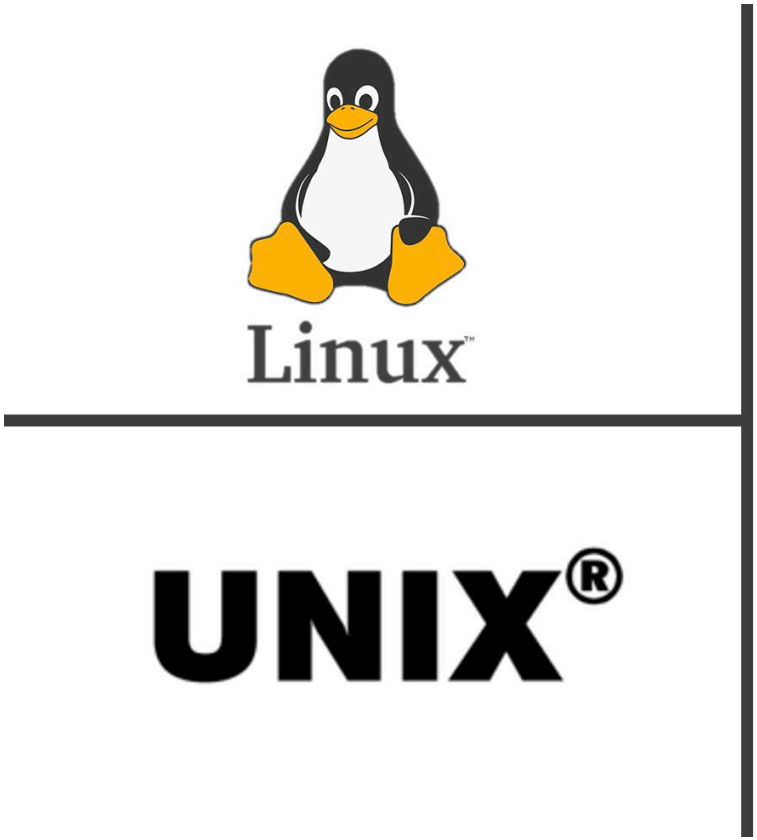

프로세스 정보

목차

학습목표

- 리눅스 시스템에서 프로세스가 무엇인지 이해한다.
- 함수를 사용해 프로세스의 속성을 검색할 수 있다.
- 프로세스의 실행 시간을 측정할 수 있다.
- 환경 변수를 설정하고 사용할 수 있다.

01. 개요

■ 프로세스

▪ 프로세스

- 프로세스: 현재 실행 중인 프로그램을 의미
- 리눅스 시스템에서는 동시에 여러 프로세스가 실행
- 프로세스가 계속 실행 상태에 있는 것은 아니며, 실행, 수면, 실행 대기 등 규칙에 따라 여러 상태로 변함
- 현재 리눅스 시스템에서 실행 중인 프로세스를 확인하려면 ps, top 명령을 사용
- 시스템에서 프로세스를 식별하는 데는 프로세스 ID(PID)를 사용하고, 관련 프로세스들이 모여 프로세스 그룹을 구성

▪ 세션

- POSIX 표준에서 제안한 개념으로, 사용자가 로그인해서 작업하는 터미널 단위로 프로세스 그룹을 묶은 것

표 6-1 프로세스 식별 함수

기능	함수
메모리와 스왑 상태 검색	int sysinfo(struct sysinfo *info);
PID 검색	pid_t getpid(void);
부모 PID 검색	pid_t getppid(void);
프로세스 그룹 ID 검색	pid_t getpgrp(void); pid_t getpgid(pid_t pid);
프로세스 그룹 ID 변경	int setpgid(pid_t pid, pid_t pgid);
세션 리더 ID 검색	pid_t getsid(pid_t pid);
세션 생성	pid_t setsid(void);

01. 개요

■ 프로세스

▪ 프로세스 실행 시간

- 시스템 실행 시간: 프로세스에서 커널의 코드를 수행한 시간
- 사용자 실행 시간: 사용자 모드에서 프로세스를 실행한 시간
- 프로세스 실행 시간을 측정하려면 times( ) 함수를 사용

표 6-2 프로세스 실행 시간 측정 함수

기능	함수
프로세스 실행 시간 측정	clock_t times(struct tms *buf);

▪ 환경 변수

- 환경 변수를 사용하면 프로세스 환경을 설정하거나 설정된 환경을 검색할 수 있음

표 6-3 프로세스 환경 설정 함수

기능	전역 변수와 함수
환경 설정 전역 변수 사용	extern char **environ;
환경 변수 검색	char *getenv(const char *name);
환경 변수 설정 및 삭제	int putenv(char *string);
	int setenv(const char *name, const char *value, int overwrite);
	int unsetenv(const char *name);

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

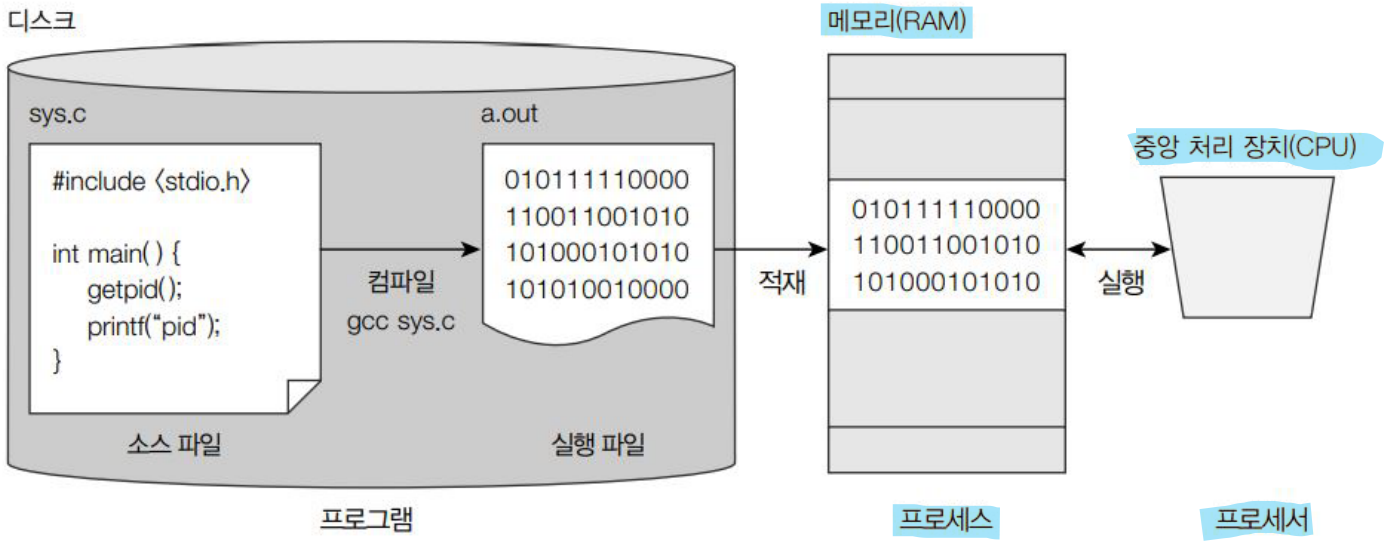

그림 6-1 프로그램과 프로세스, 프로세서의 관계

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

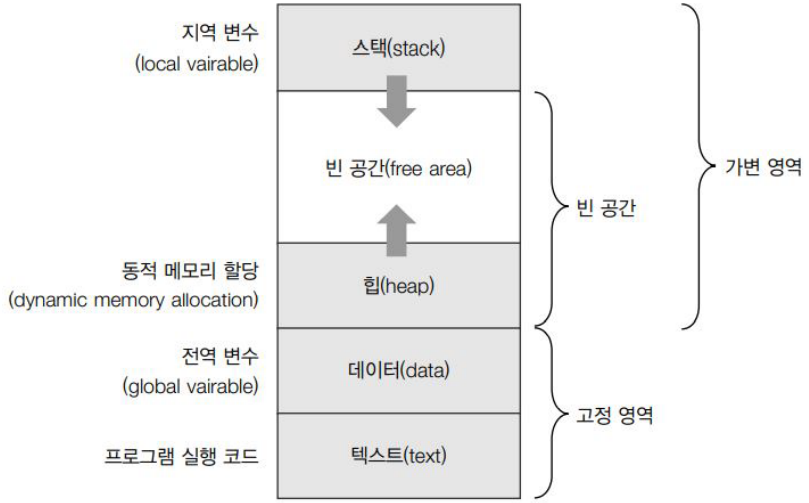

그림 6-2 프로세스의 기본 구조

02. 프로세스의 개념

■ 프로세스 상태 변화

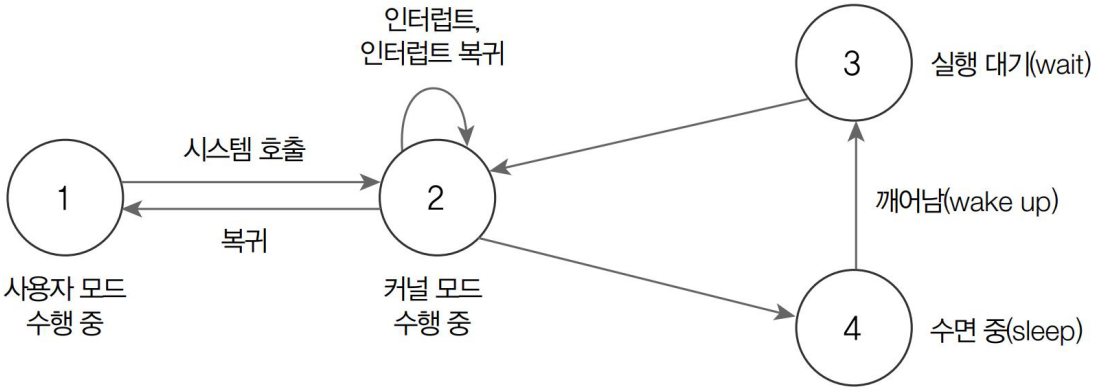

그림 6-3 프로세스의 상태 및 전이

① 프로세스는 사용자 모드에서 먼저 실행됨
② 사용자 모드에서 시스템 호출을 하면 커널 모드로 전환되어 실행
③ 수면 중이던 프로세스가 깨어나 실행 대기 상태가 되면 바로 실행할 수 있도록 준비
④ 커널 모드에서 실행 중 입출력 완료를 기다릴 때와 같이 더 이상 실행을 계속할 수 없을 때 수면 상태로 전환

- 수면 상태 : 입출력을 완료했을 때 혹은 다른 프로세스가 종료되기를 기다릴 때는 프로세스가 잠든 때
- 실행 대기 상태: 수면 상태에서 기다리다가 해당 사건이 발생하면 깨어나 실행 대기 상태로 전환

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  3월19 ?       00:00:10 /sbin/init splash
root           2       0  0  3월19 ?       00:00:00 [kthreadd]
root           3       2  0  3월19 ?       00:00:00 [rcu_gp]
root           4       2  0  3월19 ?       00:00:00 [rcu_par_gp]
root           6       2  0  3월19 ?       00:00:00 [kworker/0:0H-kblockd]
root           9       2  0  3월19 ?       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 저정한 후 0을 리턴하고 오류가 발생하면 –1을 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, 1분, 5분, 15분 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 :** 사용자에 할당된 메모리의 총 크기를 저장
- **freehigh :** 사용 가능한 사용자 메모리의 크기를 저장
- **mem_unit** : 메모리 크기를 바이트 단위로 저장
- **_f** : 64바이트 크기를 맞추기 위한 패딩

02. 프로세스의 개념

■ [예제 6-1] 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
```

- **07행** sysinfo() 함수로 정보를 실행
- **09행** 시스템의 총 메모리 크기를 출력한다.
- **10행** 시스템에서 현재 사용 가능한 메모리의 크기를 출력한다.
- **11행** 현재 실행 중인 프로세스 수를 출력
- **실행 결과** 시스템의 총 메모리는 4096770048바이트(약 4GB)이고, 사용 가능한 메모리는 1307561984바이트(약 1GB), 현재 실행 중인 프로세스는 총 505개임을 알 수 있음

03. 프로세스 식별

■ PID 검색

- PID는 0번부터 시작
- 0번 프로세스: 스케줄러로, 프로세스에 CPU 시간을 할당하는 역할 수행, 커널의 일부분이므로 별도의 실행 파일은 없음
- 1번 프로세스: init로 프로세스가 새로 생성될 때마다 기존 PID와 중복되지 않은 번호가 할당
- 현재 프로세스의 PID를 검색하려면 getpid() 함수를 사용

■ PID 검색 : getpid(2)

[함수 원형]

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

pid_t getpid(void);
```

- getpid() 함수의 특징
  - 이 함수를 호출한 프로세스의 PID를 리턴

03. 프로세스 식별

■ PPID 검색 : getppid(2)

▪ PPID

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

```
# ps -ef | more
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 3월19 ?        00:00:10 /sbin/init splash
root           2     0  0 3월19 ?        00:00:00 [kthreadd]
root           3     2  0 3월19 ?        00:00:00 [rcu_gp]
root           4     2  0 3월19 ?        00:00:00 [rcu_par_gp]
(생략)
root         709     1  0 3월19 ?        00:00:00 /usr/bin/VGAuthService
root         711     1  0 3월19 ?        00:02:58 /usr/bin/vmtoolsd
root         729     1  0 3월19 ?        00:00:00 /usr/sbin/acpid
(생략)
```

```
#include <sys/types.h>                                    [함수 원형]
#include <unistd.h>

pid_t getppid(void);
```

03. 프로세스 식별

■ [예제 6-2] getpid(), getppid() 함수로 PID와 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
```

- **05행** getpid() 함수로 현재 프로세스의 ID를 검색해 출력
- **06행** getppid() 함수로 현재 프로세스의 부모 프로세스 ID를 검색해 출력
- **실행 결과** PID는 17826고 PPID는 9423임을 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
```

03. 프로세스 식별

■ PGID 검색 : getpgrp(2), getpgid(2)

```
#include <sys/types.h>                                    [함수 원형]
#include <unistd.h>

pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

- pid : PGID를 구하려는 프로세스의 ID

- getpgrp( ) 함수의 특징
  - 이 함수를 호출하는 프로세스가 속한 그룹의 PGID를 리턴
  - getpgid() 함수는 pid 인자로 지정한 프로세스가 속한 그룹의 PGID를 리턴
  - 만일 인자가 0이면 getpgid() 함수를 호출한 프로세스의 PID를 리턴

03. 프로세스 식별

■ [예제 6-3] getpgrp(), getpgid() 함수로 PGID 검색하기

```
01 #include <unistd.h>
02 #include <stdio.h>
03
04 int main() {
05     printf("PID : %d₩n", (int)getpid());
06     printf("PGRP : %d₩n", (int)getpgrp());
07     printf("PGID(0) : %d₩n", (int)getpgid(0));
08     printf("PGID(18020) : %d₩n", (int)getpgid(18020));
09 }
```

실행

```
$ ps
    PID TTY          TIME CMD
   9423 pts/1    00:00:00 bash
  17900 pts/1    00:00:00 ps
$ ch6_3.out
PID : 18028
PGRP : 18028
PGID(0) : 18028
PGID(18020) : 18018
```

- **05행** PID를 검색해 출력
- **06~07행** 현재 프로세스의 PGID를 두 가지 방법으로 검색
- **08행** PID가 18020인 프로세스가 속한 그룹의 PGID를 검색

```
$ ps -ef | more | sleep 300 &
$ ps
    PID TTY          TIME CMD
   9423 pts/1    00:00:00 bash
  18020 pts/1    00:00:00 sleep
  18021 pts/1    00:00:00 ps
```

- **실행 결과** PID와 PGID가 같음을 알 수 있음, 이는 현재 프로세스가 단독 프로세스여서 프로세스 그룹에 홀로 속해 있기 때문
  파이프로 연결해 생성한 프로세스 그룹의 PGID를 검색한 결과는 18018로, 인자로 지정한 PID 18020과는 다른 프로세스임을 알 수 있음

03. 프로세스 식별

■ pgid 변경 : setpgid(2)

```
#include <sys/types.h>                                    [함수 원형]
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

- pid : 프로세스 그룹에 속한 프로세스의 ID
- pgid : 새로 지정할 PGID

- setpgid() 함수의 특징
  - pid가 가리키는 프로세스의 PGID를 pgid로 지정한 값으로 지정
  - pid와 pgid가 같다면 pid에 해당하는 프로세스가 그룹 리더가 됨
  - 만일 pid가 0이면 이 함수를 호출하는 현재 프로세스의 PID를 사용
  - pgid가 0이면 pid로 지정한 프로세스가 PGID

03. 프로세스 식별

■ 세션

- 세션의 정보
  - POSIX 표준에서 제안한 개념으로 사용자가 로그인해 작업하고 있는 터미널 단위로 프로세스 그룹을 묶은 것
  - 프로세스 그룹이 관련 있는 프로세스를 그룹으로 묶은 개념이라면, 세션은 관련 있는 프로세스 그룹을 모은 개념
  - 프로세스 그룹 단위로 작업 제어를 수행할 때 사용

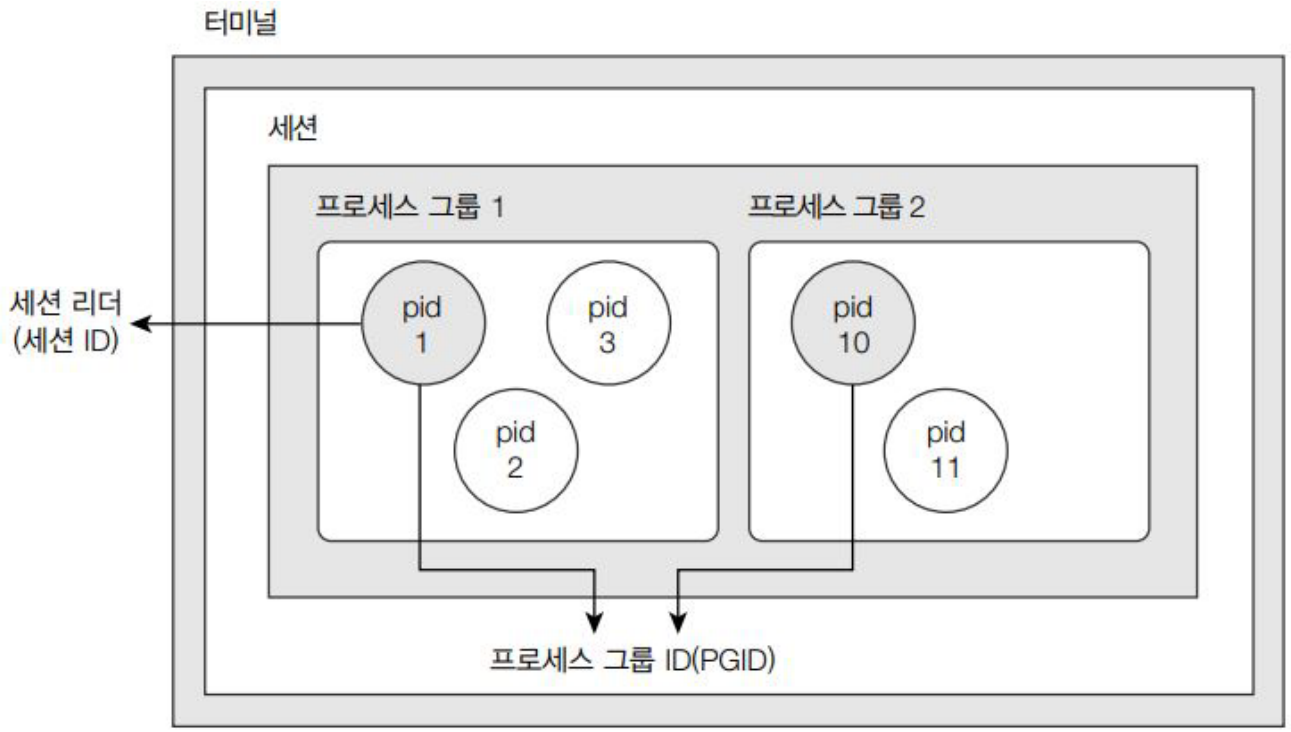

그림 6-4 프로세스, 프로세스 그룹, 세션의 관계

03. 프로세스 식별

■ 세션 검색 : getsid(2)

```
#include <sys/types.h>                                    [함수 원형]
#include <unistd.h>

pid_t getsid(pid_t pid);
```

- pid : 자신이 속한 세션의 ID를 구하려는 프로세스의 ID
- 세션 Id
  - 프로세스가 새로운 세션을 생성하면 해당 프로세스는 세션 리더가 되고 세션 리더의 PID는 세션 ID가 됨
- getsid() 함수의 특징
  - pid로 지정한 프로세스가 속한 세션의 ID를 리턴
  - 만일 pid가 0이면 현재 프로세스의 세션 ID를 리턴

03. 프로세스 식별

■ [예제 6-4] getsid() 함수로 세션 ID 검색하기

```
01 #include <unistd.h>
02 #include <stdio.h>
03
04 int main() {
05      printf("PID : %d₩n", (int)getpid());
06      printf("PGID : %d₩n", (int)getpgrp());
07      printf("SID : %d₩n", (int)getsid(0));
08 }
```

실행

```
$ ch6_4.out
PID : 18094
PGID : 18094
SID : 9423
$ ps
    PID TTY          TIME CMD
   9423 pts/1    00:00:00 bash
  18095 pts/1    00:00:00 ps
```

- **05행** 현재 프로세스의 ID를 검색해 출력
- **06행** 현재 프로세스가 속한 그룹의 ID를 검색해 출력
- **07행** 현재 프로세스가 속한 세션의 ID를 검색해 출력
- **실행 결과** PID와 PGID가 같다는 사실을 알 수 있다. 세션 ID는 배시 셸의 PID로 이 세션의 리더가 배시 셸임을 알 수 있음

03. 프로세스 식별

■ 세션 생성 : setsid(2)

```
#include <sys/types.h>                                    [함수 원형]
#include <unistd.h>

pid_t setsid(void);
```

- setsid() 함수의 특징
  - setsid() 함수를 사용하면 새로운 세션을 만들 수 있음
  - setsid() 함수를 호출하는 프로세스가 프로세스 그룹 리더가 아니면 새로운 세션을 만들어 세션 리더가 됨
  - 새로운 세션에서 프로세스 그룹 리더가 됨
  - setsid() 함수를 호출한 프로세스가 새로운 세션과 프로세스 그룹의 유일한 프로세스가 됨
  - 호출에 성공하면 새로운 세션 ID를 리턴하고 오류가 발생하면 –1 을 리턴

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 모드에 서 실행한 시간과 사용자 모드에서 실행한 시간을 합해 구할 수 있음
- times() 함수는 커널 모드에 서 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_utime의 합계 시간
- **tms_cstime** : times() 함수를 호출한 프로세스의 모든 자식 프로세스가 사용한 시스템 모드/실행 시간과 tms_stime의 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 클록 틱
  - times() 함수는 임의의 시점으로부터 경과된 클록 틱 수를 리턴하고 오류가 발생하면 –1을 리턴

04. 프로세스 실행 시간 측정

■ [예제 6-5] times() 함수로 실행 시간 측정하기

```c
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int i;
    time_t t;
    struct tms buf;
    clock_t ct, t1, t2;

    ct = sysconf(_SC_CLK_TCK);
    printf("Clock tick : %ld₩n", ct);

    if ((t1 = times(&buf)) == -1) {
        perror("times 1");
        exit(1);
    }

    for (i = 0; i < 99999999; i++)
        time(&t);
    sleep(1);

    if ((t2 = times(&buf)) == -1) {
        perror("times 2");
        exit(1);
    }

    printf("t1: %ld₩n", t1);
    printf("t2: %ld₩n", t2);
    printf("utime : %ld₩n", buf.tms_utime);
    printf("stime : %ld₩n", buf.tms_stime);
    printf("Real time : %.1f sec₩n", (double)(t2 - t1) / ct);
    printf("User time : %.1f sec₩n", (double)buf.tms_utime / ct);
    printf("System time : %.1f sec₩n", (double)buf.tms_stime / ct);
}
```

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
```

04. 프로세스 실행 시간 측정

■ [예제 6-5] times() 함수로 실행 시간 측정하기

- **14행** sysconf(_SC_CLK_TCK) 함수로 클록 틱 값을 검색, 15행의 출력 결과를 보면 초당 클록 틱 값은 100
- **17행** times() 함수를 호출해 리턴받은 현재 클록 틱 값을 저장
- **22~24행** 실제 작업을 수행하는 부분으로, time() 함수를 9,999,999번 호출하고 sleep(1) 함 수를 호출해 1초간 대기
- **26행** 다시 times() 함수를 호출해 클록 틱 값을 측정
- **31~32행** 17행과 26행에서 times() 함수를 호출해 리턴받은 클록 틱 값을 출력, 실행 결과를 보면 두 값의 차이는 100
- **33~34행** times() 함수가 검색해 저장한 tms 구조체의 값 중에서 tms_utime과 tms_stime 값을 출력
  실행 결과를 보면 tms_utime이 39, tms_stime이 0
- **35행** 함수 실행 결과인 클록 틱 값을 초당 클록 틱으로 나누어 프로그램 실행에 소요된 전체 시간을 출력
- **36~37행** 사용자 실행 시간, 시스템 실행 시간을 초 단위로 환산해 출력
- **실행 결과** 전체 실행 시간이 1.4초, 사용자 실행 시간이 0.4초, 시스템 실행 시간이 0초임을 알 수 있음
  전체 실행 시간과 사용자 실행 시간이 차이 나는 이유는 sleep() 함수로 1초 대기했기 때문

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 environ의 특징
  - 전역 변수 environ은 환경 변수 전체에 대한 포인터
  - 이 변수를 사용해 환경 변수를 검색할 수 있음

05. 환경 변수의 활용

■ [예제 6-6] 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=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)
```

- **09행** environ의 주소를 임시 포인터인 env에 저장
- **10~13행** env의 주소를 증가시키며 환경 변수를 출력
- **실행 결과** 실행 결과가 env 명령의 결과와 동일함을 알 수 있음

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 environ과 같음

05. 환경 변수의 활용

■ [예제 6-7] 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)
```

- **03행** main() 함수의 인자로 환경 변수에 대한 포인터를 받음
- **06~10행** 인자로 받은 환경 변수의 값을 차례로 출력
- **실행 결과** [예제 6-6]과 동일

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

■ [예제 6-8] 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
```

- **07행** getenv() 함수를 호출해 환경 변수 SHELL을 검색하고 결괏값을 문자형 포인터 val에 저장
- **08~11행** getenv() 함수의 리턴값이 널이면 SHELL 변수가 정의되어 있지 않은 것, 리턴 값이 널이 아니면 리턴한 결과를 출력
- **실행 결과** SHELL의 값이 /bin/bash임을 알 수 있음

05. 환경 변수의 활용

■ 환경 변수 설정 : putenv(3)

```
#include <stdlib.h>                                    [함수 원형]

int putenv(char *string);
```

- string : 설정할 환경 변수와 값으로 구성한 문자열

- putenv() 함수의 특징
  - 설정할 환경 변수를 '환경 변수명=값' 형태로 지정하여 프로그램에서 환경 변수를 설정
  - putenv() 함수는 기존의 환경 변숫값은 변경하고, 새로운 환경 변수는 malloc()으로 메모리를 할당해 추가
  - 수행에 성공하면 0을 리턴

05. 환경 변수의 활용

■ [예제 6-9] 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
```

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

- [예제 6-9]를 실행한 후 다시 env 명령으로 검색해보면 TERM의 값이 예제를 실행하기 이전 상태로 남아 있음

```
$ env | grep TERM
TERM=xterm
```

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 변숫값
- overwrite : 덮어쓰기

- setenv() 함수의 특징
  - putenv() 함수처럼 환경 변수를 설정하지만 다른 점은 변수와 환경 변숫값을 각각 인자로 지정
  - setenv() 함수는 name에 지정한 환경 변수에 value의 값을 설정
  - overwrite는 name으로 지정한 환경 변수에 이미 값이 설정되어 있을 경우 덮어쓰기 여부를 지정
  - overwrite 값이 0이 아니면 덮어쓰기를 하고 0이면 덮어쓰기를 하지 않는다

05. 환경 변수의 활용

■ [예제 6-10] 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
```

- **07~11행** getenv() 함수를 호출해 환경 변수 TERM의 값을 검색해 출력
- **13행** setenv() 함수로 TERM의 값을 vt100으로 변경하고, overwrite 값을 0으로 지정
- **14~15행** 다시 getenv() 함수로 TERM의 값을 확인
- **17행** setenv() 함수로 TERM의 값을 vt100으로 변경, overwrite 값은 1로 지정
- **18~19행** 다시 getenv() 함수로 TERM의 값을 확인
- **실행 결과** 13행과 같이 overwrite 값이 0일 때는 TERM의 값이 변경되지 않고, 17행과 같이 overwrite의 값이 1일 때 변경됨을 알 수 있음

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
  - name에 지정한 환경 변수를 삭제
  - 현재 환경에 name으로 지정한 환경 변수가 없으면 기존 환경을 변경하지 않음