TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/3/23
수업일수	22 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

1. ls 구현

1. ls 옵션

(1) ls -a : 숨김 파일을 포함하여 디렉토리의 모든 항목 표시 (. 으로 시작한다.)
(2) ls -d : 디렉토리 정보만 표시
(3) ls -F : 파일이 디렉토리인 경우 ‘ / ‘, 파일이 실행 가능한 경우 ‘ * ’, 파일이 소켓인 경우 ‘ = ‘, 파일이 선입선출법인 경우 ‘ | ‘, 기호 링크인 경우 ‘ @ ’를 각 파일 이름의 뒤에 추가한다.
(4) ls -m : 쉼표로 구분
(5) ls -r : 역순으로 정렬
(6) ls -R : 하위 서브 디렉토리의 내용도 순차적으로 표시
(7) ls -s : KB 단위로 나타냄
(8) ls -t : 최종 수정시간을 기준으로 나타낸다.
(9) ls -u : 최종 수정시간 대신 최종 액세스 시간
(10) ls -l : 내부에 존재하는 파일과 디렉토리에 대한 상세한 정보를 나타낸다. 파일의 속성&권한 / 링크 / 그룹/파일의 크기 / 파일 생성시간과 날짜 / 파일명 순으로 목록이 보여진다.
(11) : ls -al : ls -a 와 ls -l 의 옵션을 섞은 것.

2. ls -R 구현

재귀를 이용한 ls-R 구현

~소스코드

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

```
void recursive_dir(char *dname); //함수선언을 미리 하고 main 함수 아래에 함수 내용을 서술하는 방식

int main(int argc, char **argv)
{
	recursive_dir("."); //현재 디렉토리의 하위 디렉토리와 파일들을 볼 것 이다.
	return 0;
}

void recursive_dir(char *dname)
{
	struct dirent *p;
	struct stat buf;
	DIR *dp;
	chdir(dname);//경로 디렉토리 위치를 바꾸는 함수
	dp=opendir(".");//현재 디렉토리에 대한 포인터를 얻음
	printf("₩t%s : ₩n",dname);
	while(p=readdir(dp))//리스트가 다 순회할 때 까지 리스트를 출력함
		printf("%s₩n",p->d_name);
	rewinddir(dp);//되감기,포인터를 맨 앞으로 가져다 놓음

	while(p=readdir(dp))
	{
		stat(p->d_name,&buf);//파일 종류를 buf 에 저장
		if(S_ISDIR(buf.st_mode))
			if(strcmp(p->d_name,".") && strcmp(p->d_name,".."))//자기 자신 디렉토리를 만나면 들어가지 않는다.("."과 ".."을 제낀다)
				recursive_dir(p->d_name);//다른 디렉토리로 들어가고 읽을게 없으면 넘어간다.
	}

	chdir("..");//끝나면 이전 디렉토리로 돌아감
	closedir(dp);
}
```

```
~결과
xeno@xeno-NH:~/proj/0323$ ./a.out
           . :
a.out
1.c
.
..
```

2.fork()

1.fork 기본 활용법

자기가 실행된 시점 이후의 밑의 라인부터 복사한다.

기존의 프로세스와 생성된 프로세스 중 누가 먼저 실행 될 지는 모름. 보편적으로 자식이 먼저 실행된다.

```
~소스코드

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

int main(void)
{

printf("before₩n");
fork(); //밑의 소스코드부터 복제한다
printf("after₩n");

return 0;
}
```

~결과
before // 복사 이전
after
after

2.fork 프로세스 실행 순서

~소스코드

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

int main(void)
{

	pid_t pid;//pid_t : int 랑 같은거
	pid=fork(); // 밑의 라인들을 복제, 자식의 pid 값을 반환
	if(pid>0)
		printf("parent₩n");
	else if(pid==0)
		printf("child₩n");
	else
	{
		perror("fork()");//에러가 나면 출력, 어떤 문제가 생기는지 에러메세지를 출력해주는 함수
		exit(-1);
	}
	return 0;
}
```

~결과
parent
child //어떤 사람은 child 가 먼저 나올 수 있음

3. pid

~소스코드

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

int main(void)
{

        pid_t pid;
        pid=fork(); //자식 프로세스 생성
        if(pid>0) // 부모 프로세스이면
                printf("parent : pid = %d, cpid = %d₩n",getpid(),pid); //자기자신의 pid 와 자식의 pid 를 얻음
        else if(pid==0) // 자식 프로세스이면
                printf("child : pid = %d, cpid = %d₩n",getpid(),pid);
                        //자기자신의 pid 와 자식의 pid 를 얻음
                        //자식이 없어서 0이 나옴
                        //fork 하는 순간 자식의 pid 가 생성된다.
                        // 값은 0이 나올수 도 있고 쓰레기 값이 들어갈 수 도 있다.
        else
{
                perror("fork()");
        exit(-1);
```

```
}
return 0;
}
```

~결과

```
parent : pid = 4266, cpid = 4267
child : pid = 4267, cpid = 0

//parent 의 cpid 값과 child 의 pid 값이 같음
```

4.fork 와 멀티태스킹

~소스코드

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

int main(void)
{
        pid_t pid;
        int i;
        pid=fork();
        if(pid>0) //parent 이면
        {
                while(1) // 무한 반복한다
                {
                        for(i=0;i<26;i++)// A~Z 까지 출력한다.
                        {
                                printf("%c",i+'A');
```

```
                                        fflush(stdout);
                                }
                        }
                }
        else if(pid==0) // child 이면
        {
                while(1) // 무한반복한다.
                {
                        for(i=0;i<26;i++) // a~z 까지 출력한다.
                        {
                                printf("%c",i+'a');
                                fflush(stdout);
                        }
                }
        }
        else //parent 나 child 가 아닌 경우는 에러로 -1을 반환한다.
        {
                perror("fork()"); //fork()에 대한 에러를 출력한다.
                exit(-1);

        }
        printf("₩n");
        return 0;
}
```

~결과

대문자가 나오면 parent 가 실행되는 것 이고, 소문자가 나오면 child 가 실행되는 것 이다.
각 프로세스는 대문자와 소문자를 무한 반복하도록 코드가 짜여져 있지만 실행 결과를 보면 대문자와 소문자가 나오기를 반복하는데, 이는 CPU 가 한번에 한 작업밖에 할 수 없는데 multitasking 을 하려면 프로세스들을 순차적으로 빠르게 실행시키는 것이다. 각 프로세스마다 할당된 시간이 있고 프로세스가 실행될 때 Run Queue 에 위치해 있다가 프로세스를 끝마치지 못한다면 Wait Queue 에 위치한다. 그리고 다음 프로세스가 Run Queue 에 올라가 실행하게 되고 이 프로세스도 할당된 시간 내에 끝내지 못하면 Wait Queue 로 올라간다. 이후 Wait Queue 에 위치했던 이전 프로세스가 다시

Run Queue 에 올라가 실행하는 것을 반복한다. Wait Queue 에서 Run Queue 로 올라갈 때 (대소문자가 변경 될 때 마다) context switching 이 이루어져 이전에 미쳐 실행되지 못하고 끝난 이후의 작업을 실행하는 것이다. 이를 매우 빠르게 처리하므로 Multitasking 이 가능 한 것 처럼 보인다.

5.각 프로세스간 데이터 공유 1

#include<stdio.h>
 	
int main(void)
{
 	
int a=10;
printf("&a=%#p₩n",&a); //a 의 주소를 출력
sleep(1000); // 1초 지연시킨다.
return 0;
}
	#include<stdio.h>
	
	int main(void)
	{
	
	int *p=0x7ffcb0a69434; //왼쪽에서 출력된 a 의 주소를 p 에 대입
	printf("&a : %#p₩n",*p); //a 의 주소 출력
	return 0;
	}

gcc -o test 6.c xeno@xeno-NH:~/proj/0323$ ./text & int *p=0x7ffcb0a69434;	xeno@xeno-NH:~/proj/0323$ ./a.out Segmentation fault (core dumped) 각 프로세스는 데이터를 공유할 수 없다

프로세스마다 권한이 별도 권한이있는데 오른쪽 소스에서는 왼쪽의 프로세스에 접근하려 하여 segmentation fault 가 뜨게 된다. 작업을 효율적으로 분담하기 위해 프로세스를 분리하는데, 각 프로세스에게 정보전달을 할 수 없음 그래서 pipe 나 message queue, shared memory(실제 물리메모리를 공유)를 이용한다.

6. is process shared VM?

1) PIC 가 필요한 이유 - PIC : 우선순위 인터럽트 제어기. 작업을 분담할 때 정보를 전달하는 매커니즘
2) 프로세스끼리는 가상메모리를 공유할 수 없음 -
3) 프로세스를 왜 분할하는가. - 메모리에 데이터를 전부 올릴 수 없기 때문에

7. 전역변수와 프로세스

~소스코드

```
#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)//parent
        {
                printf("global : %d ,local : %d₩n",global,local);// 100, 10
        }
        else if (pid==0)//child
        {
                global ++;
```

```
                local++;
                printf("global : %d, local : %d₩n",global, local); //101 , 11
        }
        Else //parent 도 child 도 아니면
        {
                perror(fork()); //fork()에 대한 에러의 설명을 출력
                exit(-1); //-1을 반환한다
        }
        printf("₩n");
        return 0;
}
```

~결과
global : 100 ,local : 10 //parent 프로세스

global : 101, local : 11 //child 프로세스

글로벌 값이 갱신 되지 않는 이유는 프로세스가 달라서이다.
C.O.W : Copy On Write (메모리에)쓰기를 사용할 때 복사가 이루어지게 된다.
당장은 text 가 필요해서 text 를 복사하고, 그 다음은 stack 영역을, 다음은 data 를 복사한다. 메모리에 쓰기 작업이 발생할 때 복사한다.

8. 프로세스간 파이프 통신

~소스코드

```
#include<unistd.h>
#include<stdio.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); //myfifo 라는 파일을 만든다
	if((pid=fork())>0) //parent 일 때
	{
		for(;;)
		{
			ret = read(0,buf, sizeof(buf)); // 키보드로부터 받은 문자열을 buf 에 저장한다
			buf[ret]=0;
			printf("Keyboard : %s₩n",buf); // 키보드로부터 받은 문자열 출력
		}
	}
	else if(pid==0) // child 일 때
	{
		for(;;)
		{
			ret=read(fd,buf,sizeof(buf)); // myfifo 에 저장된 문자열을 읽어 buf 에 저장
			buf[ret]=0;
			printf("myfifo : %s₩n",buf); // buf 를 출력한다
		}
	}
	else
	{
		perror("fork()");
		exit(-1);
	}
```

```
        close(fd);
        return 0;
}
```

~결과

```
xeno@xeno-NH:~/proj/0323$ mkfifo myfifo
xeno@xeno-NH:~/proj/0323$ ls
1.c  2.c  3.c  4.c  5.c  6-1.c  6.c  7.c  8.c  a.out  myfifo  test
xeno@xeno-NH:~/proj/0323$ ./a.out
dafdfdaf
Keyboard : dafdfdaf

myfifo : admgaglag

myfifo : asdf

myfifo : adf

myfifo : dsf

myfifo :

myfifo : sd


xeno@xeno-NH:~/proj/0323$ cat > myfifo
admgaglag
asdf
adf
dsf

sd
```

프로세스 분할이 되어 논블록킹으로 권한설정을 하지 않아도 된다. 매우 빠르게 멀티태스킹이 되어서 그렇다.

9.좀비 프로세스

```
#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) // parent 프로세스
		sleep(1000); // 1초 지연한다
	else if(pid==0) //child 프로세스
		;  //아무것도 안하고 리턴되어서 자식 프로세스가 죽음
	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}
```

새 터미널을 열고 ps -ef | grep a.out 명령어 입력

```
xeno@xeno-NH:~/proj/0323$ ps -ef | grep a.out
xeno        2469   2217   0 15:42 pts/2      00:00:00 ./a.out
xeno        2470   2469   0 15:42 pts/2      00:00:00 [a.out] <defunct>
xeno        2494   2471   0 15:43 pts/18     00:00:00 grep --color=auto a.out
```

<defunct>라고 나오는 것이 좀비 프로세스이다.

Child 가 아무것도 하지 않고 return 0되어 프로세스가 죽게 되었는데 이를 처리해 줄 parent 프로세스가 지연 중이어서 좀비 프로세스가 생성되었다.

10.wait

~소스코드

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

int main(void)
{

        pid_t pid;
        int status;
        if((pid=fork())>0) //parent 이면
        {
                wait(&status); // 자식 프로세스가 종료 될 때 까지 아무 일도 하지않고 기다린다
                printf("status : %d₩n",status);
        }
```

```
        else if(pid==0) //child 이면
        {
                exit(7);
        }
        else
        {
                perror("fork()");
                exit(-1);
        }
        return 0;
}
```

~결과
status : 1792

1792/256=7 → exit (7)

status 에는 자식 프로세스가 종료될 때 상태정보가 저장된다. 프로세스가 정상적으로 종료되었다면 하위 8비트에는 0이 저장되고 상위 8비트에는 프로세스가 종료되게 한 exit 함수의 인수가 기록된다. 하지만 비정상적으로 종료된다면 status 의 하위 8비트에는 프로세스를 종료한 시그널의 번호가 저장되며 상위 8비트에 0이 저장된다.

위의 코드는 정상 종료되어 반환된 7의 값을 보기 위한 코드이다

8bit shift 하면 256나눌 필요가 없다.
기본적으로 프로세스는 시그널을 맞으면 죽는다. 시그널이 비정상 종료에 사용됨.

11. 프로세스 정상종료/비정상 종료

~소스코드 : 정상종료

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

int main(void)
{
        pid_t pid;
        int status;
        if((pid=fork())>0)
        {
                wait(&status);
                printf("status :0x%x₩n",(status >> 8) & 0xff);
        //status 를 오른쪽으로 8번 쉬프트하고 0xff 를 &시켜 7자리수
까지만 나오도록 한다. 첫 비트는 다른 용도가 있을 것이다.
        }
        else if(pid==0)
        {
                exit(7); //정상종료로 7을 반환한다
        }
        else
        {
                perror("fork()");
                exit(-1);
        }
        return 0;
}
```

~소스코드 : 비정상 종료

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

int main(void)
{
        pid_t pid;
        int status;
        if((pid=fork())>0)
        {
                wait(&status);
                printf("status :0x%x₩n",WEXITSTATUS(status));
        //16진수로 status 값을 출력
        }
        else if(pid==0)
        {
                abort(); //비정상 종료로 0을 반환한다
        }
        else
        {
                perror("fork()");
                exit(-1);
        }
        return 0;
}
```

~결과 status :0x7	~결과 status :0x0

12. 시그널과 비정상 종료 반환값

~소스코드

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

int main(void)
{

        pid_t pid;
        int status;
        if((pid=fork())>0)
        {
                wait(&status);
                printf("status :0x%x₩n",status); // status 값을  16진수로  출력한다.
        }

        else if(pid==0)
        {
                abort(); //비정상  종료로  0을  반환한다.
        }
```

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

~결과
status :0x86
~결과 : 시그널 값 제외
status :0x6

kill -l 하면 리눅스 상의 시그널들을 볼 수 있는데 위의 결과 값을 보면 시그널을 받아 값이 크게 나오게 되었다 시그널 값을 제외한 값을 보기 위해 128을 빼 결과를 보면 0x6의 결과가 나오는 것을 확인할 수 있다.

3. 오답노트

[임베디드 애플리케이션 분석 - 2 번]

2. 배열에 아래와 같은 정보들이 들어있다.

2400, 2400, 2400, 2400, 2400, 2400, 2400,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
2400, 2400, 2400, 2400, 2400, 2400, 1, 2, 3, 4,
5, 1, 2, 3, 4, 5, 2400, 2400, 2400, 2400, 2400, 5000,
1, 2, 3, 4, 5, 5000, 5000, 500, 500, 500, 500, 500,
1, 2, 3, 4, 5, 500, 500, 500, 500, 500, 500, 500,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 234, 345, 26023, 346, 345, 234,
457, 3, 1224, 34, 646, 732, 5, 4467, 45, 623, 4, 356, 45, 6, 123,
3245, 6567, 234, 567, 6789, 123, 2334, 345, 4576, 678, 789, 1000,
2400, 2400, 2400, 2400, 2400, 2400, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5,
2400, 2400, 2400, 2400, 978, 456, 234756, 5000, 5000, 5000, 2400, 500, 5000, 2400, 500,
500, 500, 500, 500, 500, 1, 2, 3, 4, 5, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 1, 2, 3, 4, 5, 500, 500, 500, 5000, 2400, 5000,
5000, 5000, 5000, 5000, 5000, 5000, 5000, 1, 2, 3, 4, 5, 5000, 5000,
5000, 5000, 2400 5000, 500, 2400, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000,
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 5000, 5000, 5000, 5000, 5000,
1, 2, 3, 4, 5, 5000, 5000, 5000, 5000, 5000, 234, 4564, 3243, 876,
645, 534, 423, 312, 756, 235, 75678, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000,
500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400,
5000, 500, 2400, 5000, 500, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8,
9, 6, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000,
500, 2400, 5000,

여기서 가장 빈도수가 높은 3 개의 숫자를 찾아 출력하시오!
함수에서 이 작업을 한 번에 찾을 수 있도록 하시오.
(찾는 작업을 여러번 분할하지 말란 뜻임)

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

typedef struct __bt{
	int data;
	int count;
	struct __bt *left;
	struct __bt *right;
}bt;

bt* get_node(void)
{
	bt *tmp;
	tmp=(bt*)malloc(sizeof(bt));
	tmp->left=NULL;
	tmp->right=NULL;
	tmp->count=0;
}
void ins_bt(bt **mid, int data)
{
	bt *tmp=*mid;

	if(*mid==NULL)
	{
		*mid=get_node();
		(*mid)->data=data;
		printf("%d₩n",(*mid)->data);
		return ;
	}
	else if (tmp->data == data)
	{
		tmp->count+=1;
```

```
                printf("count : %d₩tdata : %d₩n",tmp->count,tmp->data);
                return ;
        }
        else if (tmp->data >data)
        {
                ins_bt(&tmp->left,data);
        }
       else if (tmp->data <data)
        {
                ins_bt(&tmp->right,data);
        }

}

int main(void)
{
        int i,size;
        int arr[]={2400, 2400, 2400, 2400, 2400, 2400, 2400,
    1,      2,     3,      4,          5,      1,        2,     3,      4,          5,
    2400, 2400,    2400,     2400, 2400,     2400, 1,        2,     3,      4,
    5,      1,        2,     3,      4,          5,      2400, 2400, 2400, 2400, 2400, 5000,
    1,        2,     3,      4,          5,      5000, 5000, 500,    500,    500,    500,    500,
    1,        2,     3,      4,          5,      500,      500,    500,      500,    500,    500,    500,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      1,        2,     3,      4,          5,
    1,        2,     3,      4,          5,      1,        2,     3,      4,          5,      6,      7,      8,      9,      10,
    11,      12,      13,     14,       15,     16,     17,     18,     234,    345,    26023, 346,      345,    234,
    457,    3,      1224, 34,     646,    732,    5,        4467,      45,     623,      4,      356,    45,     6,      123,
```

```
3245, 6567,   234,    567,   6789,    123,   2334, 345,   4576, 678,   789,   1000,
2400, 2400,    2400, 2400, 2400, 2400, 1,       2,    3,     4,         5,     1,       2,    3,     4,         5,
2400, 2400, 2400,   2400,    978,    456,   234756, 5000,    5000, 5000,    2400, 500, 5000, 2400, 500,
500,   500,    500,    500,   500,     1,       2,    3,     4,         5,     500,   500,   500,   500,   500,
500,   500, 500, 500, 500, 1,       2,    3,     4,         5,     500, 500, 500, 5000, 2400, 5000,
5000, 5000, 5000, 5000, 5000, 5000,   5000,     1,       2,    3,     4,         5,     5000, 5000,
5000, 5000, 2400 5000, 500, 2400, 5000, 5000, 5000, 5000, 5000,   5000,    5000, 5000,
1,       2,    3,     4,         5,     1,       2,    3,     4,         5,     5000, 5000, 5000, 5000, 5000,
1,       2,    3,     4,         5,     5000, 5000,   5000,    5000, 5000,    234,   4564, 3243, 876,
645,   534,   423,    312,    756,   235,     75678, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000,
500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400, 5000, 500, 2400,
5000, 500, 2400, 5000, 500, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8, 9, 6, 7, 8,
9, 6,   500, 2400, 5000,   500, 2400, 5000,   500, 2400, 5000,   500, 2400, 5000,   500, 2400, 5000,
500, 2400, 5000};
   bt *mid=NULL;
   bt *cou=NULL;
   size=sizeof(arr)/sizeof(int);
   for (i=0;i<size;i++)
   {
         ins_bt(&mid,arr[i]);
   }

   return 0;
```

[임베디드 애플리케이션 분석 - 9 번]
함수 포인터를 반환하고 함수 포인터를 인자로 취하는 함수의 주소를 반환하고
    인자로 int 2 개를 취하는 함수를 작성하도록 한다.
    (프로토타입이 각개 다를 수 있으므로 프로토타입을 주석으로 기술하도록 한다)

```
#include<stdio.h>

void aaa(void)
{
        printf("aaa!₩n");
}
void bbb(void)
{
        printf("bbb₩n");
}
void (*ccc(void *bbb(void)))(void)
{
        bbb();
        return aaa;
}
void (*(*(* ddd(int a,int b))(void))(void))(void) //이부분을 구현 못하겠습니다 ㅠ
{
        printf("%d",a+b);
        return ccc;
}
int main(void)
{
ddd(1,2);
return 0;
}
```

[임베디드 애플리케이션 분석 - 12번]
21. 함수 포인터를 활용하여 float 형 3 by 3 행렬의 덧셈과
int 형 3 by 3 행렬의 덧셈을 구현하도록 하시오.

```
#include<stdio.h>

void ffunc(float brr[3][3])
{
	int i,j;

	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{

			printf("%f₩t",brr[i][j]+brr[i][j]);
		}

		printf("₩n");
	}
}
void ifunc(int arr[3][3])
{
	int i,j;

	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
			printf("%d₩t",arr[i][j]+arr[i][j]);
		}
		printf("₩n");
```

```
        }

}
int main(void)
{
        int arr[3][3]={{1,2,3},{3,2,1},{1,1,1}};
        float brr[3][3]={{1.1,1.2,1.3},{3.1,2.2,4.1},{3.1,2.1,0.1}};

        ifunc(arr);
        ffunc(brr);

        return 0;
}
```