TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
22일차 (2018. 03. 23)

목차

학습 내용 복습

- fork()
  >wait()
  > How to extract status?
- aboat()
- pid_t
- fflush()
- Is process shared VM?
- C.O.W
- 학습 예제
- ls -alR 구현 해보기

fork()

fork함수가 실행된 시점 이후의 라인부터 복사한다. 복사된 프로세스를 자식프로세스라고 하고, 자식 프로세스는 fork() 가 있는 줄부터 실행이 된다. 부모는 fork() 실행함으로써 자식 프로세스의 pid 값을 반환 받고, 자식 프로세스는 fork()를 실행 함으로써 0을 반환 받는다. 자식 프로세스는 생성 즉시 곧바로 프로세스를 생성할 수 없기 때문이며, pid는 고유값이기 때문이다. 이렇게 fork 에 의해 프로세스가 2개가 되고(부모, 자식) 각각의 프로세스는 독립적으로 수행이 된다. 부모, 자식 프로세스의 실행 순서는 프로세스 스케줄링에 따라 다르다.

이때, 부모프로세스는 자식 프로세스의 상태에 대해 받는 wait(&status)가 있다. 자식 프로세스가 종료되면, 반드시 부모 프로세스가 처리해 주어야 한다. 자식의 시신은 부모가 처리해 주어야 하기 때문이다. 만약 자식 프로세스가 종료되는 순간 부모 프로세스가 sleep() 등에 의해 처리해 주지 못한다면 자식 프로세스는 **좀비 프로세스**(자기를 수습해달라고 돌아다니는 상태가 되기 때문에 붙은 이름이다)가 된다. SIGCHLD는 죽은 자식 프로세스가 신호를 보내는 것으로 죽은 상태를 알리기 위해 메모리에 남아있기 때문에 메모리 누수가 발생된다. 이 후, 부모 프로세스가 깨면 처리해주지만, 부득이하게 부모 프로세스도 죽은 상태이면 한 단계 위의 부모(조부모) 프로세스가 처리해준다.

> wait()

시스템 콜로 자식 프로세스의 상태를 확인하는 것이다. 자식 프로세스가 종료될 때까지 기다리고 어떻게 종료되었는지 확인하는 시스템 콜이다. ()안에 status가 있으면 그 곳으로 자식 프로세스가 종료될 때 상태정보가 저장된다. 자식 프로세스가 정상 종료(return, exit() 등) 가 되면 전체 16bit 레지스터의 상위 8bit에 값을 종료된 값을 반환하고, 비정상적으로 종료가 되면 하위 8bit에 값을 반환한다.

> How to extract status?

return 이나 exit는 정상종료이다. 반면, 프로세스는 시그널을 맞으면 죽게 된다. 이것은 비정상종료이다. 함수 시그널의 종류가 여러 가지가 있다.

aboat()

시스템 콜로 프로세스에 시그널을 주는 함수이다. 즉, 프로세스를 죽인다.

pid_t

프로세스 아이디(pid)를 저장하는 타입이라는 의미이다. int로 변수 선언한 것과 같은 작용을 한다. 시스템마다 프로세스번호가 int일 수도 있고 아닐 수도 있기 때문에 pid_t를 사용하는 것이 이식성면에서 더 나은 코드가 된다고 한다.

fflush()

버퍼를 비우는 함수이다. 표준출력, 표준입력, 표준에러는 그냥 입출력하는 것이 아니라 각자의 버퍼가 존재하고, 그 버퍼를 사이에 끼워서 입출력을 한다. 그리고 적정한 때에 데이터를 이동시키고 버퍼를 비운다. 표준입력버퍼는 개행문자가 들어오면 데이터를 이동시키고 버퍼를 비운다고 한다. 그리고 fflush() 는 버퍼를 비우는 시점을 프로그래머가 정할 수 있게 해주는 c언어 표준함수이다.

OS마다 버퍼를 비우는 방식이 다르다. 버퍼를 비우는 시점은 운영체제가 결정하는데 윈도우 같은 경우에는 표준출력/에러 버퍼에 데이터가 들어오자 마자 바로 비우고 데이터를 내보내지만 리눅스는 개행문자('₩n')가 들어와야지 버퍼를 비우고 출력을 한다고 한다. (그래서 리눅스에서 개행문자로 끝나지 않는 문자열을 바로 출력하려면 fflush(stdout); 를 출력하는 부분 바로 아래에 놔둬야 한다.??)

Is process shared VM(가상 메모리)?

1) IPC 가 필요한 이유

IPC 프로세스 간 통신(interprocess communication)의 약어이다. 프로세스 사이에 데이터를 주고받는 일을 말한다. **작업을 분담할 때, 정보를 전달하는 매커니즘이다.** 프로세스 간 통신은 실제로 한 컴퓨터 내의 프로세스 간보다도 네트워크 상의 객체 사이에 데이터를 주고받을 수 있도록 설계된 것이 많다. 예를 들면, 4.3BSD의 경우 프로세스 간 통신은 프로세서가 아니라 소켓 사이에 이루어진다. 이때 상대방 소켓은 네트워크 상의 어떤 컴퓨터에 있어도 된다. 프로세스 간 통신의 구체적인 실현 수법으로는 세마포(semaphore), 신호, 파이프 등이 쓰인다.

2) 프로세스끼리는 가상메모리를 공유할 수 없다. 각각의 프로세스는 다른 프로세스의 페이징에 대한 권한이 없다.

3) 프로세스를 왜 분할 하는가? 메모리에 데이터를 전부 올릴 수 없기 때문이다.

C.O.W

Copy On Write 의 약자로, 메모리에 쓰기를 사용할 때 복사가 이루어지게 된다. text가 먼저 필요해서 text를 복사하고(기계어를 돌기 위해서 먼저 복사해야 한다), 그 다음은 stack 영역을, 다음은 data를 복사한다. 즉, **메모리 쓰기 작업할 때 복사한다.**

학습예제

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

void recursive_dir(char *dname);  //  함수 포인터가 아니고 뒤에 함수를 써주기 위해 함수 선언을 한 것(프로토 타입 작성)이다. 위에 선언만 되어있어도 컴파일러가 밑의 함수를 찾아서 알아서 읽는다. 컴파일은 순차적으로 진행되기 때문에 선언문 없이 뒤에 함수를 적으면 문법 오류가 생긴다. 규모가 커질 때 유용하다.

int main(int argc, char *argv[])          //밑 코드에서 인자를 "."로 받아서 main 인자를 쓸 필요가 없다.
{     recursive_dir(".");
      return 0;                       }

void recursive_dir(char *dname)           //메인 코드에 따라 "."넘겨준다.  파일은 디렉토리도 포함되기 때문
{   struct dirent *p;                     //디렉트로 포인터에 있는 리스트들을 선언.
   struct stat buf;
   DIR *dp;
   chdir(dname);                          //디렉토리 포인터를 현재 위치로 바꾸어준다.
   dp = opendir(".");
   printf("₩t%s : ₩n", dname);           //'dname = .' 즉, 디렉토리 이름을 출력한다.
```

```
  while(p= readdir(dp))
    printf("%s₩n", p->d_name);          // 이안에 리스트가 다 순회할 때까지 하위 파일이름을 출력한다.
  rewinddir(dp);                         //되감기 하는 것! dp를 다시 읽을 수 있도록 포인터를 앞으로 옮김.
  while(p=readdir(dp))
  {     stat(p->d_name, &buf);
          if(S_ISDIR(buf.st_mode))          //디렉토리 판별
      if(strcmp(p->d_name, ".")&& strcmp(p→d_name,".."))      //숏컷. 앞이 0이면 값이 0이니까 pass된다.
          recursive_dir(p->d_name);       // 다른 디렉토리로 들어가게 된다.
  }// 현재 디렉토리는 다 읽었으나 만족을 못해서 밑의 코드로 상위로 올려버려서 만족시키는 것이다
   chdir("..");               // 상위로 올라간다.
   closedir(dp);          }          //오픈 한 것 닫아준다.
```

결과
디렉토리에 있는 숨김 파일을 포함하여 파일 리스트가 출력된다.

Fork() 관련 학습 예제

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{    printf("before₩n");
    fork();
    printf("after₩n");
    return 0;               }
```

결과
before
after
after   // fork()의 밑으로 복사한 프로세스가 1개 더 생성 되기 때문에 after가 한 번 더 출력 된 것이다.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main (void)
{   pid_t pid;
  pid = fork();           // fork는 자식의 pid값을 반환한다. pid에 그 값을 저장한 것이다.
  if(pid>0)               // 부모 프로세스는 자식의 pid값이 있으니까 참이다.
   printf("parent₩n");
  else if(pid==0)         //자식 프로세스는 바로 다른 프로세스를 생성하지 못하므로 pid값이 0이다.
   printf("child₩n");
  else                    //fork가 제대로 실행되었다면 나오지 않으므로 포크에서 뭐가 잘못 되었는지 확인
```

```
  {    perror("fork()");    //어떤 문제가 있는지 에러 메세지 출력해준다.
    exit(-1);         }       // 정상적 종료가 아니므로 -1값을 갖는다.
   return 0;                  }
```

결과

parent

child

어떤 프로세스가 먼저 실행되는가를 알아보는 예제였다. 기존 프로세스와 생성된 프로세스는 누가 먼저 시작될 지는 모른다. 강의실을 parent가 먼저 실행되는 쪽이 많았다.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main (void)
{    pid_t pid;   //기계어 레벨에서 생각할 것! 선언하기만 했으니까 메모리에 할당된 상태는 아니다.
    pid = fork(); // 이렇게 대입할 때는 쓰레기 값이 있어도 상관이 없다. 연산에 직접적으로 관련이 없는
                  상태이므로 상관이 없다. 프로세스를 생성하는 것이다.
    if(pid >0)    //getpid는 자기 자신의 값(pid)을 가져온다. 기본적으로 리턴 값은 자식 프로세스 아이디다.
      printf("parent : pid = %d, cpid = %d₩n", getpid(), pid);
    else if(pid==0)       //자식이 없으니까 pid == 0이 나오는 것. 프로세스는 2개이기 때문
      printf("child : pid = %d, cpid = %d₩n", getpid(), pid);
    else
  {    perror("fork()");
          exit(-1);           }
return 0;                           }
```

결과

parent : pid = 5102, **cpid = 5103**

child : pid = 5103, cpid = 0

위의 cpid와 chlil의 pid 값이 같은지 알아보기 위해 작성된 예제이다. fork의 리턴 값이 자식 프로세스의 pid이므로 같아야 한다.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main (void)
{        pid_t pid;
         int i;
         pid = fork();
         if(pid>0)          //부모 프로세스
        {      while(1)     //무한 반복한다.
                {       for(i=0; i<26; i++)
```

```
                {       printf("%c", i+'A');              // 대문자 출력
                        fflush(stdout);                   //출력하는 buf를 비우라는 것
                }
        }
    }
    else if(pid == 0) //자식 프로세스
    {       while(1)
        {       for(i=0; i<26; i++)
                {       printf("%c", i+'a');              //소문자 출력
                        fflush(stdout);
                }
        }
    }
    else
    {       perror("fork()");
            exit(-1);           }
    printf("₩n");
    return 0;                   }
```

결과

대문자와 소문자가 섞여서 나온다. 원래 각 프로세스는 대문자와 소문자를 무한 반복하도록 코드가 짜여져 있지만 실행 결과를 보면 대문자와 소문자가 나오기를 반복한다. 이는 프로세스 A, B가 같이 돌고 있다는 것을 뜻한다. 프로세스가 동시다발적으로 돌아간다는 것을 확인하는 예제이다. CPU는 한번에 한 작업밖에 할 수 없는데 multitasking 을 하려면 프로세스들을 순차적으로 빠르게 실행시키는 것이다. 각 프로세스마다 할당된 시간이 있기 때문이다. 대문자가 나오고 소문자가 나오게 될 때 context switching이 돌아가는 것이다. 엄청 빠르게 돌아가는데 이는 cpu가 그만큼 빠르기 때문이다. 때문에 multi tasking으로 보이는 것이다.

```
#include <stdio.h>
int main (void)
{    int a =10;
    printf("&a = %#p₩n", &a);  // a의 주소값을 출력한다.
    sleep(1000);                // 프로세스를 지연시킨다.
    return 0;     }
```

결과 &a = 0x7ffdc53f5354

7로 숫자가 나오는데 유저 영역인 것을 확인한 것이다. 경계선이 8이니까 f니까 엄청 가까이 있다. 이는 스택을 쓰고 있으니까 7로 나오는 것이다.

```
#include <stdio.h>
int main (void)
{   int *p = 0x7ffdc53f5354;
   printf("&a : %#p₩n", *p);
   return 0;      }
```

결과

Segmentation fault (core dumped)

원래 의도는 위 두개의 예제로 10이 출력 되게 하는 것이었는데 segmentation fault가 났다. 이는 프로세스가 다르기 때문이다. **프로세스마다 권한이 별도**로 있는데 아래 소스에서 위쪽의 프로세스에 접근하려 하여 segmentation fault 가 뜨게 된 것이다. **메모리 자체에도 권한**이 있다. 그 권한을 커널에 있는 mm_struct에 있다. 즉, 여러 프로세스간에는 서로 프로세스상의 가상메모리를 공유하지 못한다. 이는 fork로 이루어진 부모 프로세스와 자식 프로세스의 관계라 하더라도 안 된다. 하지만 프로세스는 작업을 분담하거나 하나의 프로세스를 특화시키기 위해서는 나누어야 한다. 이렇게 나누어 놓고 공유가 되지 못한다면 각 프로세스에 정보를 전달하지 못할 것이다. 그렇다면, 어떻게 해야할까? 그래서 도입된 것이 파이프(pipe)다. 메세지 큐(message queue)랑 쉐어드 메모리(shared memory, 물리 메모리 공유)를 이용한다.

```
#include <unistd.h>
#include <stdio.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)                    // 부모 프로세스
        {    printf("global:%d, local :%d₩n", global, local);          }
        else if(pid==0)              //자식 프로세스
        {     global++;
              local++;
              printf("global :%d, local:%d₩n", global, local);         }
        else
        {     perror("fork()");
                exit(-1);                                              }
    printf("₩n");
    return 0;     }
```

결과

```
global:100, local :10
global :101, local:11
```

글로벌 값이 갱신 되지 않는 이유는 프로세스가 달라서이다. C.O.W(Copy On Write)로 메모리에 쓰기를 사용할 때 복사가 이루어지게 된다.

```
#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);
    if(pid = fork()>0)
     {  for(;;)
        { ret = read(0, buf, sizeof(buf));  //키보드 입력
          buf[ret] = 0;
          printf("Keyboard : %s₩n", buf);         }
     }
     else if(pid == 0)
      { for(;;)
        {  ret = read(fd, buf, sizeof(buf));  //파이프 입력
           buf[ret] = 0;
           printf("myfifo : %s₩n", buf);         }
      }
     else
      { perror("fork()");
        exit(-1);                    }
close(fd);
return 0;       }
```

결과

mkfifo myfifo 로 피포 만든 후에 실행하고 새터미널 창에서 cat>myfifo하면 통신이 된다.

논블로킹 안 써도 논블로킹으로 되는 것은 다른 프로세스이기 때문이다. 즉, 프로세스가 분할되어 논블로킹으로 설정하지 않아도 되는 상태이다. 부모 프로세스와 자식 프로세스가 각기 다른 일을 할 것을 지정해주었기 때문이며, 매우 빠르게 멀티테스킹이 되기 때문에 가능한 것이다. 서로 다른 곳에서 입력 받기를 기다리고 있다.

```
#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)
       sleep(1000);
     else if(pid == 0)
       ;                              // 아무런 실행동작을 하지 않고 return 0;을 하므로 죽은 프로세스가 된다.
     else
     {  perror("fork()");
        exit(-1);                  }
    return 0;                          }
```

결과

좀비 프로세스가 생성된다. 실행시키고(아무것도 안뜨고 커서가 껌벅거리기만 한다) 새 터미널 창 쳐서 ps-ef|grep a.out 을 하면 defunct가 뜨는 것을 볼 수 있다. 이것은 죽은 프로세스이다. 부모프로세스가 깨어나면 거두어줘서 이 defunct가 사라진다.

```
#include <unistd.h>
#include <stdio.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);
       printf("status : %d₩n", status);    }
   else if(pid == 0)
      exit(7);
   else
    { perror("fork()");
      exit(-1);        }
  return 0;                  }
```

결과

status : 1792

알 수 없는 값이 뜨는데 이 때, status를 8bit shift 하면 출력이 7이된다. 이는 정상종료 되어 반환된 값을 보기 위함이었다. status 에는 자식 프로세스가 종료될 때 상태정보가 저장되기 때문이다.

```
#include <unistd.h>
#include <stdio.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);
       printf("status : 0x%x₩n", (status>> 8) & 0xff);    }
```

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

결과

status : 0x7 //& 0xff 뒤의 값만 출력하여 깔끔하게 만들기 위해 추가 된 것이다.

```
#include <unistd.h>
#include <stdio.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);
       printf("status : %d₩n", WEXITSTATUS(status));    }
   else if(pid == 0)
      exit(7);
   else
    { perror("fork()");
      exit(-1);                    }
  return 0;                              }                    //      마찬가지로 결과가 status : 7 나온다.

#include <unistd.h>
#include <stdio.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);
       printf("status : %d₩n", status);          }
```

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

결과

status : 134

여기서 WEXITSTATUS를 붙여주었으면 출력이 0으로 나온다

status에 -128을 하면 6이 나온다. 왜 6이 나올까?

```
sue100012@sue100012-Z20NH-AS51B5U:~/project/2_23$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

abort()는 강제 종료로 비정상 종료이다. 위는 시그널 목록을 보여주는 것으로 이 목록 값을 반환한 것이다.

이때 왜 그냥 하면 134가 나오고 -128을 해야지 시그널 번호가 나올까?

시그널을 맞아서 죽으면 죽은 메모리 값을 반환한다. 이때, 최상위 비트를 세팅할까 아닐까 여부를 최상위 비트에서 표시한다. 최상위 비트 1비트 다른 용도로 사용되어서 이걸 제외하기 위해 -128을 쓴다.

& 0x7f 를 해도 같은 값이 나온다. 최상위 비트가 세팅이 되어있을 수도 아닐 수도 있다. 그래서 & 0x7f 를 쓰는 것이 더 좋은 방법이다.

ls -alR 구현 해보기

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

```
#include <stdio.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <dirent.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
        DIR* dp;
    struct stat buf;
        struct dirent* p;
        struct passwd* pw;
        struct group* gr;
        struct tm *tm;
        char ch;
        char perm[11]="----------";
        char rwx[4]="rwx";
        char sst[4]="sst";
        int i,cmd;
    cmd = getopt(argc, argv,"a");

        stat(argv[1],&buf);
        if(S_ISDIR(buf.st_mode))//디렉토리
                perm[0]='d';
        if(S_ISREG(buf.st_mode))//레귤러(일반) : 파일
                perm[0]='-';
        if(S_ISFIFO(buf.st_mode))//파이프
                perm[0]='P';
        if(S_ISLNK(buf.st_mode))//바로가기 파일
                perm[0]='l';
         if(S_ISSOCK(buf.st_mode))
                perm[0]='s';
        if(S_ISCHR(buf.st_mode))//캐릭터 디바이스
                perm[0]='c';
        if(S_ISBLK(buf.st_mode))//블록 디바이스
                perm[0]='b';
    while (p = readdir(dp))
    {if (cmd != 'a') // a아닐때
        {if (p->d_name[0] == '.')
            continue;
            for(i=0;i<9;i++)
```

```
            {
                if((buf.st_mode>>(8-i))&1)
                    perm[i+1]=rwx[i%3];
            }
            for (i=0;i<3;i++)
            {
                if((buf.st_mode>>(11-i))&1)
                    if(perm[(i+1)*3]=='.')
                        perm[(i+1)*3]==sst[i]^0x20;
                    else
                        perm[(i+1)*3]=sst[i];
            }
        }
        }
    printf("%s₩n",perm);
        printf("%lu",buf.st_nlink);
        pw=getpwuid(buf.st_uid);
        printf("%s",pw->pw_name); //real password 이름 표시
        gr = getgrgid(buf.st_gid);
        printf("%s",gr->gr_name); //그룹이름 표시
        printf("%lu",buf.st_size); // 사이즈 표시
        tm=localtime(&buf.st_mtime);
        printf("%d-%02d-%02d %02d %02d",tm->tm_year + 1900, tm->tm_mon+1,tm->tm_mday, tm-
>tm_hour, tm->tm_min); // 시간표시
        printf("₩n");
        return 0;
}
```