인덕대학교

프로세스 관리
- 프로세스 관리

컴퓨터소프트웨어학과

김병국 교수

학습목표

- 운영체제 실습을 위한 환경을 구축할 수 있다.
- 프로세스의 상태를 제어할 수 있다.
- 문맥 교환에 대한 동작 방식을 이해한다.
- fork() 함수를 사용할 수 있다.

목차

1. 실습환경 구축 (1/3)

❑ 가상 머신(Virtual Machine)

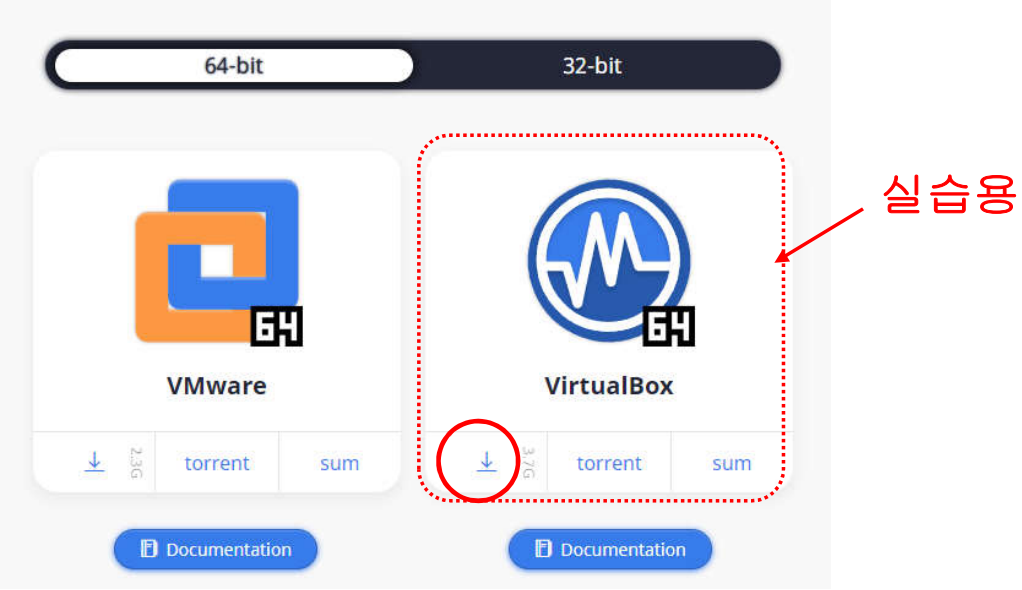
- 게스트 운영체제의 동작 환경을 제공하는 가상의 하드웨어 환경을 제공
- 실습용 가상 머신 : VirtualBox
  - 다운로드: https://www.virtualbox.org/ (공짜)

[실습용 가상 머신]

1. 실습환경 구축 (2/3)

❑ 운영체제

- 리눅스 배포판을 통한 운영체제 기능을 실험
- 실습용 운영체제 : Kali Linux (버전: 2022.1)
  - 데이반 리눅스계열로 각종 해킹 관련 툴들이 포함된 리눅스 배포판
  - 다운로드: https://www.kali.org/get-kali/#kali-virtual-machines (공짜)



[Kali Linux 이미지 다운로드 화면]

1. 실습환경 구축 (3/3)

❑ 로그인 정보

- 계정 : kali , 비밀번호: kali

❑ 실습을 위한 환경 설정

- 쉘 변경
  - 명령어: chsh

```
[kali@kali:~]$chsh
Password:
Changing the login shell for kali
Enter the new value, or press ENTER for the default
        Login Shell [/usr/bin/bash]: /usr/bin/bash
[kali@kali:~]$echo $0
/usr/bin/bash
[kali@kali:~]$
```

추천 쉘

현재 쉘 확인(환경변수 **$0**)

[쉘 변경 명령 및 현재 쉘 확인]

- 프롬프트 변경
  - 파일명: ~/.bashrc

```
[kali@kali ~]$echo $PS1
[\u@\h \W]\$
[kali@kali ~]$
```

2. 프로세스 상태 제어 실습 (1/3)

인덕대학교

❑ 실습 준비

- 가상머신: VirtualBox
- 운영체제: Kali Linux

❑ 정지(휴식) 상태 실습 (1/3)

- 명령어: ps
  - 프로세스들에 대한 상태정보를 표시

```
[kali@kali ~]$sleep 100 &
[1] 2608
[kali@kali ~]$ps f
    PID TTY      STAT   TIME COMMAND
   2451 pts/1    Ss+    0:00 /usr/bin/bash
   2000 pts/0    Ss     0:00 /usr/bin/bash
   2608 pts/0    S      0:00  \_ sleep 100
   2609 pts/0    R+     0:00  \_ ps f
[kali@kali ~]$
```

❑정지(휴식) 상태 실습 (2/3)

- 명령어: kill
  - 지정한 프로세스에게 시그널을 전송
  - 시그널 번호는 앞에 '-' 문자를 붙임
  - 시그널의 리스트는 'kill -l' 명령을 통해 확인 가능

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

2. 프로세스 상태 제어 실습 (3/3)

❑ 정지(휴식) 상태 실습 (3/3)

1. 두 개의 터미널 창 생성
2. 한 곳(A창)에서는 “sleep 1000” 명령 실행
3. 다른 창(B창)에서는 “ps f”명령을 통해 프로세스 상태 파악
4. B창에서 “kill -19 [sleep에 해당하는 PID]”를 입력
5. B창에서 “ps f”명령을 통해 프로세스 상태 다시 파악
6. B창에서 “kill -18 [sleep에 해당하는 PID]”를 입력
7. B창에서 “ps f”명령을 통해 프로세스 상태 다시 파악

```
[kali@kali ~]$ps f
    PID TTY      STAT   TIME COMMAND
   2451 pts/1    Ss     0:00 /usr/bin/bash
   2612 pts/1    S+     0:00  \_ sleep 1000
   2000 pts/0    Ss     0:00 /usr/bin/bash
   2613 pts/0    R+     0:00  \_ ps f
[kali@kali ~]$kill -19 2612
[kali@kali ~]$ps f
    PID TTY      STAT   TIME COMMAND
   2451 pts/1    Ss+    0:00 /usr/bin/bash
   2612 pts/1    T      0:00  \_ sleep 1000
   2000 pts/0    Ss     0:00 /usr/bin/bash
   2614 pts/0    R+     0:00  \_ ps f
[kali@kali ~]$kill -18 2612
[kali@kali ~]$ps f
    PID TTY      STAT   TIME COMMAND
   2451 pts/1    Ss+    0:00 /usr/bin/bash
   2612 pts/1    S      0:00  \_ sleep 1000
   2000 pts/0    Ss     0:00 /usr/bin/bash
   2615 pts/0    R+     0:00  \_ ps f
[kali@kali ~]$
```

```
[kali@kali:~]$
[kali@kali:~]$sleep 1000

[1]+  Stopped                 sleep 1000
[kali@kali:~]$
[kali@kali:~]$jobs
[1]+  Running                 sleep 1000 &
[kali@kali:~]$
```

3. 문맥 교환 (1/2)

❑ 요리 작업의 전환

- 주문서를 바꾸는 것과 동시에 작업 환경을 바꾸는 것

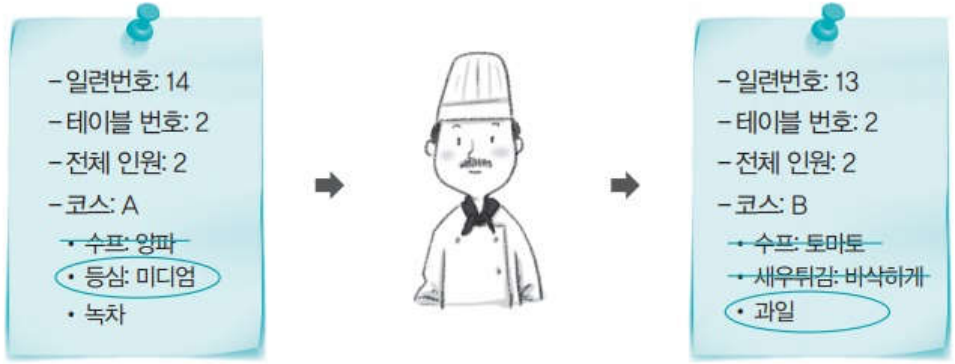


그림 3-14 **요리 작업의 전환 과정**

❑ 문맥 교환

- CPU를 차지하던 프로세스를 옮기고 새로운 프로세스를 실행
- 직전 작업 정보들을 모두 PCB를 통해 백업
- 원복 시 기록된 PCB 백업정보들을 재활용

3. 문맥 교환 (2/2)

❑문맥 교환 절차

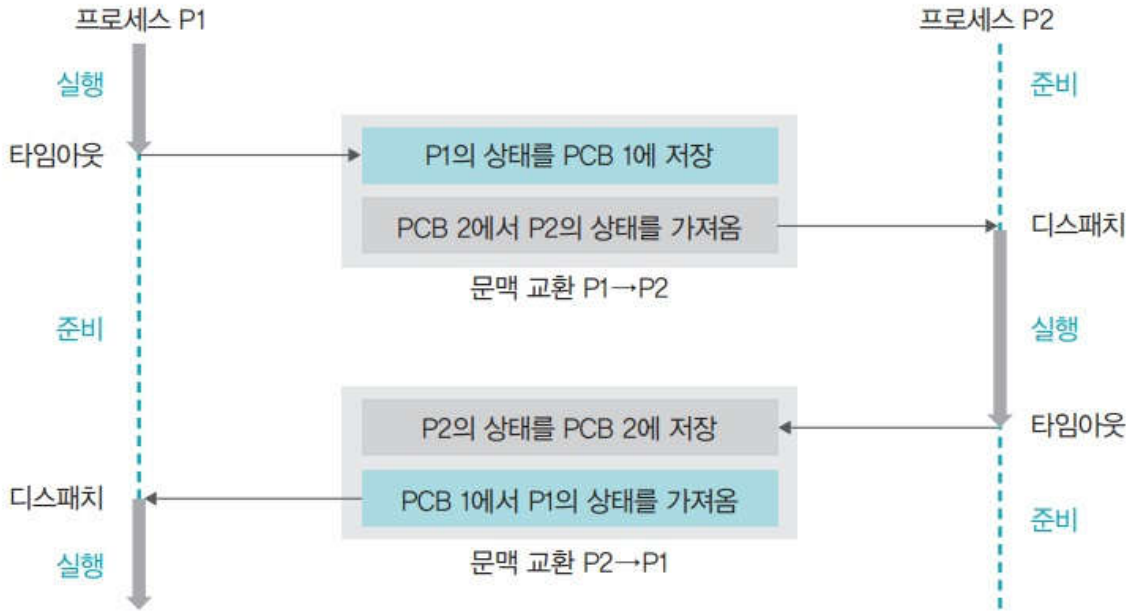


그림 3-15 **문맥 교환 과정**

4. 프로세스의 구조 (1/2)

❑ 메모리내 프로세스 구조 (1/2)

- 코드 영역(Code/Text/Instruction Area)
- 데이터 영역(Data Area)

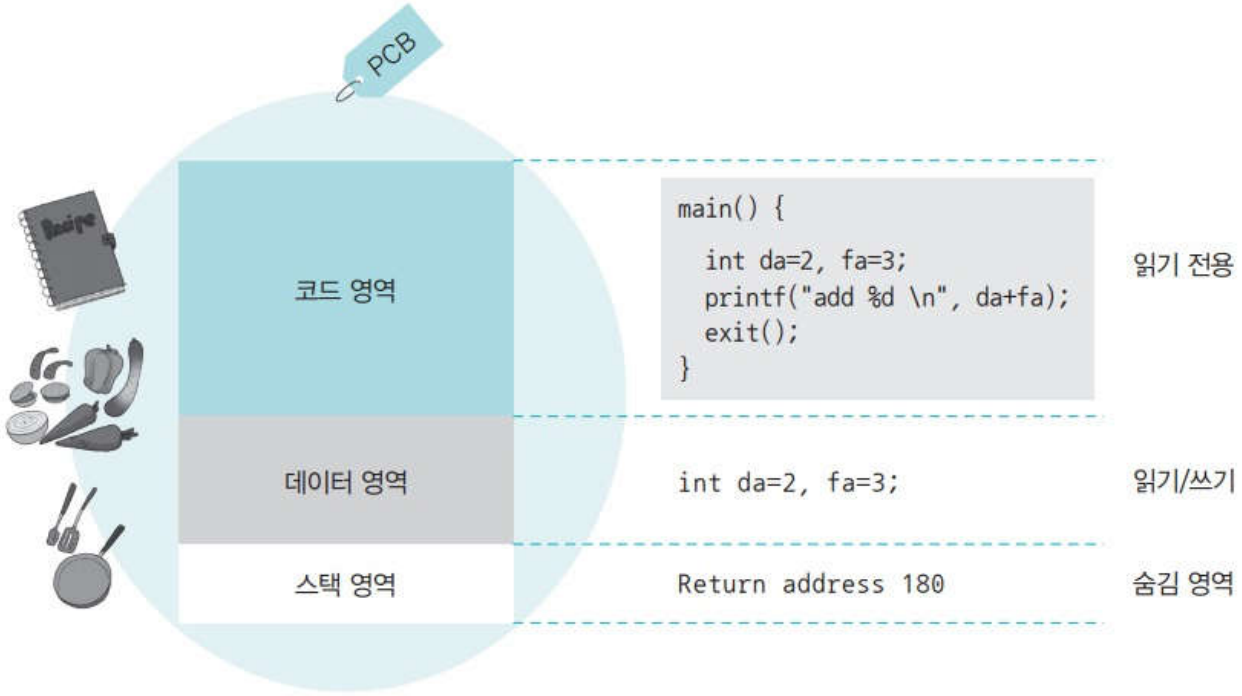


그림 3-16 프로세스의 구조

4. 프로세스의 구조 (2/2)

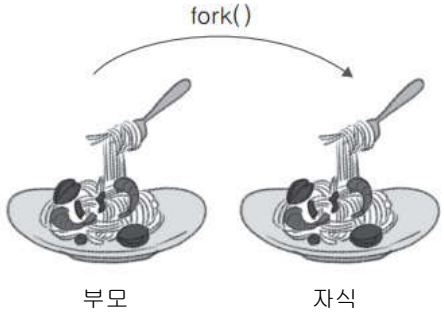
❑ 메모리내 프로세스 구조 (2/2)

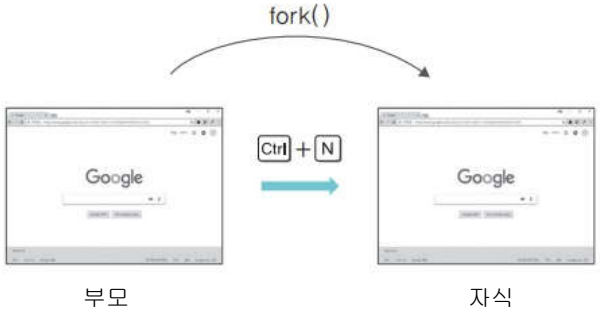
- 코드 영역(Code/Text/Instruction Area)
  - 프로그램 명령들이 기술된 곳
  - 읽기 전용
- 데이터 영역(Data Area)
  - 코드가 실행되면서 사용하는 변수나 파일 등의 각종 데이터를 모아놓은 곳
  - 읽기와 쓰기 가능
- 스택 영역(Stack Area)
  - 운영체제가 프로세스를 실행하기 위해 부수적으로 필요한 데이터를 모아놓은 곳
  - 함수를 호출하면 함수를 수행하고 원래 프로그램으로 되돌아올 위치를 이 영역에 저장
  - 운영체제가 사용자의 프로세스를 작동하기 위해 유지하는 영역
  - 사용자에게는 보이지 않음

5. 프로세스 생성과 복사 (1/3)

❑ fork() 시스템 함수

- 실행중인 프로세스로부터 새로운 프로세스를 생성
- 자신과 동일한 프로세스를 자식 프로세스로 생성



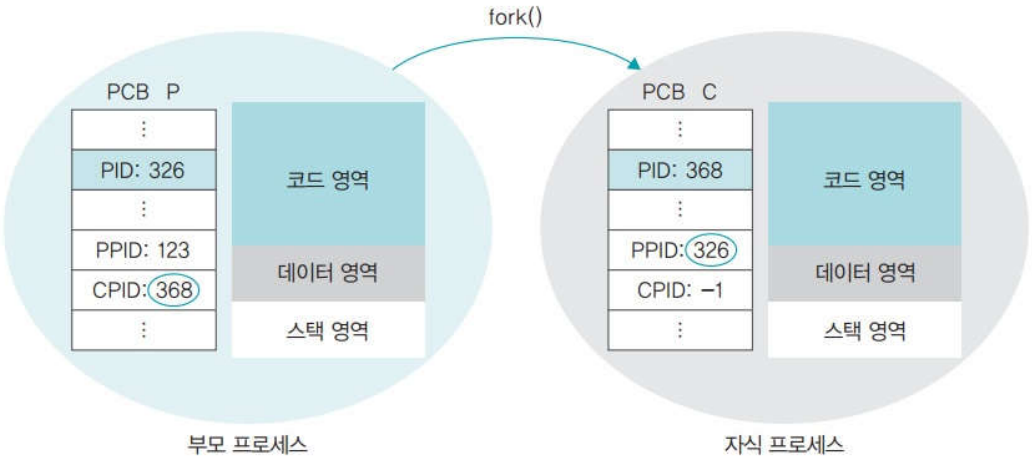


5. 프로세스 생성과 복사 (2/3)

❑ fork() 동작 과정

- PCB를 포함한 부모 프로세스 영역의 대부분(데이터와 스택 영역)이 자식 프로세스에 복사되어 새로운 프로세스가 만들어짐

- 단, PCB의 내용 중 다음이 변경됨
  - 프로세스 구분자(PID)
  - 메모리 관련 정보
  - 부모 프로세스 구분자(PPID)
  - 자식 프로세스 구분자(CPID)



5. 프로세스 생성과 복사 (3/3)

❑ fork() 사용의 장점

- 이미 메모리에 적재된 프로세스를 기반으로 새로 만들어지기 때문에 프로세스의 생성 속도가 빠름
- 추가 작업 없이 자원을 상속할 수 있음
- 시스템 관리를 효율적으로 할 수 있음
  - 부모-자식 관계의 구조 때문임

6. fork() 함수 실습

❑코드 작성 및 실행

- 코드 예시

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

int
main(void)
{
        int nPid=0;

        nPid = fork();

        if(nPid == 0)
        {
                printf("I am a child.(PID: %d)\n", nPid);
        }
        else
        {
                printf("I am a parent.(PID: %d)\n", nPid);
        }

        return 0;
}
```

- 실행 결과

```
[kali@kali 03_2]$vi fork.c
[kali@kali 03_2]$gcc fork.c -o fork
[kali@kali 03_2]$./fork
I am a parent.(PID: 3455)
I am a child.(PID: 0)
[kali@kali 03_2]$
```

7. 프로세스의 계층 구조 (1/4)

❑유닉스의 프로세스 계층 구조

- 유닉스의 모든 프로세스는 init 프로세스의 자식이 되어 트리 구조를 이룸
- ps명령을 통해 확인할 경우 init가 1번으로 할당됨을 확인해 볼 수 있음

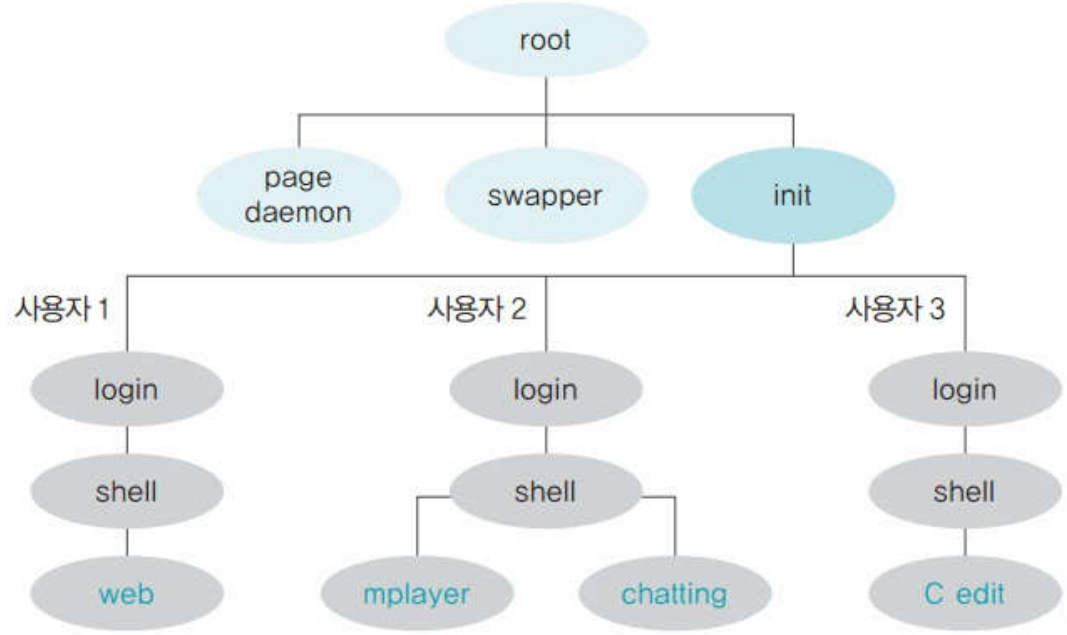


그림 3-23 유닉스의 프로세스 계층 구조

7. 프로세스의 계층 구조 (2/4)

❑장점 (1/2)

- 체계화된 프로세스들에 대한 관리가 가능
- 로그인을 위한 프로세스의 경우, 사용자별 fork()를 수행하면되므로 메모리 중복(코드영역)을 최소화 할 수 있음

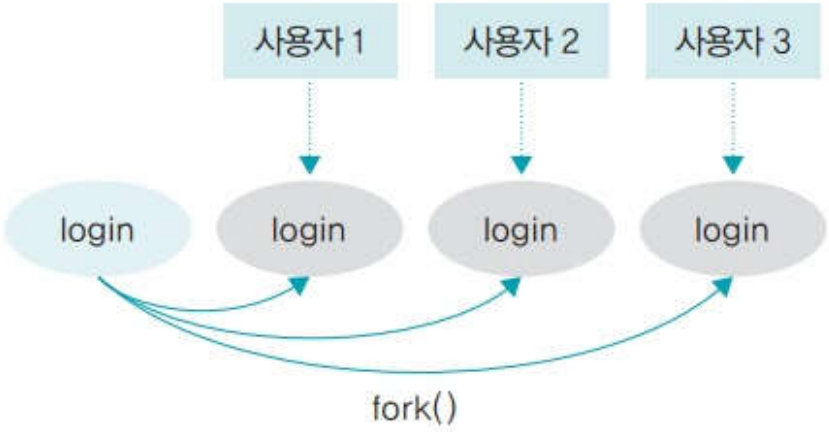
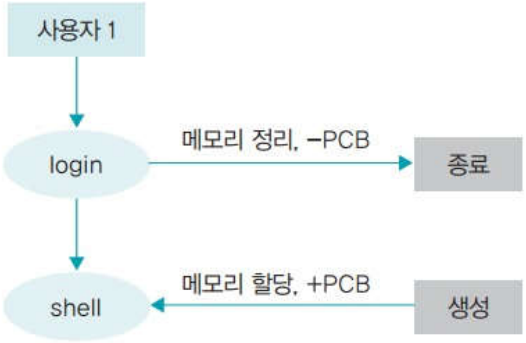


그림 3-24 **여러 사용자를 동시 처리**

❑장점 (2/2)

- 재사용이 용이



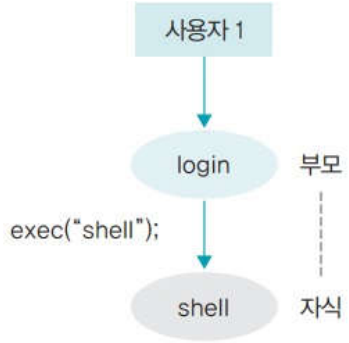


그림 3-25 **프로세스의 재사용**

- 자원 회수가 쉬움
  - 상하 관계로 인해 프로세스의 관리 및 호출 주체를 파악하기 쉬움

7. 프로세스의 계층 구조 (4/4)

❑ 고아(Orphan) 프로세스

- 프로세스가 종료된 후에도 비정상적으로 남아 있는 프로세스
- 부모 프로세스가 자식보다 먼저 죽는 경우
- 해결안:
  - C 언어의 exit( ) 또는 return 문을 통해 자식 프로세스가 작업이 끝났음을 부모 프로세스에 알림

```
int main() {
    printf("Hello \n");
    exit(0);
}
```

그림 3-26 exit 함수

수고하셨습니다.