

## CSE3030 어셈블리 프로그래밍 #6 문자 반복

아래 좌측과 같은 입력 파일 in.txt가 있다.

이 파일의 각 줄의 첫 글자 k는 1 ~ 9 사이의 수자이다. 다음 빙칸이 정확히 한 개 있고 이어서 임의의 문자열이 주어졌다(문자열의 크기는 20자 이내이다(CR, LF를 제외한 실제 글자수. Space, tab 등 white character 포함)).

해결해야 할 문제는 in.txt를 한 줄씩 읽어, 세번째 글자부터 시작하는 문자열의 글자를 k 번씩 반복하여 출력하는 것이다. 물론 redirection을 사용하여 파일 입출력이 가능해야 한다. 위 예에 대해 출력 파일 out.txt는 아래 우측과 같다.

입력 파일 in.txt

```
2 ABCD  
1 Hello world!  
3 Hi Hi
```

출력 파일 out.txt

```
AABBCCDD  
Hello world!  
HHHiiii HHHiiii
```

### 주의 사항

입력 파일은 이를 편집하기에 따라 아래와 같이 마지막 부분의 처리가 서로 다를 수 있는데, 이러한 파일 모두를 문제 없이 처리할 수 있도록 프로그램을 작성하여야 한다.

(i) 마지막에 빙 줄이  
    없는 경우

(ii) 마지막에 아무것도 없는  
    빙 줄이 있는 경우

(iii) 마지막에 빙칸만 있는 줄이  
    있는 경우(빈 것처럼 보임)

```
2 ABCD  
3 Hi
```

```
2 ABCD  
3 Hi
```

```
2 ABCD  
3 Hi
```

### 프로그램 제출

- 파일 이름 : snnnnnnnHW06.asm (여기서, nnnnnnn은 자신의 학번 뒤 6 자리).
- 작성한 .asm 파일을 마감 전 사이버 캠퍼스에 제출 (late 없음).
- 제출 마감 : 사이버 캠퍼스에 지정되어 있음.

### 주의 및 참고사항 (숙지하세요)

- 모든 어셈블리 instruction 및 기능을 사용할 수 있으며, 프로그램은 명령 프롬프트 뿐만 아니라 redirection을 사용하여 실행할 수 있어야 합니다.
- 채점 시 어셈블리 오류가 발생하면, 이유 불문 점수가 없습니다.
- 작성 후 다시 검토하여 코드+데이터 크기 합이 가능한 작도록 프로그램을 개선 합시다.
- 초반부에 작성자, 기능, 입력, 출력 등을 comment로 기록하여야 하며, 중간 중간에 이해를 위한 주석 달기, 빙 줄 삽입, 들여쓰기 등을 하여 보기 좋게 하여야 합니다.
- 출력 형식은 위에 보인 것과 동일하여야 합니다.
- 파일 이름 역시 위 요청 사항과 동일하여야 합니다(결과는 윈도우 함수 WriteFile을 사용하여 출력).
- 지금까지 기술한 사항에 위반되거나 미흡하면 감점하거나 또는 점수를 주지 않습니다.
- 프로그램 복사는 철저히 점검할 것입니다. 복사로 판정되면 이유불문 쌍방 0점 처리합니다.

다음 쪽에 프로그램 실행 예를 보인다.

## 프로그램 실행 예

```
D:\Work>s074419H06
```

```
2 123
```

```
112233
```

```
1 Hello
```

```
Hello
```

```
D:\Work>type inH06.txt
```

```
2 ABCD
```

```
1 Hello world!
```

```
3 Hi Hi
```

```
D:\Work>s074419H06 < inH06.txt
```

```
AABBCCDD
```

```
Hello world!
```

```
HHHiii HHHiii
```

```
D:\Work>s074419H06 < inH06.txt > outH06.txt
```

```
D:\Work>type outH06.txt
```

```
AABBCCDD
```

```
Hello world!
```

```
HHHiii HHHiii
```

```
D:\Work>
```

## CSE3030 어셈블리 프로그래밍 숙제 #6 Redirection of IO를 위한 보조 자료

Redirection이란 standard input 및 output 용도로 작성한 프로그램을 파일에서 읽고 또는 파일에 쓸 수 있도록 하는 기능을 의미한다. 예를 들어 **prog.exe**라는 파일을 다음과 같이 명령 프롬프트에서 실행하면 파일에서 입력을 읽어 파일로 출력한다(파일에서 읽고 standard output으로 출력하거나 또는 standard in에서 읽어 파일에 쓰는 것도 가능하다).

```
prog.exe < in.txt > out.txt
```

불행히도 Irvine library의 입출력 함수들은 Win32 API의 콘솔 IO를 사용하기 때문에 redirection 기능이 불가능하다. 따라서, 본 과제에서는 redirection이 가능한 Win32 API 함수를 몇 개 소개하는데 이를 이해하고 이용하여 간단한 연습 문제를 해결해 보도록 한다.

강의에서도 언급하겠지만, 교재 11 장에 API 함수에 대하여 설명하고 있는데, 본 과제를 위하여 11.1.1의 일부와 11.1.6의 일부 내용을 이해하는 것이 필요하다.

**Handle**이란 파일 포인터와 같은 의미이며, stdio를 위하여 다음과 같은 심볼이 정의되어 있다.

```
STD_INPUT_HANDLE : standard input  
STD_OUTPUT_HANDLE : standard output  
STD_ERROR_HANDLE : standard error output
```

이들을 이용하여 다음과 같이 코딩하면 **stdinHandle**과 **stdoutHandle**은 각각 **stdin**과 **stdout**에 대응하는 handle(즉, 파일 pointer)이 된다. 여기서, **INVOKE**는 매개 변수가 있는 함수를 호출하는 방법이며, **GetStdHandle**은 함수 이름, **STD\_INPUT\_HANDLE**과 **STD\_OUTPUT\_HANDLE**은 각각 함수의 인수이다.

```
.data  
    stdinHandle HANDLE ? ; ; HANDLE은 C 언어에서 FILE과  
    stdoutHandle HANDLE ? ; ; 같은 형태의 data type  
.code  
    INVOKE GetStdHandle, STD_INPUT_HANDLE ; ; 이 함수는  
    mov stdinHandle, eax ; ; eax로 handle 반환  
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE  
    mov stdoutHandle, eax
```

디스크의 파일을 open하고 close하는 방법도 11.1.6에 기술되어 있는데, 여기서는 redirection에 초점을 맞추고 있기 때문에 생략한다.

Stdio에 대응하는 handle을 얻은 후에는 다음에 보이는 함수를 사용하여 파일에 데이터를 쓰거나 또는 파일에서 데이터를 읽을 수 있다. 여기서, 데이터는 byte 단위이다.

```
ReadFile PROTO, ; ; 읽기 함수(이 함수는 EAX, EBX, ECX, EDX를 보전하지 않는다)  
handle:DWORD, ; 위에서 얻은 handle을 전달(stdinHandle)  
pBuffer:PTR BYTE, ; 입력 버퍼 offset(offset이 저장된 reg도 가능)  
nBufsize:DWORD, ; 읽고자 하는 최대 byte 수  
pBytesRead:PTR DWORD, ; 실제로 읽은 개수 저장을 위한 변수offset(위와 동일)  
pOverlapped:PTR DWORD ; 무시(0 을 전달). 의미가 있지만 생략한다.
```

```
WriteFile PROTO, ; ; 쓰기 함수(이 함수는 EAX, EBX, ECX, EDX를 보전하지 않는다)  
fileHandle:DWORD, ; 출력 handle(stdoutHandle)  
pBuffer:PTR BYTE, ; 출력 버퍼 offset((offset이 저장된 reg도 가능)  
nBufsize:DWORD, ; 쓰고자 하는 최대 byte 수  
pBytesWritten:PTR DWORD, ; 실제로 쓴 개수 저장을 위한 변수 offset  
pOverlapped:PTR DWORD ; 무시(0 을 전달)
```

예를 들어 다음과 같이 정의된 데이터가 있다고 하자. Handle 변수는 앞에서 설명한 함수 GetStdHandle을 호출하여 이미 저장되어 있다고 가정한다.

```
.data
stdinHandle HANDLE ?
stdoutHandle HANDLE ? ; BUF_SIZE는 버퍼 크기로 앞에 어딘가 정의
inBuf BYTE BUF_SIZE DUP(?) ; input buf
bytesREAD DWORD ? ; 실제로 읽은 byte 개수 저장
outBuf BYTE BUF_SIZE DUP(?) ; output buf
bytesWRITE DWORD ? ; 실제로 쓰여진 byte 개수 저장
```

함수 **ReadFile**과 **WriteFile**를 사용하여 string을 읽고 쓰는 코드의 예는 아래와 같다.

```
.code
INVOKE ReadFile,
    stdinHandle, ; input handle
    ADDR inBuf, ; 입력 버퍼 주소(INVOKE에서는 OFFSET 대신 ADDR을 사용)
    BUF_SIZE, ; 읽고자 하는 최대 크기(상수 값이나 reg에 저장해서 전달해도 무방)
    ADDR bytesREAD, ; 실제로 읽은 byte 수
    0

INVOKE WriteFile,
    stdoutHandle, ; output handle
    edx, ; 출력 버퍼 주소가 edx에 저장되어 있을 경우
    ebx, ; 쓰고자 하는 문자의 수가 ebx에 저장되어 있을 경우(상수 값도 가능)
    ADDR bytesWrite, 0 ; 인수마다 줄바꿈을 안해도 무방하다
```

입력을 keyboard로 직접 입력할 경우 함수 ReadFile은 엔터를 누를 때까지 입력한 문자를 읽어 버퍼에 저장한다. 그러나, redirection을 사용할 경우 이 함수는 주어진 최대 개수(**nBufsize**) 만큼 무조건 읽기 때문에 CR, LF로 구분된 text 파일을 redirection으로 한 줄씩 읽는 것이 매우 어렵다(사실 가능 할 것 같은데 어찌 하는지 모른다). 따라서, ~.txt 파일을 한 줄씩 읽기 위해서는 ReadFile로 한 글자씩 읽어 LF를 검출할 때까지 읽은 글자들을 버퍼에 저장할 수밖에 없다.

이렇게 파일의 현재 위치에서 줄바꿈까지 한 줄을 읽는 함수를 학생들이 작성하면 좋으나, 난이도를 줄이기 위해, 이를 위한 함수를 제공한다. **Read\_a\_Line**이라고 이름 붙인 함수의 입출력은 다음과 같다.

**입력:** EAX : File Handle을 저장(예, mov eax, stdinHandle).

EDX : 입력 버퍼의 offset을 저장(예, mov edx OFFSET inBuf).

**출력:** ECX : 읽을 글자 수(읽은 것이 없으면 0 (EOF 같은 경우))

**기능:** 파일의 현재 위치에서 CR, LF 전까지의 문자들을 읽어 EDX 가 가리키는 버퍼에 저장.

읽은 CR, LF는 무시되고, 저장된 문자열 뒤에 0을 붙인다. 그러나, ECX는 CR 이전까지 읽은 글자들의 수만 저장한다(물론 빈칸, tab 등 포함)

다음 쪽이 이 함수의 source code를 보이니 이를 이번 과제에서 사용하도록 한다. 또한, 실기 시험에도 필요할 수 있으니 꼭 이해하고 테스트해서 working code를 반드시 보관하여 재사용이 가능하도록 한다.

함수 Read\_a\_Line의 source

```
Read_a_Line PROC
    ; Input EAX : File Handle
    ;           EDX : Buffer offset to store the string
    ; Output ECX : # of chars read(0 if none(i.e. EOF)
    ; Function
    ;   Read a line from a ~.txt file until CR, LF.
    ;   CR, LF are ignored and 0 is appended at the end.
    ;   ECX only counts valid chars just before CR.

    .data
    Single_Buf__ BYTE ? ; two underscores(__)
    Byte_Read__ DWORD ? ; ""

    .code
    xor ecx, ecx ; reset counter
    Read_Loop :
        ; Note: Win32 API functions do not preserve
        ;       EAX, EBX, ECX, and EDX.
        push eax ; save registers
        push ecx
        push edx
        ; read a single char
        INVOKE ReadFile, EAX, OFFSET Single_Buf__,
                    1, OFFSET Byte_Read__, 0
        pop edx ; restore registers
        pop ecx
        pop eax
        cmp DWORD PTR Byte_Read__, 0 ; check # of chars read
        je Read_End ; if read nothing, return

        ; Each end of line consists of CR and then LF
        mov bl, Single_Buf__ ; load the char
        cmp bl, CR
        je Read_Loop ; if CR, read once more
        cmp bl, LF
        je Read_End ; End of line detected, return

        mov [edx], bl ; move the char to input buf
        inc edx ; ++1 buf pointer
        inc ecx ; ++1 char counter
        jmp Read_Loop ; go to start to read the next line

    Read_End:
        mov BYTE PTR [edx], 0 ; append 0 at the end
        ret
Read_a_Line ENDP
```