



Operating System (OS)

Lec06 : 프로세스 정보

충북대학교

강병호 (지능로봇공학과)

kang6283@chungbuk.ac.kr

02. 프로세스의 개념

■ 프로세스의 정의

■ 프로세스 / 프로세서

- 프로세스 : 실행 중인 프로그램을 의미
- 프로세서 : 인텔 코어 등과 같은 중앙 처리 장치 (CPU) 를 의미
- 프로그램 : 사용자가 컴퓨터에 작업을 시키기 위한 명령어의 집합
 - C 언어 같은 고급 언어나 셸 스크립트 같은 스크립트 언어로 작성

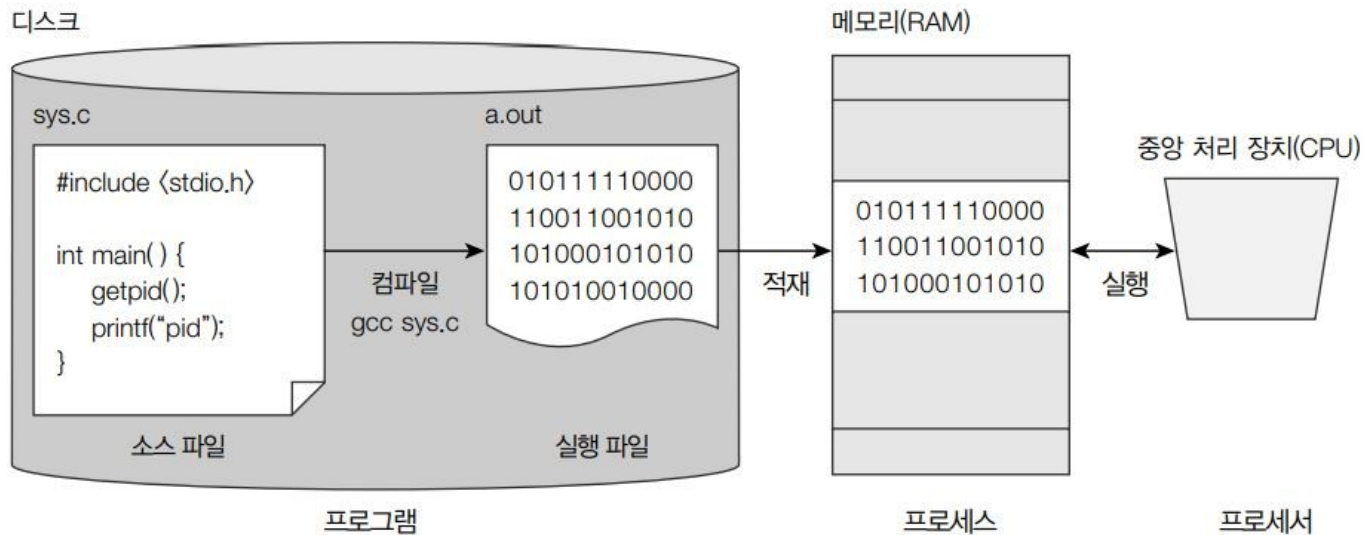


그림 6-1 프로그램과 프로세스, 프로세서의 관계

02. 프로세스의 개념

■ 프로세스의 구조

- 텍스트 영역 : 실행 코드를 저장 , 텍스트 영역은 프로세스 실행 중에 크기가 변하지 않는 고정 영역에 속함
- 데이터 영역 : 프로그램에서 정의한 전역 변수를 저장
 - 전역 변수는 프로그램을 작성할 때 크기가 고정되므로 고정 영역에 할당
- 힙 영역 : 프로그램 실행 중에 동적으로 메모리를 요청하는 경우에 할당되는 영역
 - 빈 영역→할당→할당 해제처럼 상태가 변하는 가변 영역
- 스택 영역 : 프로그램에서 정의한 지역 변수를 저장하는 메모리 영역으로 , 지역 변수를 정의 한 부분에서 할당해 사용
- 빈 공간 : 스택이나 힙과 같이 가변적인 메모리 할당을 위해 유지하고 있는 빈 메모리 영역
 - 프로세스에 할당된 빈 메모리 영역이 모두 소진되면 메모리 부족으로 프로그램 실행이 중단될 수도 있음

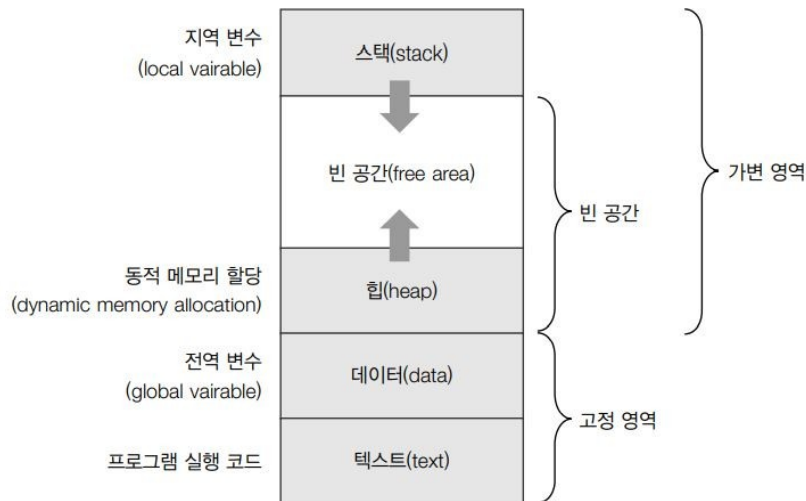


그림 6-2 프로세스의 기본 구조

02. 프로세스의 개념

■ 프로세스 목록 보기

■ PS 명령

- 실행 중인 프로세스의 목록을 보려면 ps 명령을 사용
- 아무 옵션 없이 ps 명령을 사용하면 현재 터미널에서 실행한 프로세스만 출력

```
$ ps
  PID TTY          TIME CMD
 9423 pts/1    00:00:00 bash
17762 pts/1    00:00:00 ps
```

- 시스템에서 동작하고 있는 전체 프로세스를 보려면 -ef 옵션을 지정

```
$ ps -ef | more
UID          PID    PPID  C STIME TTY          TIME CMD
root         1         0  0   3월 19 ?   00:00:10 /sbin/init splash
root         2         0  0   3월 19 ?   00:00:00 [kthreadd]
root         3         2  0   3월 19 ?   00:00:00 [rcu_gp]
root         4         2  0   3월 19 ?   00:00:00 [rcu_par_gp]
root         6         2  0   3월 19 ?   00:00:00 [kworker/0:0H-kblockd]
root         9         2  0   3월 19 ?   00:00:00 [mm_percpu_wq]
( 하략 )
```

02. 프로세스의 개념

■ 시스템 메모리 정보 보기

■ TOP 명령

- 현재 실행 중인 프로세스를 주기적으로 확인해 출력
- top 명령으로 확인할 수 있는 정보 중에서 메모리와 스왑 등에 관한 정보를 직접 검색하려면 sysinfo() 함수를 사용

\$ top

top - 16:32:02 up 1 day, 10:27, 2 users, load average: 0.00, 0.00, 0.00

Tasks: 286 total, 1 running, 285 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

MiB Mem: 3907.0 total, 1253.5 free, 886.3 used, 1767.2 buff/cache

MiB Swap: 1162.4 total, 1162.4 free, 0.0 used. 2735.7 avail Mem

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
711	root	20	0	242144	8160	6836	S	0.3	0.2	2:58.05	vmtoolsd
17664	root	20	0	0	0	0	I	0.3	0.0	0:04.33	kworker+
17770	jw	20	0	13296	4076	3312	R	0.3	0.1	0:00.04	top
1	root	20	0	167832	11996	8556	S	0.0	0.3	0:10.50	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.11	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par+
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker+
9	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_perc+

(하략)

02. 프로세스의 개념

■ 메모리와 스왑 상태 검색 : sysinfo(2)

```
#include <sys/sysinfo.h>
```

[함수 원형]

```
int sysinfo(struct sysinfo *info);
```

- info : 검색 결과를 리턴하는 sysinfo 구조체의 주소
- sysinfo() 함수의 특징
 - sysinfo 구조체에 검색 결과를 저장해 리턴
 - sysinfo() 함수가 성공하면 sysinfo 구조체에 정보를 저장한 후 0 을 리턴하고 오류가 발생하면 - 1 을 리턴

02. 프로세스의 개념

■ 메모리와 스왑 상태 검색 : sysinfo(2)

```
struct sysinfo {  
    long uptime;  
    unsigned long loads[3];  
    unsigned long totalram;  
    unsigned long freeram;  
    unsigned long sharedram;  
    unsigned long bufferram;  
    unsigned long totalswap;  
    unsigned long freeswap;  
    unsigned short procs;  
    unsigned long totalhigh;  
    unsigned long freehigh;  
    unsigned int mem_unit;  
    char _f[20-2*sizeof(long)-sizeof(int)];  
};
```

- **uptime** : 시스템 부팅 후 경과된 시간을 초 단위로 저장
- **loads** : 시스템 부하 평균을 저장하는 배열로, 1 분, 5 분, 15 분 기준으로 계산해 저장
- **totalram** : 사용 가능한 총 메모리 크기를 저장
- **freeram** : 사용 가능한 메모리의 크기를 저장
- **sharedram** : 공유 메모리의 크기를 저장
- **bufferram** : 버퍼가 사용하는 메모리의 크기를 저장
- **totalswap** : 스왑 영역의 총 크기를 저장
- **freeswap** : 사용 가능한 스왑 영역의 크기를 저장
- **procs** : 현재 실행 중인 프로세스 수를 저장
- **totalhigh** : 사용자에게 할당된 메모리의 총 크기를 저장
- **freehigh** : 사용 가능한 사용자 메모리의 크기를 저장
- **mem_unit** : 메모리 크기를 바이트 단위로 저장
- **_f** : 64 바이트 크기를 맞추기 위한 패딩

02. 프로세스의 개념

■ [실습 1] sysinfo() 함수로 메모리 크기 검색하기

```
01 #include <sys/sysinfo.h>
02 #include <stdio.h>
03
04 int main() {
05     struct sysinfo info;
06
07     sysinfo(&info);
08
09     printf("Total Ram: %ld\n", info.totalram);
10     printf("Free Ram: %ld\n", info.freeram);
11     printf("Num of Processes: %d\n", info.procs);
12 }
```

실행

```
$ ch6_1.out
Total Ram: 4096770048
Free Ram: 1307561984
Num of Processes: 505
```


03. 프로세스 식별

■ PID 검색

- PID 는 0 번부터 시작
- 0 번 프로세스 : 스케줄러로 , 프로세스에 CPU 시간을 할당하는 역할 수행 , 커널의 일부분이므로 별도의 실행 파일은 없음
- 1 번 프로세스 : init 로 프로세스가 새로 생성될 때마다 기존 PID 와 중복되지 않은 번호가 할당
- 현재 프로세스의 PID 를 검색하려면 getpid() 함수를 사용

■ PID 검색 : getpid(2)

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

[함수 원형]

```
pid_t getpid(void);
```

- getpid() 함수의 특징
 - 이 함수를 호출한 프로세스의 PID 를 리턴

03. 프로세스 식별

■ PPID 검색 : getppid(2)

■ PPID

- 부모 프로세스 : 0 번 프로세스를 제외한 모든 프로세스에는 자신을 생성한 프로세스
- PPID: 부모 프로세스의 PID
- 부모 프로세스의 PID 를 검색하려면 getppid() 함수를 사용

```
# ps -ef | more
UID      PID PPID C STIME TTY      TIME CMD
root        1  0  0  3 월 19 ?        00:00:10 /sbin/init splash
root        2  0  0  3 월 19 ?        00:00:00 [kthreadd]
root        3  2  0  3 월 19 ?        00:00:00 [rcu_gp]
root        4  2  0  3 월 19 ?        00:00:00 [rcu_par_gp]
( 생략 )
root       709  1  0  3 월 19 ?        00:00:00 /usr/bin/VGAuthService
root       711  1  0  3 월 19 ?        00:02:58 /usr/bin/vmtoolsd
root       729  1  0  3 월 19 ?        00:00:00 /usr/sbin/acpid
( 생략 )
```

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

```
pid_t getppid(void);
```

[함수 원형]

03. 프로세스 식별

■ [실습 2] getpid(), getppid() 함수로 PID 와 PPID 검색하기

```
01 #include <unistd.h>
02 #include <stdio.h>
03
04 int main() {
05     printf("PID : %d\n", (int)getpid());
06     printf("PPID : %d\n", (int)getppid());
07 }
```

실행

```
$ ch6_2.out
PID : 17826
PPID : 9423
```

- **05 행** getpid() 함수로 현재 프로세스의 ID 를 검색해 출력
- **06 행** getppid() 함수로 현재 프로세스의 부모 프로세스 ID 를 검색해 출력
- **실행 결과** PID 는 17826 고 PPID 는 9423 임을 알 수 있음
- ps 명령으로 이 터미널에서 실행 중인 프로세스를 보면 알 수 있음

```
$ ps
  PID TTY          TIME CMD
 9423 pts/1        00:00:00 bash
17832 pts/1        00:00:00 ps
```

04. 프로세스 실행 시간 측정

■ 프로세스 실행 시간

■ 프로세스 실행 시간 측정의 의의와 방법

- 시간 정보를 이용해 프로세스 실행 시간을 측정할 수 있으며 , 이것은 시스템 사용 요금을 결정하는 데 활용할 수 있음
- 프로그램에서 많은 시간을 소비하는 부분을 찾아 개선하는 데도 활용할 수 있음
- 프로세스 실행 시간은 `times()` 함수를 사용해 측정할 수 있음
- 프로세스 실행 시간은 커널 모드에서 실행한 시간과 사용자 모드에서 실행한 시간을 합해 구할 수 있음
- `times()` 함수는 커널 모드에서 실행한 시간과 사용자 모드에서 실행한 시간을 구분해서 알려줌

• 프로세스 실행 시간의 구성

- 시스템 실행 시간 : 프로세스에서 커널의 코드를 수행한 시간으로 , 시스템 호출로 소비한 시간을 의미
- 사용자 실행 시간 : 사용자 모드에서 프로세스를 실행한 시간으로 , 프로그램 내부의 함수나 반복문처럼 사용자가 작성한 코드를 실행하는 데 걸린 시간

프로세스 실행 시간 = 시스템 실행 시간 + 사용자 실행 시간

04. 프로세스 실행 시간 측정

■ tms 구조체

- tms 구조체는 sys/times.h 파일에 다음과 같이 정의

```
struct tms {  
    clock_t tms_utime;  
    clock_t tms_stime;  
    clock_t tms_cutime;  
    clock_t tms_cstime;  
};
```

- **tms_utime** : times() 함수를 호출한 프로세스가 사용한 사용자 모드 실행 시간
- **tms_stime** : times() 함수를 호출한 프로세스가 사용한 시스템 (커널) 모드 실행 시간
- **tms_cutime** : times() 함수를 호출한 프로세스의 모든 자식 프로세스가 사용한 사용자 모드 / 실행 시간과 tms_utime 의 합계 시간
- **tms_cstime** : times() 함수를 호출한 프로세스의 모든 자식 프로세스가 사용한 시스템 모드 / 실행 시간과 tms_stime 의 합계 시간

04. 프로세스 실행 시간 측정

■ 실행 시간 측정 : times(2)

```
#include <sys/times.h>
```

[함수 원형]

```
clock_t times(struct tms *buf);
```

- buf: 실행 시간을 저장할 tms 구조체의 주소
- times() 함수의 특징
 - 프로세스 실행 시간을 인자로 지정한 tms 구조체에 저장
 - times() 함수가 알려주는 시간 단위는 시계의 클럭 틱
 - times() 함수는 임의의 시점으로부터 경과된 클럭 틱 수를 리턴하고 오류가 발생하면 - 1 을 리턴

04. 프로세스 실행 시간 측정

■ [실습 4] times() 함수로 실행 시간 측정하기

```
01 #include <sys/types.h>
02 #include <sys/times.h>
03 #include <time.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main() {
09     int i;
10     time_t t;
11     struct tms buf;
12     clock_t ct, t1, t2;
13
14     ct = sysconf(_SC_CLK_TCK);
15     printf("Clock tick : %ld\n", ct);
16
17     if ((t1 = times(&buf)) == -1) {
18         perror("times 1");
19         exit(1);
20     }
21
22     for (i = 0; i < 99999999; i++)
23         time(&t);
24     sleep(1);
25
26     if ((t2 = times(&buf)) == -1) {
27         perror("times 2");
28         exit(1);
29     }
30
31     printf("t1: %ld\n", t1);
32     printf("t2: %ld\n", t2);
33     printf("utime : %ld\n", buf.tms_utime);
34     printf("stime : %ld\n", buf.tms_stime);
35     printf("Real time : %.1f sec\n", (double)(t2 - t1) / ct);
36     printf("User time : %.1f sec\n", (double)buf.tms_utime / ct);
37     printf("System time : %.1f sec\n", (double)buf.tms_stime / ct);
38 }
```

실행

```
$ ch6_5.out
Clock tick : 100
t1: 1731809110
t2: 1731809250
utime : 39
stime : 0
Real time : 1.4 sec
User time : 0.4 sec
System time : 0.0 sec
```

05. 환경 변수의 활용

■ 환경 변수의 이해

■ 환경 변수

- 환경 변수는 '환경 변수명 = 값' 형태로 구성
- 환경 변수명은 관례적으로 대문자를 사용
- 환경 변수는 셸에서 값을 설정하거나 변경할 수 있으며 함수를 이용해 읽거나 설정할 수 있음
- 현재 셸의 환경 설정을 보려면 env 명령을 사용

```
$ env
SHELL=/bin/bash
PWD=/home/jw
LOGNAME=jw
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/jw
LANG=ko_KR.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd
( 생략 )
```


05. 환경 변수의 활용

■ 전역 변수 사용 : environ

```
#include <unistd.h>

extern char **environ;
```

- 전역 변수 environ 의 특징
 - 전역 변수 environ 은 환경 변수 전체에 대한 포인터
 - 이 변수를 사용해 환경 변수를 검색할 수 있음

05. 환경 변수의 활용

■ [실습 5] environ 전역 변수로 환경 변수 검색하기

```
01 #include <unistd.h>
02 #include <stdio.h>
03
04 extern char **environ;
05
06 int main() {
07     char **env;
08
09     env = environ;
10     while (*env) {
11         printf("%s\n", *env);
12         env++;
13     }
14 }
```

실행

```
$ ch6_6.out
SHELL=/bin/bash
PWD=/home/jw/src/ch6
LOGNAME=jw
XDG_SESSION_TYPE=ttty
MOTD_SHOWN=pam
HOME=/home/jw
LANG=ko_KR.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd
(생략)
```

05. 환경 변수의 활용

■ main() 함수 인자 사용

```
#int main(int argc, char **argv, char **envp)
```

- main() 함수 인자의 특징
 - main() 함수는 아무 인자 없이 사용할 수도 있고 인자를 지정해 사용할 수도 있음
 - 리눅스에서는 환경 변수를 다음과 같이 main() 함수의 세 번째 인자로 지정해 사용할 수 있음
 - 사용 방법은 전역 변수 environ 과 같음

05. 환경 변수의 활용

■ [실습 6] main() 함수 인자로 환경 변수 검색하기

```
01 #include <stdio.h>
02
03 int main(int argc, char **argv, char **envp) {
04     char **env;
05
06     env = envp;
07     while (*env) {
08         printf("%s\n", *env);
09         env++;
10     }
11 }
```

실행

```
$ ch6_7.out
SHELL=/bin/bash
PWD=/home/jw/src/ch6
LOGNAME=jw
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/jw
LANG=ko_KR.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;
33;01:cd
(생략)
```

05. 환경 변수의 활용

■ 환경 변수 검색 : getenv(3)

```
#include <stdlib.h>
```

[함수 원형]

```
char *getenv(const char *name);
```

- name : 환경 변수명
- getenv() 함수의 특징
 - 인자로 지정한 환경 변수가 설정되어 있는지 검색해 결괏값을 저장하고 주소를 리턴
 - 검색에 실패하면 널 포인터를 리턴

05. 환경 변수의 활용

■ [실습 7] getenv() 함수로 환경 변수 검색하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 int main() {
05     char *val;
06
07     val = getenv("SHELL");
08     if (val == NULL)
09         printf("SHELL not defined\n");
10     else
11         printf("SHELL = %s\n", val);
12 }
```

실행

```
$ ch6_8.out
SHELL = /bin/bash
```

05. 환경 변수의 활용

■ 환경 변수 설정 : putenv(3)

```
#include <stdlib.h>
```

[함수 원형]

```
int putenv(char *string);
```

- string : 설정할 환경 변수와 값으로 구성된 문자열
- putenv() 함수의 특징
 - 설정할 환경 변수를 ‘환경 변수명 = 값’ 형태로 지정하여 프로그램에서 환경 변수를 설정
 - putenv() 함수는 기존의 환경 변수값은 변경하고 , 새로운 환경 변수는 malloc() 으로 메모리를 할당해 추가
 - 수행에 성공하면 0 을 리턴

05. 환경 변수의 활용

■ [실습 8] putenv() 함수로 환경 변수 설정하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 int main() {
05     char *val;
06
07     val = getenv("TERM");
08     if (val == NULL)
09         printf("TERM not defined\n");
10     else
11         printf("1. TERM = %s\n", val);
12
13     putenv("TERM=vt100");
14
15     val = getenv("TERM");
16     printf("2. TERM = %s\n", val);
17 }
```

실행

\$ ch6_9.out

```
1. TERM = xterm
2. TERM = vt100
```

- 07~11 행 getenv() 함수를 호출해 환경 변수 TERM의 값을 검색하고 출력
- 13 행 putenv() 함수를 이용해 TERM의 값을 vt100으로 바꿈
- 15~16 행 getenv() 함수를 이용해 TERM의 값을 다시 확인
- 실행 결과 환경 변수 TERM의 값이 변경

05. 환경 변수의 활용

■ 환경 변수 설정 : `setenv(3)`

```
#include <stdlib.h>
```

[함수 원형]

```
int setenv(const char *name, const char *value, int overwrite);
```

- `name` : 환경 변수명
 - `value` : 환경 변수값
 - `overwrite` : 덮어쓰기
-
- `setenv()` 함수의 특징
 - `putenv()` 함수처럼 환경 변수를 설정하지만 다른 점은 변수와 환경 변수값을 각각 인자로 지정
 - `setenv()` 함수는 `name` 에 지정한 환경 변수에 `value` 의 값을 설정
 - `overwrite` 는 `name` 으로 지정한 환경 변수에 이미 값이 설정되어 있을 경우 덮어쓰기 여부를 지정
 - `overwrite` 값이 0 이 아니면 덮어쓰기를 하고 0 이면 덮어쓰기를 하지 않는다

05. 환경 변수의 활용

■ [실습 9] setenv() 함수로 환경 변수 설정하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 int main() {
05     char *val;
06
07     val = getenv("TERM");
08     if (val == NULL)
09         printf("TERM not defined\n");
10     else
11         printf("1. TERM = %s\n", val);
12
13     setenv("TERM","vt100", 0);
14     val = getenv("TERM");
15     printf("2. TERM = %s\n", val);
16
17     setenv("TERM","vt100", 1);
18     val = getenv("TERM");
19     printf("3. TERM = %s\n", val);
20 }
```

실행

\$ ch6_10.out

1. TERM = xterm
2. TERM = xterm
3. TERM = vt100

05. 환경 변수의 활용

■ 환경 변수 설정 삭제 : unsetenv(3)

```
#include <stdlib.h>
```

[함수 원형]

```
int unsetenv(const char *name);
```

- name : 환경 변수명
- unsetenv() 함수의 정보
 - name 에 지정한 환경 변수를 삭제
 - 현재 환경에 name 으로 지정한 환경 변수가 없으면 기존 환경을 변경하지 않음