Xilinx Zynq FPGA,TI DSP,
MCU 기반의
프로그래밍 전문가 과정

날 짜 : 2018 . 3. 22

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

<ls_module.c>

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<dirent.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<string.h>
// 미리 함수를 정의 해준다. 나중에 개발 시에 어떤 함수가 쓰였는지 혹은 어떤 함수를 개발 할지 미리 정의를 하는 용도
void recursive_dir(char *dname);

int main(int argc, char *argv[])
{
	//  '.' 현재 디렉토리 이름을 가져온다.
	recursive_dir(".");
	return 0;
}

void recursive_dir(char *dname)
{
	struct dirent *p;
	struct stat buf;
	DIR *dp;
	// 현재 작업할 디렉토리를 인자에 있는 것으로 바꾼다.(시스템콜 명령어)
	chdir(dname);
	// 현재 디렉토리를 열고 그 값을 변수에 저장한다. 리턴은 주소가 나온다.
	dp = opendir(".");
	printf("\t%s:\n",dname);
	//현재 열어둔 주소를 디렉토리 읽는 함수로 받고 읽을게 있으면 반복
	while(p = readdir(dp))
		printf("%s\n",p->d_name);
	rewinddir(dp); // 되감기 한다. 읽어서 뒤로간 포인터의 위치를 맨앞으로 가져다 준다.
	while(p = readdir(dp))
	{
		stat(p->d_name, &buf);
		if(S_ISDIR(buf.st_mode))
			if(strcmp(p->d_name,".") && strcmp(p->d_name,".."))
			// 0 일때 돌아가는거니 둘다 아닐 때만 리컬시브로 들어가란 뜻이다.
				recursive_dir(p->d_name);
	}
	// <시험 문제> 상위돌다가 바꿔야하는 프로그램.
	chdir("..");	// 디렉토리를 상위 폴더로 이동해라 라는 뜻
	closedir(dp);
}
```

<fork1.c>

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

int main(void)
{

	printf("before\n");
	// fork() 함수는 현재 줄부터 밑에 줄을 쭉 복사해서 자식 프로세스에서 분신처럼 복사되어 동작하게 하는 녀석이다.
	fork();
	printf("after\n");
	return 0;
}
```

<fork2.c>

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

int main(void)
{
	// pid_t 는 int 형이다.
	pid_t pid;
	// 자식의 fork()함수는 자식프로세스의  pid 값을 반환한다.
	pid =fork();
	// 첫번째로 돌때는 복사된 자식의 pid 값이 있다. 그래서 if 쪽에 걸린다.
	if(pid>0)
		printf("parent\n");
	// 두번째로 돌때는 복사된 자식의 자식 pid 값은 없으므로 0 이기에 else if 문으로 들어오게 된다.
	else if(pid==0)
		printf("child\n");
	// 이도 저도 아니면 에러.
	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}
```

<fork3.c>

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


int main(void)
{

	printf("자식 나오라\n");
	pid_t pid;
	// fork() : 자식(분신 1...) 프로세스를 생성해서 그 값을 반환한다.
	// 프로세스가 두개이다. 현재 나와 그리고 자식 프로세스.
	// 원래 실행 하고 복사한거 실행하고.
	// 그래서 두번 실행 한것처럼 보인다. fork 는 복사한 것을 한번 더 프로세스로 만들어
	// 실행하니까 task_struct 가 2 개가 된거다.
	// fork 라는 선언이 있는  위치 줄에서 부터 복사가 된다.
	pid = fork();
	if(pid>0)
		// getpid() 현재 자신의 프로세스를 반환한다.
		printf("paretnt : pid = %d , cpid = %d\n", getpid(),pid);
	else if(pid == 0)
		// 자식 프로세스의 자식은 프로세스가 없으니  cpid 는 0 이 나온다.
		printf("child : pid = %d, cpid = %d\n", getpid(),pid);
	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}
```

<fork4.c>

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

int main(void)
{
	pid_t pid;
	int i;
	pid = fork();
	// 부모 프로세스쪽의 무한 루프를 돌 부분이다.
	if(pid>0)
	{
		while(1){
			for(i = 0; i<26;i++)
			{
				printf("%c",i+'A');
				fflush(stdout);
			}
		}
	}
	// 자식 프로세스쪽 무한 루프를 돌 부분이다. 오래돌면 둘의 순서가 뒤죽박죽 됨, Multi tasking 을 볼수 있다.
	else if(pid ==0)
	{
		while(1){
			for(i = 0; i<26; i++)
			{
				printf("%c", i+'a');
				fflush(stdout);
			}
		}
	}
	else
	{
		perror("fork())");
		exit(-1);
	}
	printf("\n");
	return 0;
}
```

<ps_test1.c>

```
#include<stdio.h>

int main(void)
{
	int a = 10;
	// ‘%#p’ 의 ‘#’ 은 포인터로 뽑을 때 깔끔히 뽑으려고 쓰던거 근데 요즘은 쓰지 않는다.
	printf("&a = %#p\n",&a);
	// 1000 초를 멈추어 놓겠다.
	sleep(1000);
	return 0;
}
```

<ps_test2.c>

```
#include<stdio.h>

int main(void)
{
	// 실행 파일 실행 시 인자로 ‘&’ 을 주면 그 파일의 권한 주소가 나온다. ps_test1 의 주소값 입력.
	int *p = 0x7fffdee4c734;
	printf("&a : %#p\n",*p);
	return 0;
}
```

***위의 두개 의 파일을 이용하여 두번 째꺼를 실행하면 첫번 째 파일의 메모리 권한 때문에 세그먼트 폴트가 나온다.**

***프로세스 마다 메모리의 권한이 생겨서 서로간의 메모리에 간섭을 하기 어렵다.**
(한마디로 프로세스 마다 vm 을 공유하지 않는다.)

→ 메모리의 공유를 할 해결방법:

1. message queue
2. PIPE (non block 설정에 대한 불편함이 있음)
3. 직접 권한 설정 (하지만 하려면 복잡한 절차가 따른다.)
4. IPC(inter process communication)
   → 물리 메모리를 공유 , 분할 작업을 하여 효율적 분담 처리를 한다.

<ps_test3.c>

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

// 전역 변수를 설정 한다.
int global =100;
int main(void)
{
	// 지역 변수를 설정한다.
	int local =10;
	pid_t pid;
	int i;
	//  fork() 를 통해 자식의 프로세스 pid 값을 받는다.
	pid = fork();
	if(pid>0)
		// if 에서 부모 프로세스가 실행이 된다.
		printf("global :%d , local : %d\n", global , local);
	else if(pid ==0)
	{
		// else if 에서 자식의 프로세스가 실행이 된다.
		// 하지만 전역변수는 쓸 때만( c.o.w : copy on write) 복사를 한다.
		// 한마디로 한번에가 아닌 단계별로 복사된다는 뜻.
		global++; // → 101 이된다.
		local++; // →  11 이 된다.
		printf("global : %d , local : %d\n", global, local);
	}
	else
		{
			perror("fork()");
			exit(-1);
		}
	printf("\n");
	return 0;
}
```

<ps_test4.c>

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>
#include<fcntl.h>
// 전 번에 한 단방향 채팅 프로그램. 중요한 점은 non block 을 따로 하지 않아도 된다.
int main(void)
{
	int fd , ret;
	char buf[1024];
	pid_t pid;
	// myfifo 파일을 읽기 쓰기 전용으로 open  한다.
	fd = open("myfifo", O_RDWR);
	//  자식 프로세스의 pid 가 있으면 실행 한마디로 부모 프로세스 실행
	if((pid= fork()) >0)
	{
		for(;;) // 무한 루프를 돌려서 입력한 값을 읽어 프린트 한다.
		{
			ret = read(0 , buf, sizeof(buf));
			buf[ret]=0;
			printf("Keyboard : %s\n",buf);
		}
	}
	else if(pid == 0)
	{
		// 무한 루프를 돌려서 입력한 값을 myfifo 라는 pipe 파일을 통해  부모 프로세스 터미널에 뜨게 함
		for(;;)
		{
			ret = read(fd, buf, sizeof(buf));
			buf[ret]=0;
			printf("myfifo : %s\n", buf);
		}
	}
	else
	{
		perror("fork()");
		exit(-1);
	}
	close(fd);
	return 0;
}
```

<fork5.c>

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

// 좀비 프로세스를 만든다.
int main(void)
{
	pid_t pid;
	// 부모 프로세스를 잠까 sleep 을 걸어 반응을 하지 못하게 한다.
	if((pid = fork())>0)
		sleep(500);
	// 자식 프로세스가 바로 죽어 버리고 부모에게 혼령이 되어 내 시신을 처리좀 해달라고 하는데…
	// 부모가 자고 있어서 처리를 못한다 그런 상태를 <defunct> 상태, 좀비 프로세스 상태라고 한다.
	// ps -ef |grep a.out 을 통해서 좀비 상태인지 확인이 가능한다.
	else if(pid == 0)
		;
	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}
```

```
jhb@onestar:~/My/Homework/hanbyuljung/class_22_me$ ps -ef |grep a.out
jhb        7592   5807  0 19:51 pts/18   00:00:00 ./a.out
jhb        7593   7592  0 19:51 pts/18   00:00:00 [a.out] <defunct>
jhb        7608   7594  0 19:51 pts/6    00:00:00 grep --color=auto a.out
```

<wait1.c>

```
#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() 은 child 가 시그널 상태를 보낼 때까지 기다리는 함수이다.
                wait(&status);
                // wait 으로 status 상태를 변수에 담아온다.
                printf("status: %d\n",status);
        }
        else if(pid == 0)
                // 정상 종료시에 값이 7 을 반환한다. 여기서는 안쓰는 비트 x256 이 포함되기 때문에 1790 나옴
                exit(7);
        else
        {
                perror("fork()");
                exit(-1);
        }
        return 0;
}
```

정 상 종 료

(255 위부터 공간을 쓴다)

비 정 상 종 료

(256 지점에 core dump 가 있다)

<wait4.c>

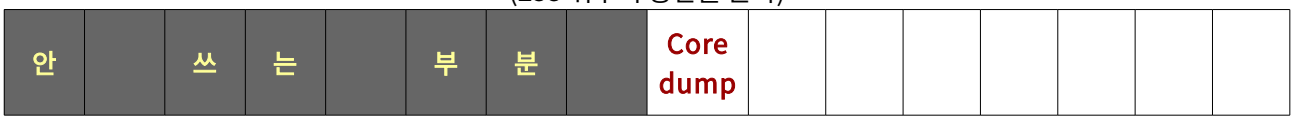
```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/wait.h>

int main(void)
{
        pid_t pid;
        int status;
        if((pid=fork()) > 0)
        {
                wait(&status);
                // 0x7f 를  엔드로 비트연산 하는 이유: 비정상 종료를 하는 프로그램이기에 그걸 보려고 하는데…
                // 밑에 있는 비정상 종료의 비트를 보면 마지막 비트가 core dump 이기 때문에 7 비트만 읽음
                // 7bit 를 16 진수로 봤을 때 다 최대값이 0x7f 임 그걸 엔드연산함, 결과값이 6 이 나옴.
                // kill -l 명령어를 통해 시그널이 어떤 녀석이 들어왔는지 확인 가능하다.
                printf("status: 0x%x\n",status & 0x7f);
                printf("status: %d\n",status & 0x7f);
        }
        else if(pid == 0)
                // 강제로 종료를 해주는 녀석 (시그널 중 하나이다.)
                abort();
        else
        {
                perror("fork()");
                exit(-1);
        }
        return 0;
}
// stty -a   → 사용되는 시그널의 값들을 전부 보여준다.
// kill -l    →  리눅스 상에 존재하는 시그널을 볼 수 있다.
```

<근데 왜 이게 16bit 인지 모르겠음>

정 상 종 료

(255 위부터 공간을 쓴다)



비 정 상 종 료

(2^7 지점에 core dump 가 있다)