 같이 역할을 수행 할 수 있다.

*term_status.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))
         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" :"");
   // 프로그램이 시그널 맞아 죽으면 맨 앞이 코어덤프라서 0x7f 를 해준다.
   // 코어덤프는 프로그램이 비정상으로 끝났을때, 어느 메모리에서 끝났는지 그것을
덤프에서 알려줄지 결정
   // 1: 코어덤프 0: 노 코어덤프
}

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

   if(pid = fork() > 0){
         wait(&status);
         term_status(status);
   }

   else if(pid == 0)
```

abort()

강제로 비정상 종료를 시킨다.

```
        abort();
  else{
        perror("fork() ");
        exit(-1);
  }

  return 0;
}
```

```
* system.c

#include <stdio.h>

int main(void){

  system("date");

  system("ls");
  printf("after\n");

  // system 함수는 내부적으로 fork 를 한 후에 exe 하는 것!
  return 0;
}
```

이전까지 execlp 같은 함수는 자체적으로 자식 프로세스를 생성하여
함수를 구동 시켜야 나머지 코드가 부모 프로세스에 의하여 실행이 가능했다.

하지만 system 함수는 약간 성격이 다르다.

내부적으로 자체 fork() 기능을 통하여 알아서 동작하도록 할 수 있다.

다만 system 함수는 이 함수의 동작이 끝나야만 다음 코드가 실행 가능하다.
번거롭게 자식 프로세스를 생성할 필요는 없지만, 종료될때 까지 기다리는 것이 싫다면 fork 를 이용하여 자식 프로세스에서 실행하도록 하자!

* system2.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}; // sh = shell // -c :해당 커맨드를 실행해
라.
   char *envp[] = {0};

   if((pid = fork()) > 0)
         wait(&status);
   else if(pid == 0){
         execve("/bin/sh",argv,envp); // 자식이 date 를 실행하게 된다.
   } // 여러개의 exe 를 실행 하려면, for 문을 통하여 여러번 포크 여러번 exe
}

int main (void){

   for(;;){
   my_system("date");
   sleep(1);
   }
   //my_system("ls");
   printf("after\n");
   return 0;
}
```

여기서 눈여겨 볼 것은
첫번째, 프로세스는 쉽게 종료가 된다는 것
터미널을 닫음으로서 ptr 이 쉽게 사라진다. (ps -ef |grep a.out 으로 확인)

execlp() 와 execve()의 다른점을 볼 수 있다.

execlp 는 문자열을 인자로 입력이 가능하다. 하지만
execve 는 문자열 배열을 입력이 가능하다. (환경 변수도 전달 가능)

*newpgm.c + newpgm_sub.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;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main (void){
   int status;
   pid_t pid;
   if((pid = fork()) > 0){
          wait(&status);
          printf("prompt >");
   }
   else if(pid == 0)
          execl("./newpgm", "newpgm", "one", "two", (char*)0);
   // 만든 프로그램을 동시 다발적으로 구동 가능!
   // 자식 프로세서를 exe 파일 구동에 이용
   return 0;
}
```

지금까지는 리눅스 명령어를 실행을 했다면
이번 시스템 프로그래밍은 내가 만든 프로그램을 실행 가능하게 한다.

작동법은 간단하다.
실행할 프로그램을 미리 gcc -o 로 생성해 놓고
그 이름을 execl 이나 execve 로 불러오면 된다.

*tmp_sig.c

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

int main(void){
   signal(SIGINT, SIG_IGN); //SIGINT = ctrl+c 를 무시하겠다.
   signal(SIGQUIT, SIG_IGN);
   signal(SIGKILL, SIG_IGN);
   pause();
   return 0;
}

//만일 데몬이 이런식으로 signal 으로 명령어를 무시하면 천하무적이 된다.
//죽일 수 있는 방법은 ?
//유일하게 신이 날리는 시그널이 있다. SIGKILL
//SIGKILL 을 무시한다고 해도 먹힌다.
//kill -9 [pid]
//이렇게 날리면 다 죽는다.
```

signal() 함수는 어떤 특정한 이벤트가 발생하였을때, 이에 대한 작동을 정의한다.
즉, 비동기 처리가 가능케 하는 함수이다.

각각 ctrl+c 나 kill 명령어 등
프로그램을 종료하는 명령어를 무시하도록
SIG_IGN 을 이용하여 비동기처리를 정의하였다.

(다만 SIGKILL 이벤트는 sinal()함수로도 정지가 불가능하다.)

*demon_process.c

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

int demon_init(void){

   int i;
   if(fork() >0) // parents 를 죽임
         exit(0);
   setsid(); // 일반 프로세서는 tts 숫자 (가상 터미널) session id
   // 프로세서가 터미널과 생명을 같이하는데 이렇게 setsid()를 하면 소속이 없어진다.
   //이러면서 tti 에 ? 가 생긴다.
   chdir("/");
   umask(0); //모든 권한 설정 root 에 있는 모든것을 사용할 수 있게 한다 (신경안써도 된다.)

   for(i=0; i<64; i++)
         close(i); //close 하는 이유는 자식이니까 부모의 정보를 받게 되는데 그것을 다 close 하는 것
   // 0 표준입력 1 표준출력 2 표준에러 등등
   // 기본적으로 64 개를 가지고 있다 리눅스는.

   signal(SIGCHLD, SIG_IGN);
   // 혹시라도 데몬이 자식프로세서를 만들 수 있다.
   // SIGCHLD 어떠한 동작지침 자식이 죽는다면 이라는 것
   // SIG_IGN 자식이 죽던말던 신경 안쓴다.
```

!!!!!!!!!!데몬 프로세스!!!!!!!!!!

setsid()

1. setsid()를 호출한 프로세스는 새로운 세션을 생성하고, 그 세션의 리더가 된다.
2. setsid()를 호출한 프로세스는 새로운 프로세스 그룹을 생성하게 되고, 그 프로세스 그룹의 그룹리더가 된다.
3. setsid()를 호출한 프로세스는 새로운 세션을 만듦으로서, 이전 세션의 제어 터미널을 사용할 수 없다. 따라서 제어 터미널을 가지고 있지 않다.

* setsid()는 프로세스 그룹 리더가 아닌 자식 프로세스가 함수를 호출해야 한다.
부모가 호출할 경우에는 에러(-1)를 리턴하게 된다. 정상 호출인 경우에는 호출 프로세스의 세션 아이디를 리턴한다.

데몬 프로세스는 입출력에 영향을 받지 않으므로
모든 입출력을 close 해준다.

리눅스는 보통 64 개의 입출력을 가지고 있으므로 모두 닫아준다.

데몬 프로세스 동작여부는 system2.c 에서 기술한 내용과 같다.
? 라는 희안한 프로세스가 나오게 된다. (ps -ef |grep a.out)
터미널을 닫아도 계속해서 동작하게 된다.

```
   return 0;
}

int main(void){
   demon_init(); //demon 프로세스 생성
   for(;;)
          sleep(1);

   return 0;
}
//영상처리 프로그램을 만들 경우에 데몬으로 생성해야 한다.
//이렇게 하지 않으면, 터미널이 꺼질 경우에 제대로 작동하지 않아서
//자율주행 등등의 역할을 못하게 된다.
```