

Chapter

03

프로세스와 프로세스 관리

1. 프로세스 개요
2. 커널의 프로세스 관리
3. 프로세스의 계층 구조
4. 프로세스 제어(fork, exec, wait, exit)



강의 목표

1. 프로세스는 운영체제가 응용프로그램을 실행시키고 종료할 때까지 관리하기 위해 만든 정보 체계임을 안다.
2. 프로세스의 구성, 주소 공간, 생명 주기 등에 관해 충분히 이해한다.
3. 커널이 어떻게 프로세스를 관리하는지 이해한다.
4. 프로세스들 사이의 부모 자식 계층 구조를 이해한다.
5. 프로세스의 제어에 대해 이해한다.
 - 자식 프로세스의 생성 과정 - `fork()` 시스템 호출을 통해
 - 프로세스의 오버레이 - `exec()` 시스템 호출을 통해
 - 자식 프로세스의 종료 대기 - `wait()` 시스템 호출을 통해
 - 프로세스의 종료 과정 - `exit()` 시스템 호출을 통해
 - 좀비 프로세스

2

1. 프로세스 개요

프로세스 개요

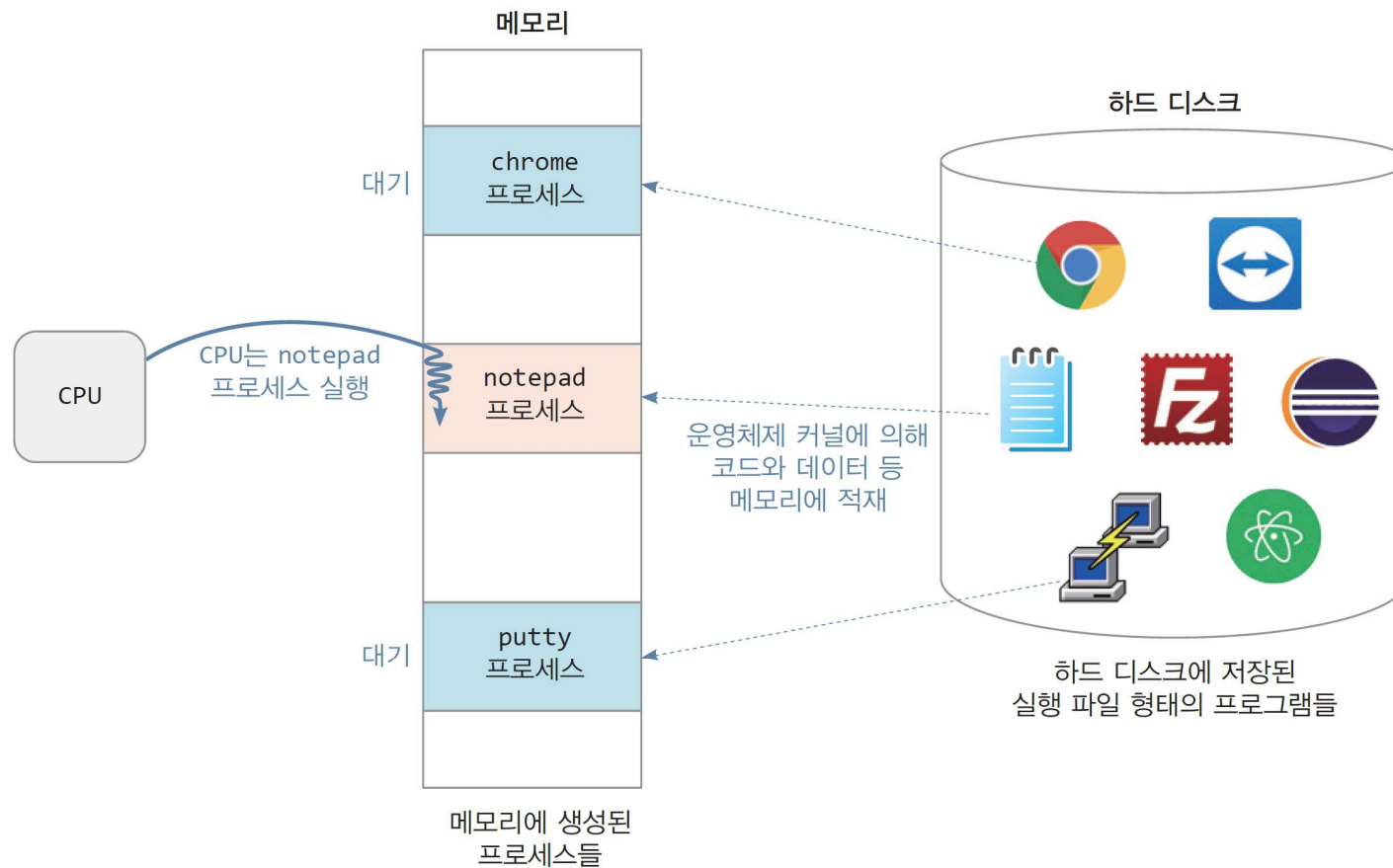
4

- 프로그램(program)
 - ▣ 하드디스크 등의 저장 매체에 저장. 실행 파일의 형태
- 프로세스(process)
 - ▣ 프로그램이 메모리에 적재되어 실행 중인 되는 상태
 - 필요한 모든 자원 할당 받음
 - 자원 : 코드 공간, 데이터 공간, 스택 공간, 힙 공간
 - ▣ 프로세스의 특징
 - 운영체제는 프로그램을 메모리 적재하고 프로세스로 다룸
 - 운영체제는 프로세스에게 실행에 필요한 메모리 할당, 이곳에 코드와 데이터 등 적재
 - 프로세스들은 서로 독립적인 메모리 공간을 가짐. 다른 프로세스의 영역에 접근 불허
 - 운영체제는 각 프로세스의 메모리 위치와 크기 정보를 관리한다.
 - 운영체제는 프로세스마다 고유한 번호(프로세스 ID) 할당
 - 프로세스의 관한 모든 정보는 커널에 의해 관리
 - 프로세스는 실행-대기-잠자기-대기-실행-종료 등의 생명 주기를 가짐
 - 프로세스 생성, 실행, 대기, 종료 등의 모든 관리는 커널에 의해 수행

프로그램과 프로세스

5

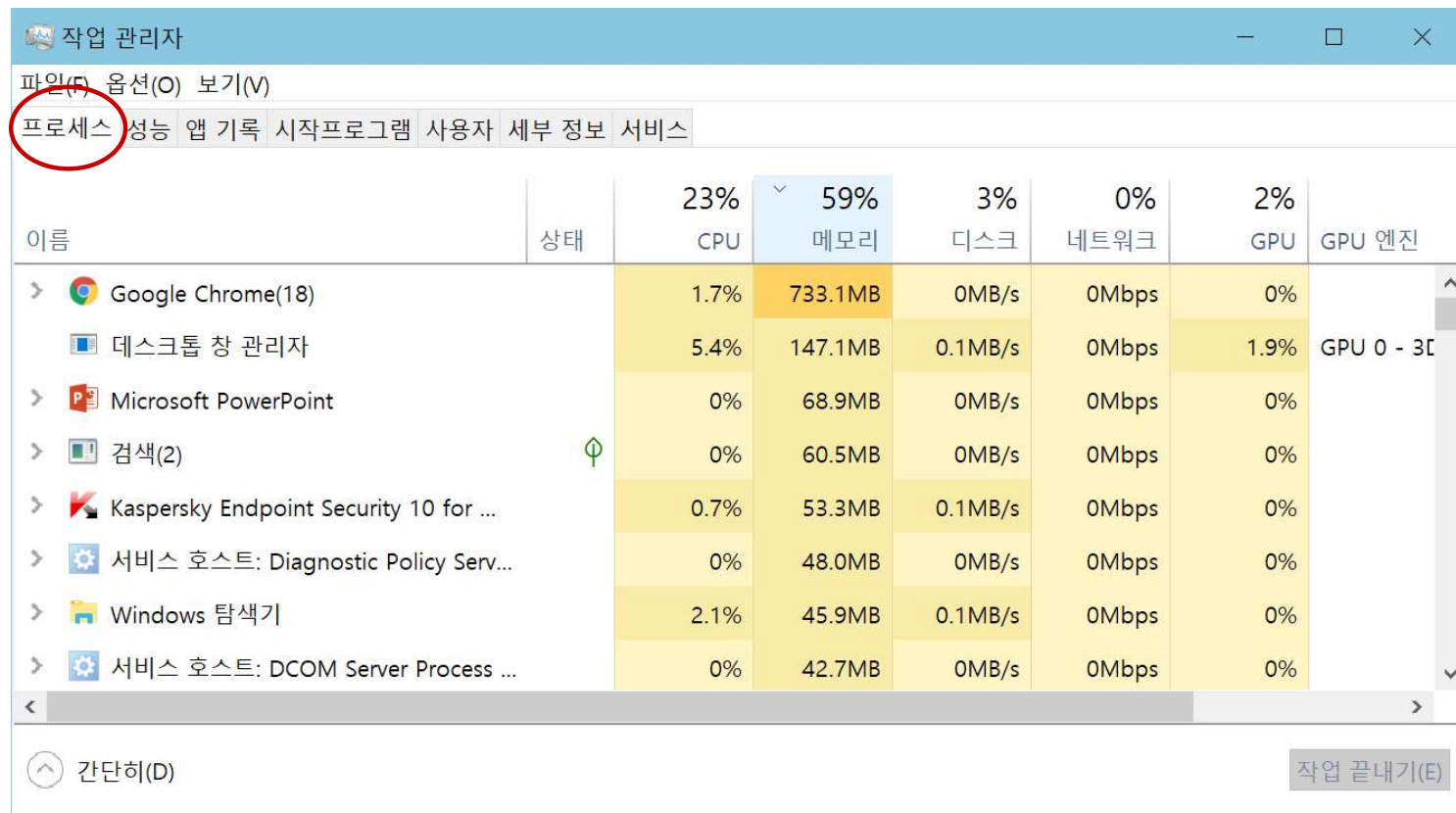
- 프로세스들은 상호 독립적인 메모리 공간에서 실행



프로세스 관리

6

- 프로세스의 생성에서 종료까지, 관리는 모두 커널에 의해 이루어짐
 - 커널 영역에 프로세스 테이블을 만들고, 프로세스들 목록 관리
- 관리 내용
 - 프로세스 생성, 실행, 일시 중단 및 재개, 정보 관리, 프로세스 통신, 프로세스 동기화, 프로세스 중단, 프로세스 컨텍스트 스위칭



이름	상태	23% CPU	59% 메모리	3% 디스크	0% 네트워크	2% GPU	GPU 엔진
Google Chrome(18)		1.7%	733.1MB	0MB/s	0Mbps	0%	
데스크톱 창 관리자		5.4%	147.1MB	0.1MB/s	0Mbps	1.9%	GPU 0 - 3D
Microsoft PowerPoint		0%	68.9MB	0MB/s	0Mbps	0%	
검색(2)		0%	60.5MB	0MB/s	0Mbps	0%	
Kaspersky Endpoint Security 10 for ...		0.7%	53.3MB	0.1MB/s	0Mbps	0%	
서비스 호스트: Diagnostic Policy Serv...		0%	48.0MB	0MB/s	0Mbps	0%	
Windows 탐색기		2.1%	45.9MB	0.1MB/s	0Mbps	0%	
서비스 호스트: DCOM Server Process ...		0%	42.7MB	0MB/s	0Mbps	0%	

리눅스에서 실행 중인 프로세스 목록 보기 사례

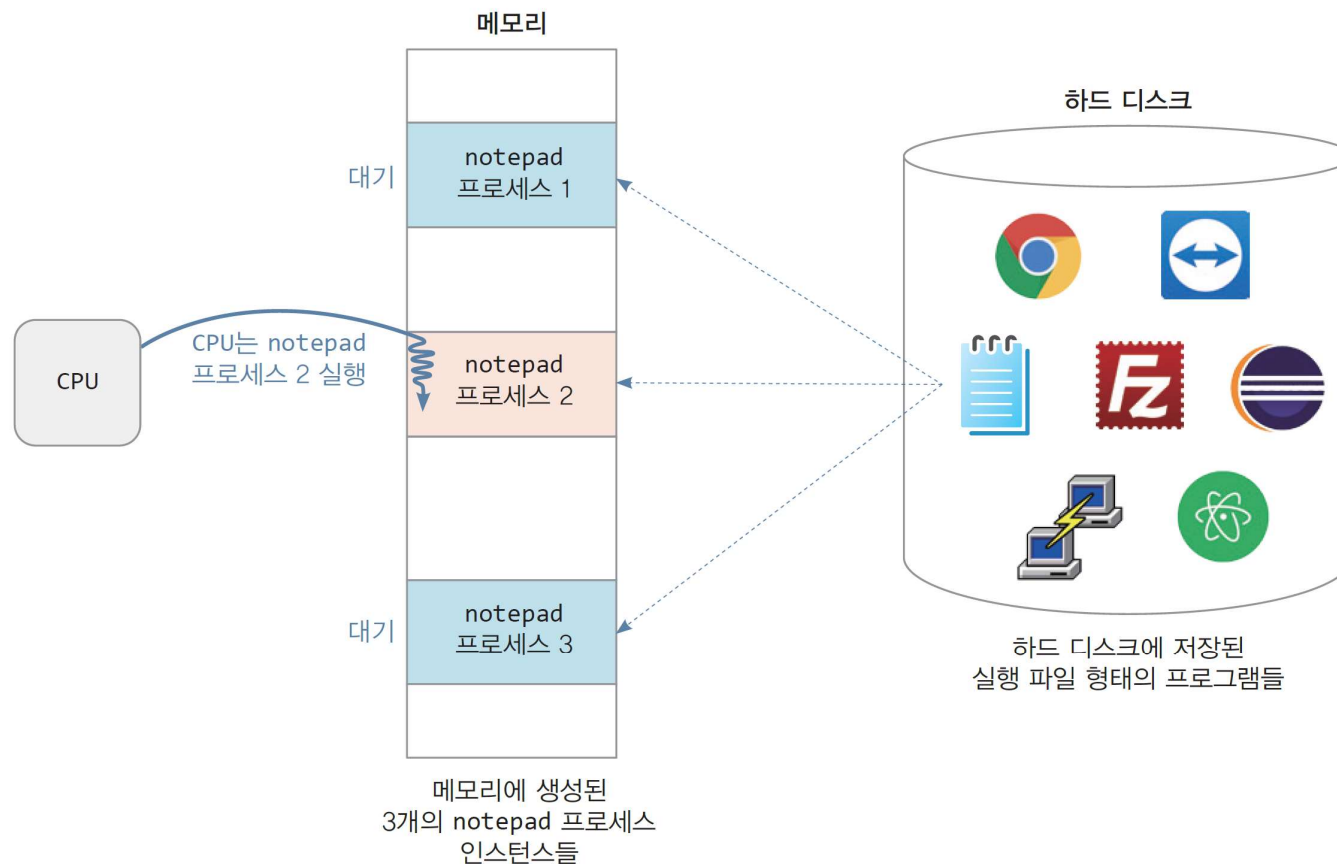
7

```
$ ps -ax
  PID TTY          STAT TIME  COMMAND
    1 ?           Ss    16:57 /sbin/init splash
    2 ?           S      0:01 [kthreadd]
    4 ?           I<     0:00 [kworker/0:0H]
    6 ?           I<     0:00 [mm_percpu_wq]
    7 ?           S      7:44 [ksoftirqd/0]
    8 ?           I   247:12 [rcu_sched]
    9 ?           I      0:00 [rcu_bh]
   10 ?           S      0:00 [migration/0]
   11 ?           S      0:22 [watchdog/0]
   12 ?           S      0:00 [cpuhp/0]
   13 ?           S      0:00 [cpuhp/1]
   14 ?           S      0:29 [watchdog/1]
   15 ?           S      0:00 [migration/1]
   16 ?           S   12:16 [ksoftirqd/1]
   18 ?           I<     0:00 [kworker/1:0H]
   19 ?           S      0:00 [cpuhp/2]
   20 ?           S      0:22 [watchdog/2]
.....
```

프로그램의 다중 인스턴스

8

- 한 프로그램을 여러 번 실행시키면 어떻게 될까?
 - ▣ 프로그램 실행 시마다 독립된 프로세스 생성 -> 프로세스들을 프로그램의 **다중 인스턴스**라고 부름
 - 각 프로세스에게 독립된 메모리 공간 할당
 - 각 프로세스를 별개의 프로세스로 취급



CPU 주소 공간

9

- CPU 주소 공간(CPU address space)
 - ▣ CPU가 주소선을 통해 액세스할 수 있는 전체 메모리 공간
 - ▣ 공간 크기
 - CPU 주소선의 수에 의해 결정
 - ▣ 32비트 CPU -> 32개의 주소선 -> 2^{32} 개의 주소 -> 2^{32} 바이트 -> 4GB 공간
 - 1번지의 저장 공간 크기는 1바이트
 - 주소 공간은 0번지부터 시작 ~
 - ▣ CPU 주소 공간보다 큰 메모리? 있어도 액세스 불가능
 - ▣ CPU 주소 공간보다 작은 량의 메모리? 가능
 - CPU가 설치된 메모리의 주소 영역을 넘어 액세스하면 시스템 오류
 - 예) 32비트 CPU를 가진 컴퓨터(4GB까지 메모리 액세스 가능)에 2GB의 메모리가 설치되어 있을 때 2GB를 넘어서 액세스하면 없는 메모리를 액세스하므로 심각한 오류 발생

프로세스 구성 - 4개의 메모리 영역

10

1. 코드(code) 영역

- 실행될 프로그램 코드가 적재되는 영역
 - 사용자가 작성한 모든 함수의 코드
 - 사용자가 호출한 라이브러리 함수들의 코드

2. 데이터(data) 영역

- 프로그램에서 고정적으로 만든 변수 공간
 - 전역 변수 공간, 정적 데이터 공간
 - 사용자 프로그램과 라이브러리 포함
- 프로세스 적재 시 할당, 종료 시 소멸

3. 힙(heap) 영역

- 프로세스의 실행 도중 동적으로 사용할 수 있도록 할당된 공간
 - `malloc()` 등으로 할당받는 공간은 힙 영역에서 할당
 - 힙 영역에서 아래 번지로 내려가면서 할당

4. 스택(stack) 영역

- 함수가 실행될 때 사용될 데이터를 위해 할당된 공간
 - 매개변수들, 지역변수들, 함수 종료 후 돌아갈 주소 등
 - 함수는 호출될 때, 스택 영역에서 위쪽으로 공간 할당
 - 함수가 `return`하면 할당된 공간 반환
- 함수 호출 외에 프로세스에서 필요시 사용 가능



프로세스 주소 공간

11

- 프로세스 주소 공간
 - ▣ 프로세스가 실행 중에 접근할 수 있도록 허용된 주소의 최대 범위
 - ▣ 프로세스 주소 공간은 **논리 공간(가상 공간)**
 - 0번지에서 시작하여 연속적인 주소
- 프로세스 주소 공간의 크기
 - ▣ CPU가 액세스할 수 있는 전체 크기
 - 32비트 CPU의 경우 4GB(윈도우, 리눅스 모두 동일)
 - 프로세스 주소 공간 크기는 프로세스의 현재 크기와 다름
 - ▣ 프로세스 주소 공간의 크기
 - 프로세스가 액세스할 수 있는 최대 크기(32비트 CPU 경우 4GB)
 - ▣ 프로세스의 크기
 - 적재된 코드 +
 - 전역 변수 +
 - 힙 영역에서 현재 할당받은 동적 메모리 공간 +
 - 스택 영역에 현재 저장된 데이터 크기

프로세스의 사용자 공간과 커널 공간

12

□ 프로세스 주소 공간 = 사용자 공간 + 커널 공간

▣ 사용자 공간

- 프로세스의 코드, 데이터, 힙, 스택 영역이 순서대로 할당되는 공간
- 코드와 데이터 영역의 크기는 프로세스 적재 시 결정,
- 힙은 데이터 영역 바로 다음부터 시작하고,
- 스택은 사용자 공간의 바닥에서 시작하여 거꾸로 자람
- 힙 영역은 높은 번지로 자라고, 스택은 낮은 번지로 자람
 - 예) 처음 `malloc(1000)`으로 동적 할당받는 공간은 데이터 영역 바로 다음의 힙 시작부분부터 할당
 - 예) 처음 함수가 호출될 때 할당되는 스택 공간은 스택 영역 바닥부터 위로 할당

▣ 커널 공간

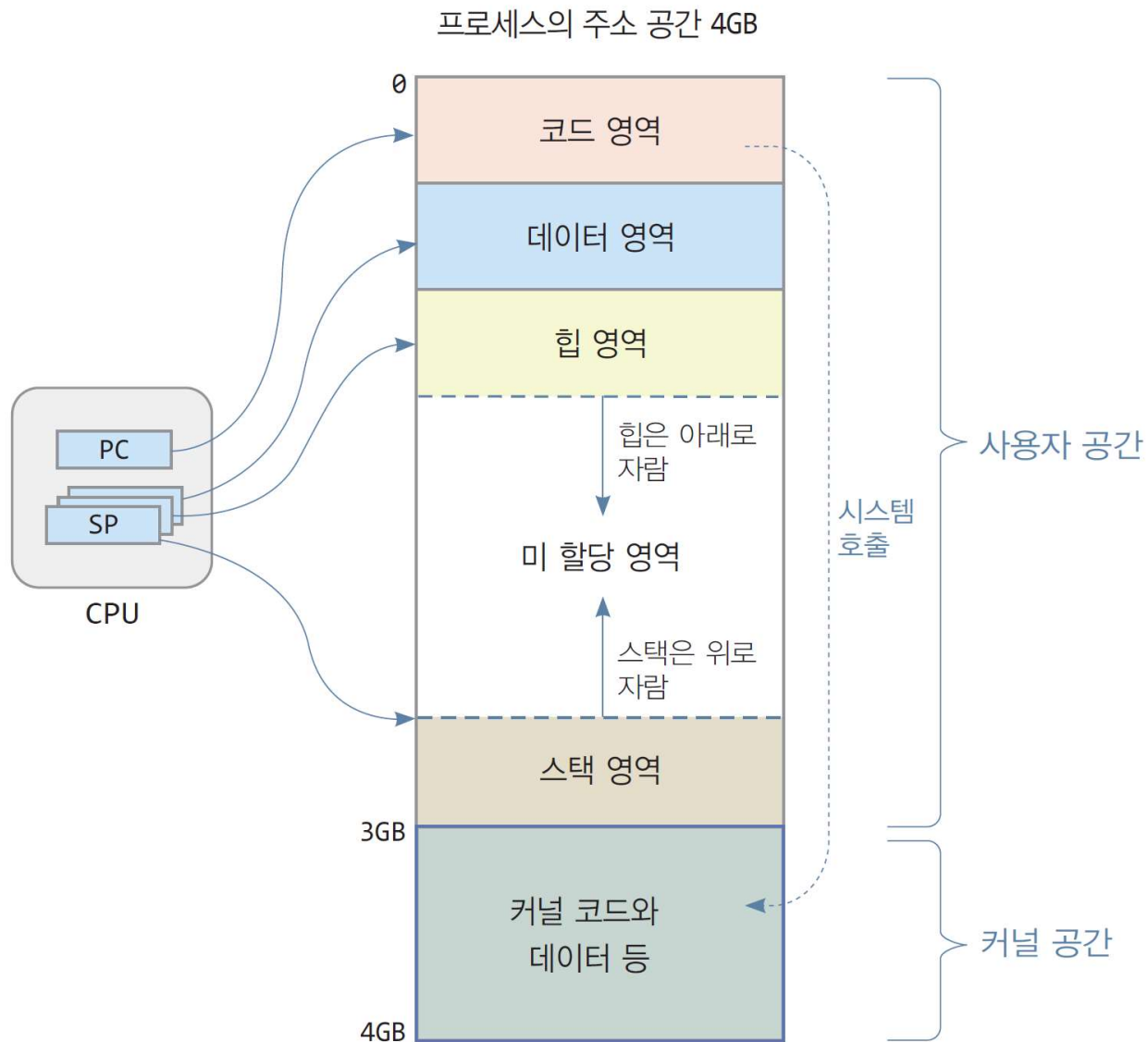
- 프로세스가 시스템 호출을 통해 이용하는 커널 공간
- 커널 코드, 커널 데이터, 커널 스택(커널 코드가 실행될 때)이 존재

□ 결론

- ▣ 프로세스의 코드와 데이터는 실행 파일에 결정된 상태로 코드 영역과 데이터 영역에 적재 -> 실행 중에 크기가 변하지 않음
- ▣ 프로세스는 사용자 공간의 최대 범위까지 동적할당 받으면서 힙 영역과 스택 영역을 늘려갈 수 있음

프로세스 주소 공간 사례 - 32비트 CPU에서 리눅스의 프로세스 주소 공간

13



커널 공간의 의미

14

- 각 프로세스는
 - ▣ 독립된 사용자 공간 소유
 - ▣ 커널 공간 공유
- 커널 공간
 - ▣ 프로세스가 사용자 코드에서 시스템 호출을 통해 커널 코드 실행할 때 커널 공간 사용
 - 커널 코드를 실행하고 있는 것은 사용자 프로세스
 - ‘사용자 프로세스가 커널 모드에서 실행되고 있다’고 표현
 - 커널 코드가 적재된 물리 메모리의 위치 역시 사용자 프로세스가 소유한 매핑 테이블 사용
 - 사용자 영역과 커널 영역을 하나의 가상 주소 영역으로 다룬다는 의미
- 사용자 공간과 커널 공간의 결론
 - ▣ 프로세스마다 각각 사용자 주소 공간이 있다.
 - ▣ 시스템 전체에는 하나의 커널 주소 공간이 있다.
 - ▣ 모든 프로세스는 커널 주소 공간을 공유한다.

프로세스의 주소 공간은 가상 주소 공간

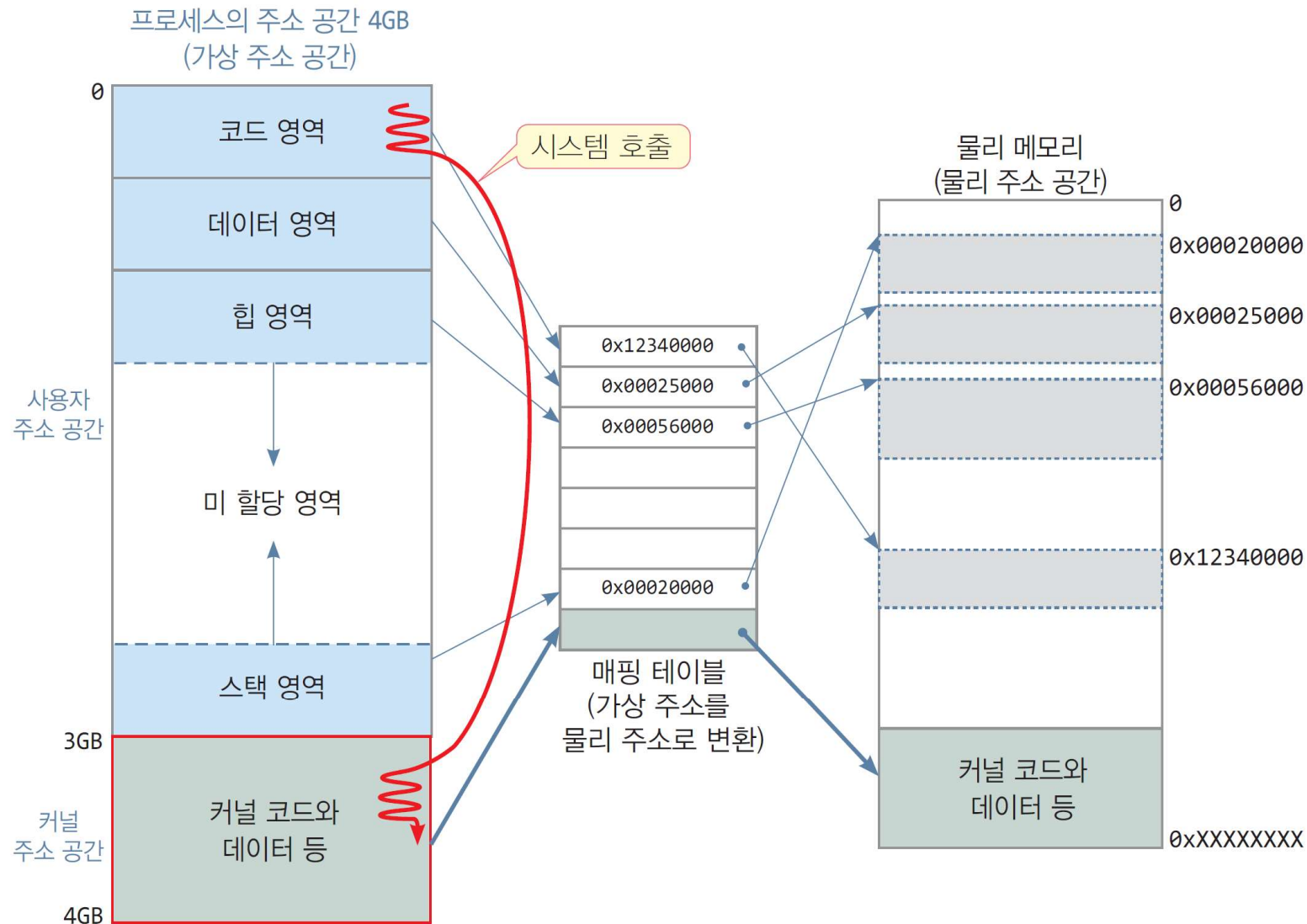
15

- 프로세스의 주소 공간은 **가상 공간**
 - ▣ 프로세스가 사용하는 주소는 **가상 주소**이다.
 - 프로세스에서 0번지는 가상 주소 0번지
 - 물리 메모리의 0번지 아님
 - 가상 주소는 0번지부터 시작
 - 프로세스 내의 코드 주소, 전역 변수에 대한 주소, malloc()에 의해 리턴된 주소, 스택에 담긴 지역 변수의 주소는 모두 가상 주소
 - ▣ 프로세스의 주소 공간(가상 주소 공간)은 사용자나 개발자가 보는 관점
 - 사용자나 개발자는 프로그램이 **0번지부터** 시작하여,
 - **연속적인** 메모리 공간에 형성되고,
 - 최대 크기의 메모리가 설치되어 있다고 상상
 - ▣ 실제 상황
 - 설치된 물리 메모리의 크기는 주소 공간보다 작을 수 있고,
 - 프로세스의 코드, 데이터, 힙, 스택은 물리 메모리에 흩어져 저장됨
 - 연속적인 메모리 공간이 아님

가상 주소 공간의 물리 메모리로의 매핑

16

- 사용자는 연속적인 공간(가상 주소 공간)으로 생각 -> 그러나 가상 주소의 데이터가 물리 메모리에 분산되어 있어 어느 물리 번지에 있을지 알 수 없음



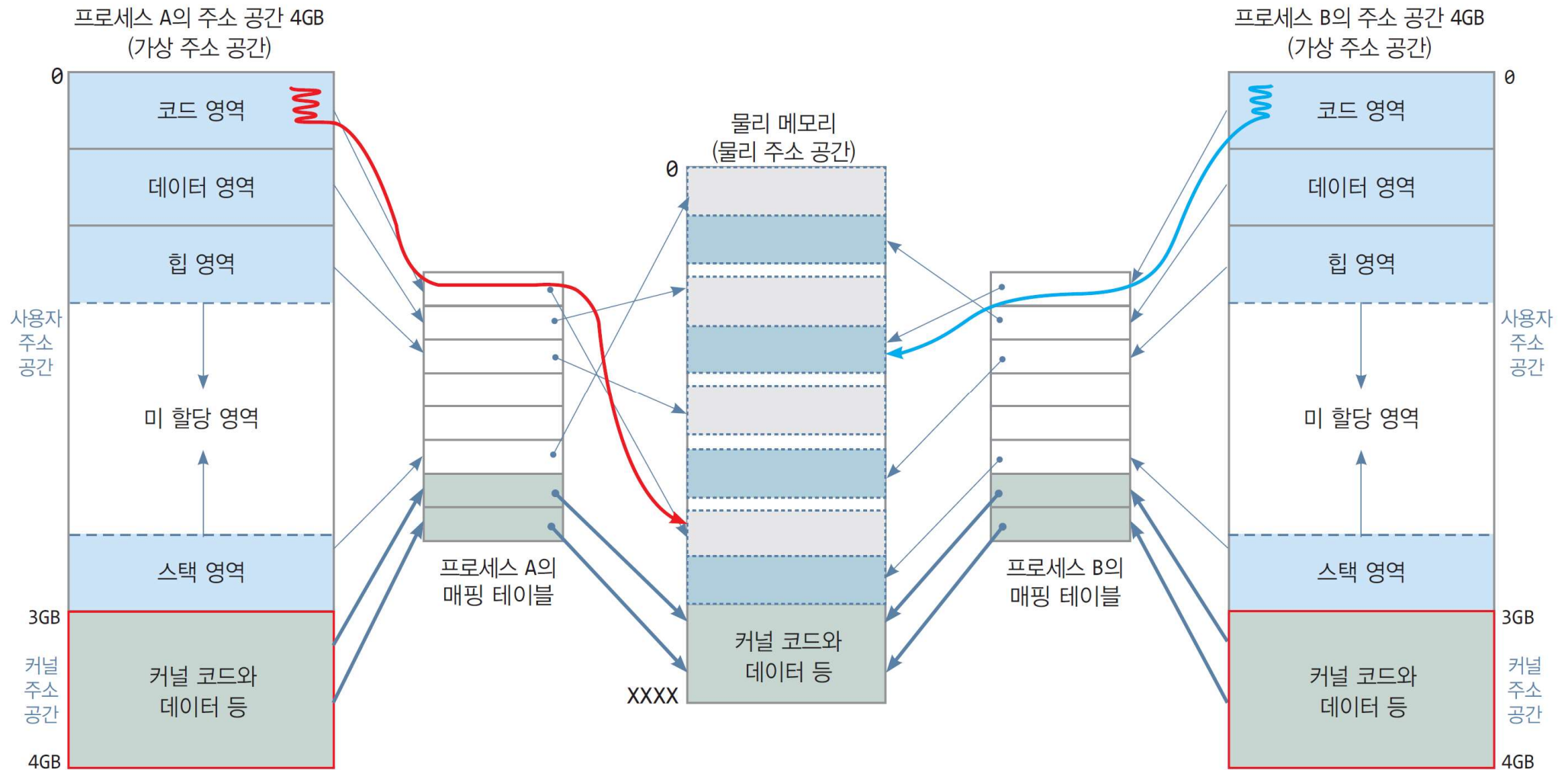
프로세스의 주소 공간은 가상 주소 공간

17

- 프로세스의 주소 공간은 **가상 공간**
 - ▣ 프로세스의 주소 공간은 사용자나 개발자가 보는 관점
 - 자신이 작성한 프로그램이 **0번지부터** 시작하여,
 - **연속적인** 메모리 공간에 형성되고,
 - CPU가 액세스할 수 있는 최대 크기의 메모리가 설치되어 있다고 상상
 - ▣ 실제 상황
 - 설치된 물리 메모리의 크기는 주소 공간보다 작을 수 있고,
 - 프로세스의 코드, 데이터, 힙, 스택은 물리 메모리에 흩어져 저장됨
 - 연속적인 메모리 공간이 아님
- 프로세스 주소 공간은 각 프로세스마다 주어지는가? 예
 - ▣ 프로세스마다 주소 공간은 별개이다.
- 그러면, 프로세스 주소 공간은 충돌하는가? 아니오
 - ▣ 프로세스 주소 공간은 가상 주소 공간이며,
 - ▣ 가상 주소가 물리 주소로 매핑되므로, 물리 메모리에서는 충돌하지 않는다.

프로세스들의 가상 주소 공간과 물리 메모리

18



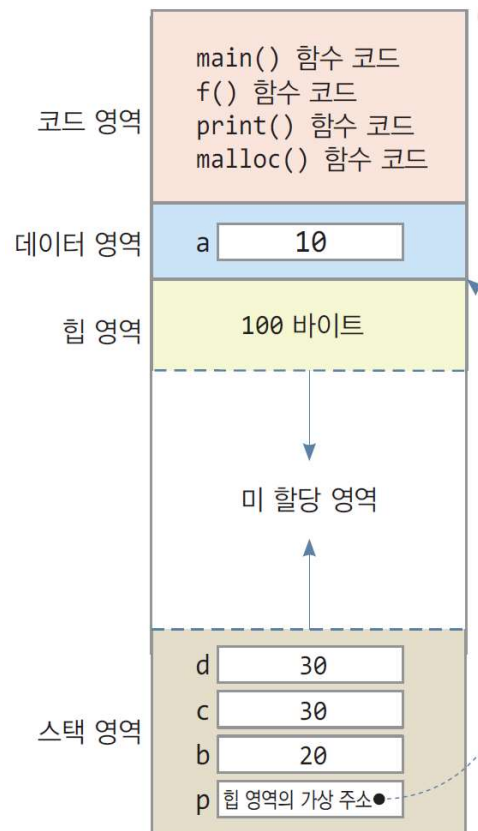
여러 프로세스들이 커널 공간 공유

탐구 3-1 프로세스의 구성 영역 그려보기

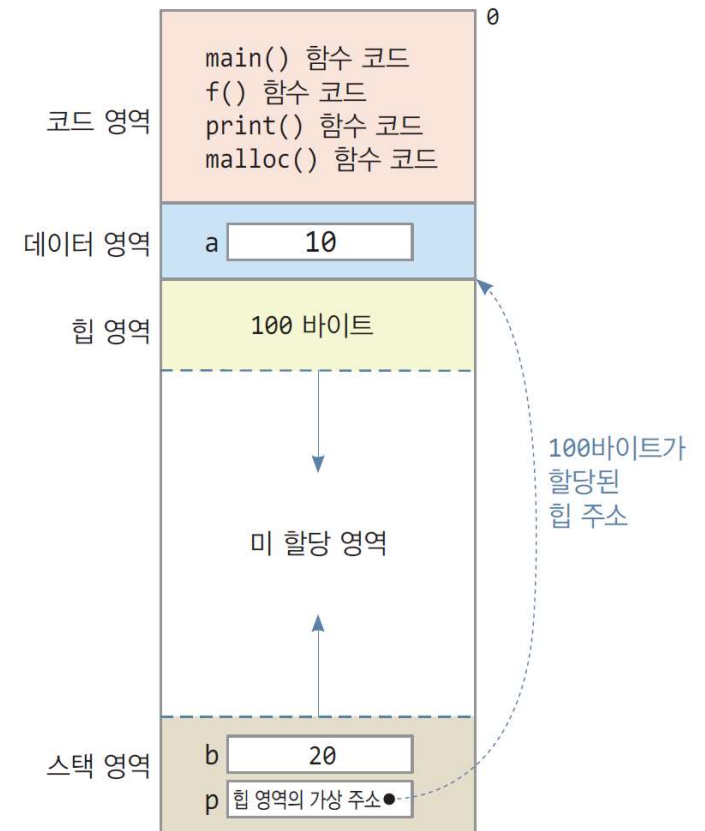
19

다음 프로그램이 적재된 프로세스가 실행을 시작하여 main() 함수 내에 printf()를 호출하기 직전, 프로세스의 사용자 주소 공간을 그려보라.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a=10;
5 void f(int c) {
6     int d=c;
7     printf("%d", d);
8 }
9 int main() {
10     int b=20;
11     int* p = (int*)malloc(100);
12     f(30);
13     printf("%d", b);
14     return 0;
15 }
```



(a) 라인 7이 끝난 시점



(b) 라인 12가 끝난 시점

20

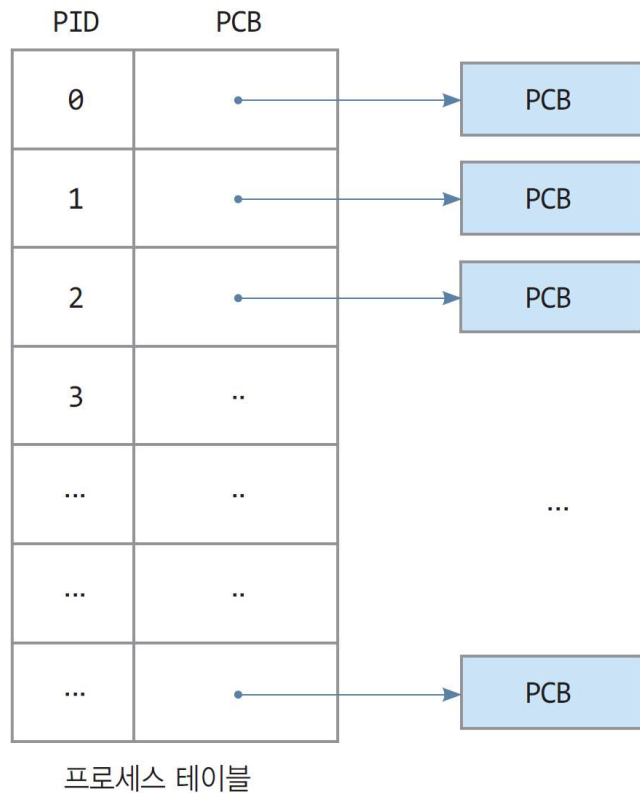
2. 커널의 프로세스 관리

프로세스 테이블과 프로세스 제어 블록

21

- 프로세스 테이블(Process Table)
 - ▣ 시스템의 모든 프로세스들을 관리하기 위한 표
 - ▣ 시스템에 한 개만 있음
 - ▣ 구현 방식은 운영체제마다 다름
- 프로세스 제어 블록(Process Control Block, PCB)
 - ▣ 프로세스에 관한 정보를 저장하는 구조체
 - ▣ 프로세스당 하나씩 존재
 - ▣ 프로세스가 생성될 때 만들어지고 종료되면 삭제
 - ▣ 커널에 의해 생성, 저장, 읽혀지는 등 관리
- 프로세스 테이블과 프로세스 제어 블록의 위치
 - ▣ 커널 영역, 커널 코드(커널 모드)만이 액세스 가능

프로세스 테이블과 프로세스 제어 블록(PCB)



프로세스 번호(PID)

부모프로세스 번호(PPID)

프로세스 상태(process state)

프로세스 컨텍스트
(PC, SP, 범용 레지스터 등 CPU 레지스터들)

스케줄링 정보
(priority, nice 값, 프로세스가 사용한 CPU 시간 등)

프로세스의 종료 코드(exit code)

프로세스의 오픈 파일 테이블

메모리 관리를 위한 정보들
(페이지 테이블 주소 등)

프로세스사이의 통신 정보들

회계 정보(accounting info)

프로세스 소유자 이름(user name)

기타 프로세스가 사용중인 입출력 장치 목록 등

프로세스 제어 블록(PCB)에 저장되는 정보

23

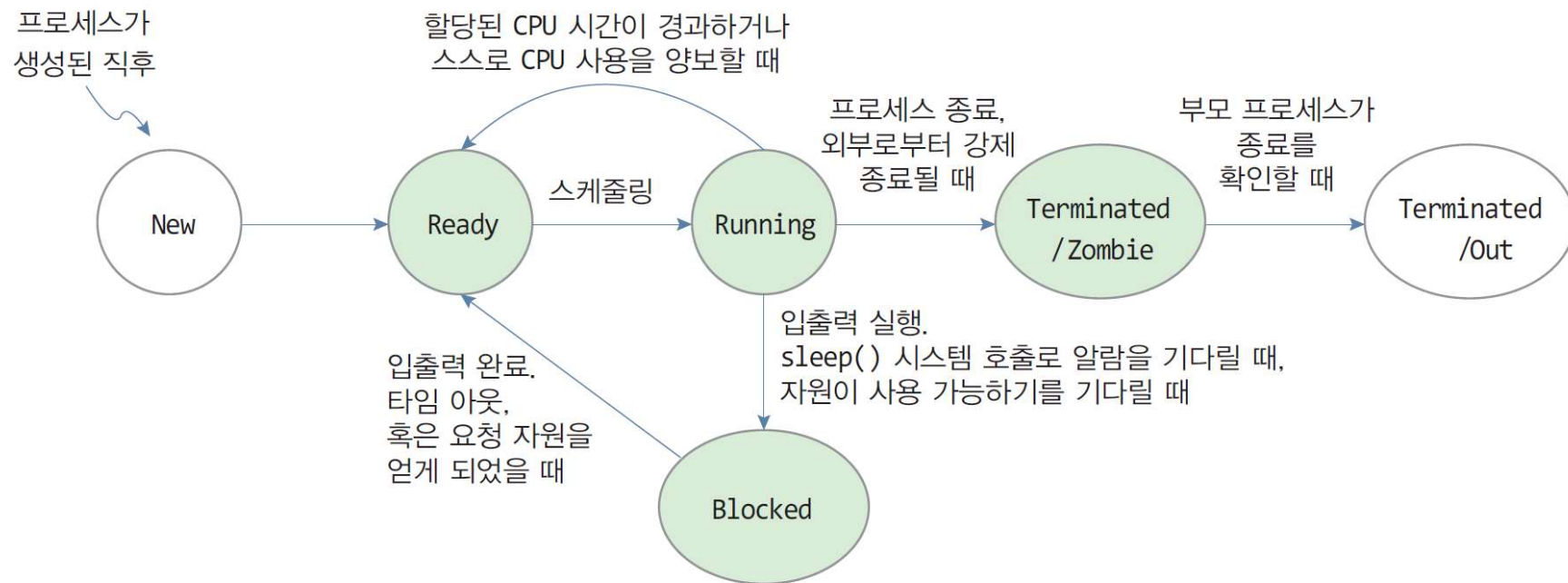
1. 프로세스 번호(PID, Process ID) : 0과 양의 정수, 유일한 번호, 이 번호로 프로세스 구분
2. 부모 프로세스 번호(PPID, Parent Process ID) : 부모 프로세스의 PID
3. 프로세스 상태(Process State) 정보 : 준비, 실행 중, 블록(입출력 완료대기) 등
4. CPU 컨텍스트 정보
 - ▣ PC(Program Counter) : 프로세스가 선택되면 실행을 시작할 프로세스 내 코드 주소
 - 사용자 모드에 있었던 경우, 사용자 공간의 코드 주소
 - 커널 모드에 있었던 경우, 커널 공간의 코드 주소
 - ▣ SP(Stack Pointer)
 - ▣ 기타 레지스터
5. 스케줄링 정보
 - ▣ 우선 순위 값, nice 값, 스케줄 큐에 대한 포인터 등
6. 프로세스 종료 코드(정수 0~255)
 - ▣ 프로세스가 종료할 때 남기는 정수 값. exit() 시스템 호출의 매개변수값. main() 함수의 return리턴 값. 부모 프로세스에게 전달
7. 프로세스의 오픈 파일 테이블 : 열어놓은 파일 디스크립터들이 저장되는 배열(11장 참고)
8. 메모리 관리 정보
 - ▣ 페이지 테이블의 물리 메모리 주소 등
9. 프로세스 사이의 통신 정보들 : 부록A에서 다룸
10. 회계 정보
 - ▣ CPU의 사용 시간, 제한 시간, 프로세스의 총 경과시간 등, 사용료 계산이나 성능 통계에 사용
11. 프로세스 소유자 정보
 - ▣ 프로세스를 생성한 사용자의 로그인 이름이나 사용자 ID 등

* 운영체제마다 프로세스 제어 블록에 저장되는 요소와 프로세스 상태 등이 다름

프로세스 생명 주기와 상태 변이(state change)

□ 프로세스의 생명 주기

- ▣ 프로세스는 탄생에서 종료까지 여러 상태로 바뀌면서 실행
- ▣ 상태 정보는 PCB에 기록되고, 상태가 바뀔 때마다 갱신됨



프로세스의 상태

25

- New(생성 상태)
 - ▣ 프로세스가 생성된 상태. 메모리 할당 및 필요한 자원이 적재된 상태, PCB에 New 상태로 등록. 실행 준비를 마치면 Ready 상태로 바뀜
- Ready(준비 상태)
 - ▣ 프로세스가 스케줄링을 기다리는 '준비 상태'
 - ▣ 프로세스는 준비 큐에서 대기
 - ▣ 스케줄링되면 Running 상태로 되고 CPU에 의해 실행됨
- Running(실행 상태)
 - ▣ 프로세스가 CPU에 의해 현재 실행되고 있는 상태
 - ▣ CPU의 시간할당량(타임슬라이스)이 지나면 다시 Ready 상태로 바뀌고 준비 큐에 삽입
 - ▣ 프로세스가 입출력을 시행하면 커널은 프로세스를 Blocked 상태로 만들고 대기 큐에 삽입
- Blocked/Wait(블록 상태)
 - ▣ 프로세스가 자원을 요청하거나, 입출력을 요청하고(예: read() 시스템 호출) 완료를 기다리는 상태
 - ▣ 입출력이 완료되면 프로세스는 Ready 상태로 바뀌고 준비 큐에 삽입
- Terminated/Zombie 상태
 - ▣ 프로세스가 불완전 종료된 상태(좀비 상태)
 - 프로세스가 차지하고 있던 메모리와 할당받았던 자원들을 모두 커널에 의해 반환됨. 커널에 의해 열어 놓은 파일도 닫힘
 - 하지만, 프로세스 테이블의 항목과 PCB가 여전히 시스템에서 제거되지 않은 상태
 - 프로세스가 남긴 종료 코드(PCB에 있음)를 부모 프로세스가 읽어가지 않아 완전히 종료되지 않은 상태 – 좀비 상태라고 부름
- Terminated/Out 상태
 - ▣ 프로세스가 종료하면서 남긴 종료 코드(PCB에 있음)를 부모 프로세스가 읽어 가서 완전히 종료된 상태
 - ▣ 프로세스 테이블의 항목과 PCB가 시스템에서 완전히 제거된 상태

프로세스 스케줄링과 컨텍스트 스위칭

26

- 프로세스 스케줄링과 스레드 스케줄링
 - ▣ 프로세스 스케줄링
 - 과거 운영체제에서 실행 단위는 프로세스 였음
 - Ready 상태의 프로세스 중에 실행 시킬 프로세스 선택
 - ▣ 오늘날 운영체제는 스레드를 대상으로 스케줄링
 - 오늘날 프로세스 스케줄링은 없음
 - 오늘날 운영체제에서 실행 단위는 스레드
 - Ready 상태의 스레드 중 실행시킬 스레드 선택
- 그럼 프로세스는 뭐지?
 - ▣ 프로세스는 스레드들에게 공유 자원을 제공하는 컨테이너로 역할이 바뀌었음

프로세스 정보 보기

27

프로세스 테이블과 PCB에서 액세스

사례 - 리눅스 셸 명령으로 프로세스 정보 보기

프로세스를 소유한 사용자 ID 프로세스 번호 부모 프로세스 번호 프로세스의 우선순위 nice 값

```
$ ps -eal
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	1	0	0	80	0	-	29933	-	?	00:00:38	systemd
1	S	0	2	0	0	80	0	-	0	-	?	00:00:00	kthreadd
1	I	0	4	2	0	60	-20	-	0	-	?	00:00:00	kworker/0:0H
1	I	0	6	2	0	60	-20	-	0	-	?	00:00:00	mm_percpu_wq
1	S	0	7	2	0	80	0	-	0	-	?	00:03:27	ksoftirqd/0
1	I	0	8	2	0	80	0	-	0	-	?	01:20:16	rcu_sched
1	I	0	9	2	0	80	0	-	0	-	?	00:00:00	rcu_bh
1	S	0	10	2	0	-40	-	-	0	-	?	00:00:00	migration/0
5	S	0	11	2	0	-40	-	-	0	-	?	00:00:08	watchdog/0
1	S	0	12	2	0	80	0	-	0	-	?	00:00:00	cpuhp/0
1	S	0	13	2	0	80	0	-	0	-	?	00:00:00	cpuhp/1
5	S	0	14	2	0	-40	-	-	0	-	?	00:00:09	watchdog/1
1	S	0	15	2	0	-40	-	-	0	-	?	00:00:00	migration/1
1	S	0	16	2	0	80	0	-	0	-	?	00:04:32	ksoftirqd/1
1	I	0	18	2	0	60	-20	-	0	-	?	00:00:00	kworker/1:0H
1	S	0	19	2	0	80	0	-	0	-	?	00:00:00	cpuhp/2
5	S	0	20	2	0	-40	-	-	0	-	?	00:00:08	watchdog/2
.....													

로그인 이름 상태 정보

```
$ ps -U kitae -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
kitae	18150	0.0	0.0	22468	3712	pts/2	S	11:51	0:00	bash
kitae	18157	0.0	0.0	38644	3320	pts/2	R+	11:52	0:00	ps -U kitae -u

탐구 3-2 C 프로그램으로 프로세스와 부모 프로세스 번호 알아내기

28

getpid(), getppid() 시스템 호출 함수를 이용하여 프로세스 번호와 부모 프로세스 번호를 얻어내는 C 프로그램 pinfo.c를 작성하라.

pinfo.c

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

int main() {
    pid_t pid, ppid;
    pid = getpid(); // 현재 프로세스의 PID 알아내기
    ppid = getppid(); // 부모 프로세스의 PID 알아내기
    printf( " 프로세스 PID = %d, 부모 프로세스 PID = %d\n", pid, ppid);
}
```

```
$ gcc -o pinfo pinfo.c
$ ./pinfo
프로세스 PID = 31006, 부모 프로세스 PID = 30861
$
```

3. 프로세스 계층 구조

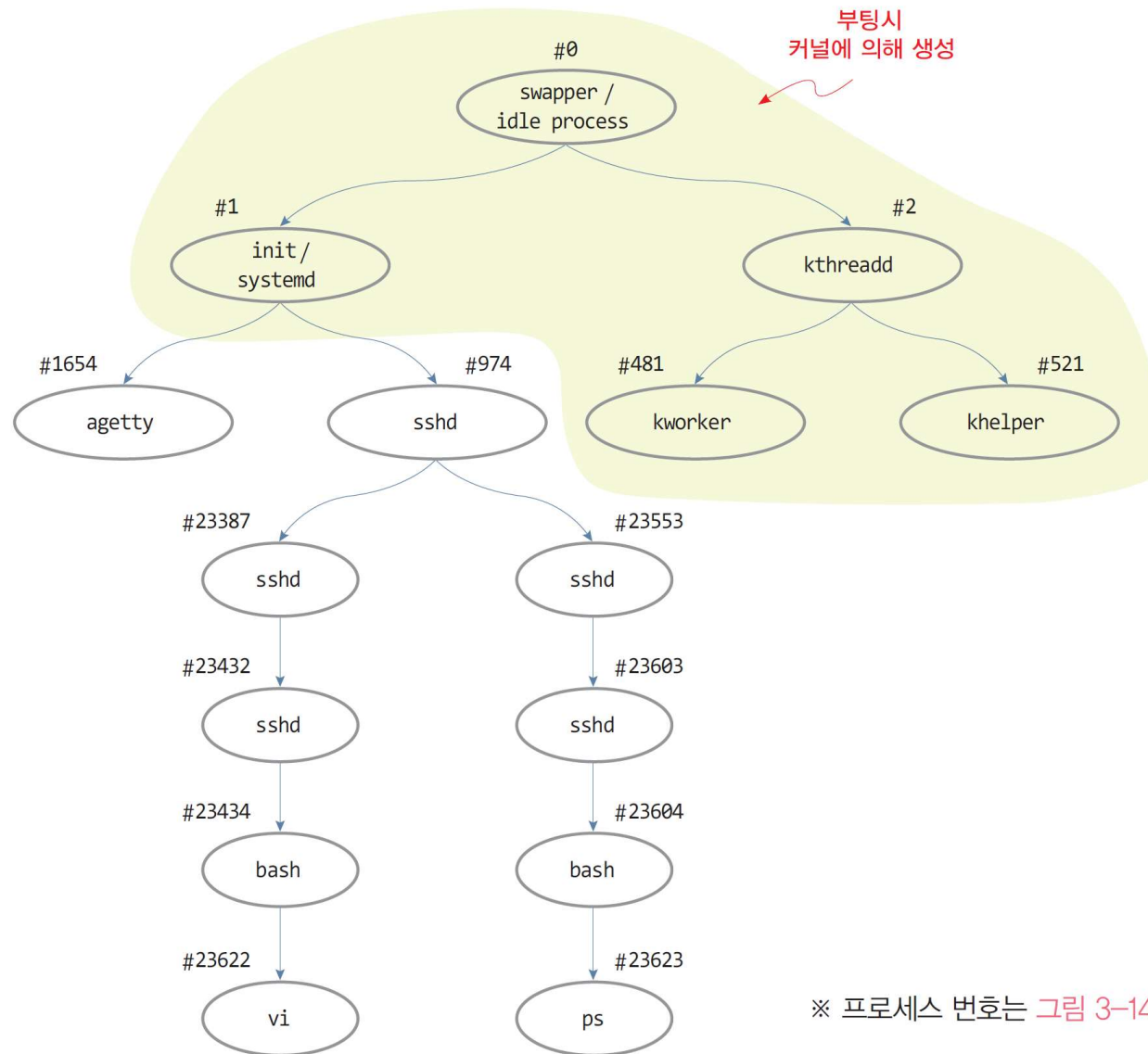
프로세스의 부모-자식 관계

30

- 프로세스는 일반적으로 부모-자식 관계
 - 윈도우에서 프로세스는 모두 동등 - 계층 관계 아님
 - ▣ #0 프로세스가 시스템 부팅 시 실행되는 최초의 프로세스, 조상 프로세스
 - ▣ 부모 프로세스는 여러 개의 자식 프로세스를 가질 수 있음
 - ▣ 모든 프로세스는 부모 프로세스를 가짐(#0 프로세스 제외)
- 자식 프로세스의 생성
 - ▣ 모든 프로세스는 프로세스(부모)에 의해 생성
 - 프로세스 생성은 시스템 호출을 통해서만 가능
 - `fork()`, `clone()` 등의 커널 코드가 자식 프로세스 생성
 - 예외 : `PID 0, 1, 2` 등의 몇몇 조상 프로세스는 시스템 호출이 아닌 수작업(*hand-craft*)으로 생성
- 리눅스 사례
 - ▣ #0 프로세스 – swapper/idle 프로세스(*hand-crafted*)
 - ▣ #1 프로세스 – init 프로세스 (*hand-crafted*)
 - 부팅 후 생성되는 모든 사용자 프로세스의 조상
 - ▣ #2 프로세스 – kthreadd 프로세스 (*hand-crafted*)
 - 커널 모드에서 커널 코드로만 실행되는 모든 커널 프로세스(thread)의 조상

부모-자식의 계층 관계로 구성되는 리눅스의 프로세스들 사례

31



※ 프로세스 번호는 그림 3-14 참고

리눅스의 프로세스 목록(부모-자식 프로세스 확인)

```
$ ps -eal
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   0    1    0    0  80   0 - 29971 -      ?        00:00:19 systemd
1 S   0    2    0    0  80   0 -      0 -      ?        00:00:00 kthreadd
1 I   0    4    2    0  60 -20 -      0 -      ?        00:00:00 kworker/0:0H
1 I   0    6    2    0  60 -20 -      0 -      ?        00:00:00 mm_percpu_wq
1 S   0    7    2    0  80   0 -      0 -      ?        00:03:06 ksoftirqd/0
1 I   0    8    2    0  80   0 -      0 -      ?        01:32:55 rcu_sched
1 I   0    9    2    0  80   0 -      0 -      ?        00:00:00 rcu_bh
1 S   0   10    2    0 -40   - -      0 -      ?        00:00:00 migration/0
5 S   0   11    2    0 -40   - -      0 -      ?        00:00:09 watchdog/0
.....
1 S   0   199    2    0  80   0 -      0 -      ?        00:31:14 usb-storage
1 I   0   202    2    0  60 -20 -      0 -      ?        00:22:29 kworker/2:1H
1 I   0   205    2    0  60 -20 -      0 -      ?        00:00:00 ttm_swap
1 I   0   206    2    0  60 -20 -      0 -      ?        00:22:24 kworker/0:1H
.....
4 S   0   963    1    0  80   0 -   6012 -      ?        00:00:00 vsftpd
4 S   0   974    1    0  80   0 -  16378 -      ?        00:00:00 sshd
0 S  108  1251    1    0  80   0 - 130908 -      ?        00:00:00 notify-osd
4 S  109  1645    1    0  80   0 -   93497 -      ?        00:00:00 whoopsie
4 S   0  1654    1    0  80   0 -   4304 -      tty1     00:00:00 agetty
5 S   0  1672    1    0  80   0 -   3764 -      ?        00:00:00 xinetd
4 S   0 23378   974    0  80   0 -   23732 -      ?        00:00:00 sshd
5 S  1000 23432 23378    0  80   0 -   23732 -      ?        00:00:00 sshd
0 S  1000 23434 23432    0  80   0 -    5934 wait    pts/8    00:00:00 bash
.....
4 S   0 23553   974    0  80   0 -   23732 -      ?        00:00:00 sshd
5 S  1000 23603 23553    0  80   0 -   23732 -      ?        00:00:00 sshd
0 S  1000 23604 23603    0  80   0 -    5934 wait    pts/9    00:00:00 bash
0 S  1000 23622 23434    0  80   0 -    8361 poll_s   pts/8    00:00:00 vi
0 R  1000 23623 23604    0  80   0 -    7549 -      pts/9    00:00:00 ps
$
```


탐구 3-3 리눅스에서 실행 중인 프로세스들의 계층 구조 보기

\$ pstree 0 명령으로 프로세스 트리 보기

```
$ pstree 0
?--kthreadd--acpi_thermal_pm
|--ata_sff
|--charger_manager
|--cpuhp/0
|--cpuhp/1
|--cpuhp/2
|--cpuhp/3
|--crypto
|--devfreq_wq
.....
|--writeback
|--xfs_mru_cache
|--xfsalloc
|--systemd--ModemManager--{gdbus}
|--{gmain}
|--NetworkManager--dnsmasq
|--{gdbus}
|--{gmain}
|--accounts-daemon--{gdbus}
|--{gmain}
|--acpid
|--agetty
|--avahi-daemon--avahi-daemon
.....
|--irqbalance
|--lightdm--Xorg--{InputThread}
|--lightdm--lightdm--upstart--at-spi-bus-laun--dbus-daemon
|--{dconf worker}
|--{gdbus}
|--{gmain}
|--at-spi2-registr--{gdbus}
|--{gmain}
|--bamfdaemon--{dconf worker}
|--{gdbus}
|--{gmain}
.....
|--rsyslogd--{in:imklog}
|--{in:imuxsock}
|--{rs:main Q:Reg}
|--rtkit-daemon--2*[{rtkit-daemon}]
|--sshd--sshd--sshd--bash--vi
|--sshd--sshd--bash--pstree
|--systemd--(sd-pam)
|--systemd-journal
|--systemd-logind
|--systemd-timesyn--{sd-resolve}
|--systemd-udev
.....
--xinetd
```

탐구 3-4 Windows에서 부모 프로세스와 자식 프로세스 보기

Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-D27F732Wuser]

File Options View Process Find Users Help

Process	CPU	Private Bytes	Working Set	PID	Description
Registry		2,760 K	20,412 K	96	
System Idle Process	60.69	52 K	8 K	0	
System	1.28	212 K	4,380 K	4	
csrss.exe		2,032 K	1,952 K	776	
wininit.exe		1,568 K	1,304 K	928	
csrss.exe	0.53	21,432 K	3,440 K	1012	
winlogon.exe		2,912 K	1,932 K	1168	
explorer.exe	6.12	371,912 K	191,612 K	9232	Windows 탐색기
MSASCuiL.exe	< 0.01	1,984 K	2,160 K	16204	Windows Defender no
Bootcamp.exe	0.04	2,940 K	3,848 K	16372	Boot Camp Manager
OneDrive.exe	0.03	25,304 K	16,364 K	17144	Microsoft OneDrive
CrossEXService.exe	0.06	2,428 K	3,340 K	16348	CrossEX Service
MagicLine4NX.exe	0.04	23,428 K	4,724 K	17804	MagicLine4NX
TeamViewer.exe	0.85	86,024 K	48,880 K	15132	TeamViewer
POWERPNT.EXE	< 0.01	88,968 K	3,616 K	15804	Microsoft PowerPoint
POWERPNT.EXE	< 0.01	130,244 K	4,568 K	8328	Microsoft PowerPoint
Zoom.exe	0.52	55,068 K	89,160 K	13952	Zoom Meetings
Zoom.exe	0.25	125,460 K	57,904 K	8128	Zoom Meetings
Hwp.exe	0.28	140,436 K	52,380 K	9064	HWP
HimTrayIcon.exe	0.01	1,668 K	8,156 K	27384	Hancom HimTrayIcon
POWERPNT.EXE	0.09	136,520 K	144,740 K	24284	Microsoft PowerPoint
chrome.exe	0.04	85,704 K	151,468 K	22600	Google Chrome
chrome.exe					

CPU Usage: 39.31% Commit Charge: 71.06% Processes: 246 Physical Usage: 66.06%

Hwp.exe:9064 Properties

TCP/IP Security Environment Job Strings
Image Performance Performance Graph GPU Graph Threads

Image File

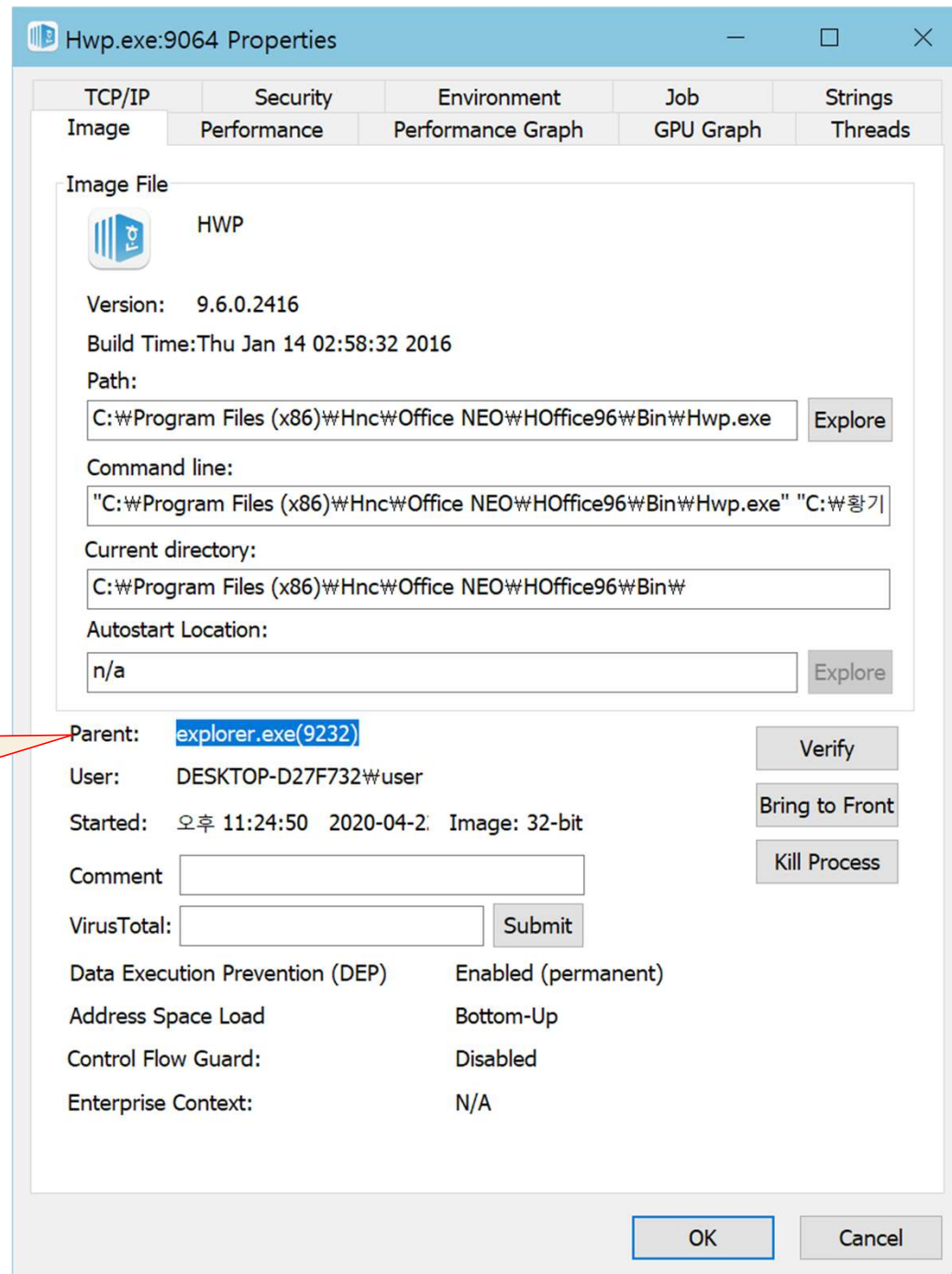
HWP

Version: 9.6.0.2416
Build Time: Thu Jan 14 02:58:32 2016
Path: C:\Program Files (x86)\Hnc\Office NEO\HOffice96\Bin\Hwp.exe
Command line: "C:\Program Files (x86)\Hnc\Office NEO\HOffice96\Bin\Hwp.exe" "C:\황기
Current directory: C:\Program Files (x86)\Hnc\Office NEO\HOffice96\Bin\W
Autostart Location: n/a
Parent: explorer.exe(9232)
User: DESKTOP-D27F732Wuser
Started: 오후 11:24:50 2020-04-2 Image: 32-bit
Comment
VirusTotal: Submit
Data Execution Prevention (DEP) Enabled (permanent)
Address Space Load Bottom-Up
Control Flow Guard: Disabled
Enterprise Context: N/A

OK Cancel

Process Explorer 프로그램 설치 후 실행

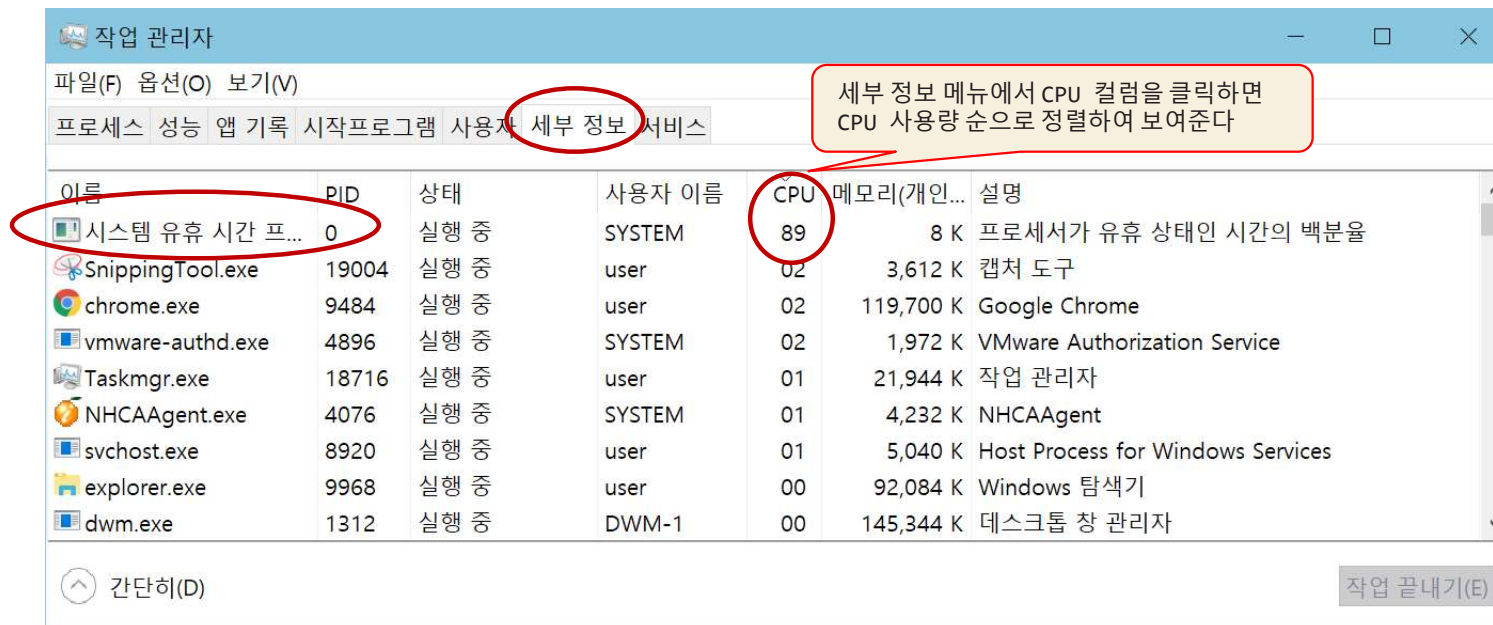
Hwp.exe 프로세스의 부모가 #9232
인 explorer.exe임을 확인할 수 있음



#0과 #1 프로세스: idle 프로세스와 init 프로세스

36

- #0 프로세스
 - ▣ 최고의 어른(조상) 프로세스
 - ▣ Unix의 #0 프로세스
 - swapper라고 불림, 부팅을 담당하고 #1 프로세스 생성
 - ▣ 리눅스의 #0 프로세스
 - idle 프로세스, 부팅 관여 없이 아무 일도 하지 않고 루프
 - 우선 순위가 가장 낮은 프로세스, 다른 프로세스가 있으면 실행될 일 없음
 - 실행중인 프로세스가 1개도 없는 상태에 빠지지 않기 위해 만든 프로세스
 - Unix 시절의 관례에 따라 0번 프로세스이므로 swapper라고도 부름
 - ▣ Windows의 #0 프로세스 : system idle process(시스템 유휴 프로세스)
 - 아무 일도 하지 않고 루프를 도는 단순 프로세스
 - 그림의 사례 - CPU 시간의 89% 소모. 사용자가 컴퓨터를 사용하고 있지 않은 시간 동안 실행



이름	PID	상태	사용자 이름	CPU	메모리(개인...)	설명
시스템 유휴 시간 프...	0	실행 중	SYSTEM	89	8 K	프로세서가 유휴 상태인 시간의 백분율
SnippingTool.exe	19004	실행 중	user	02	3,612 K	캡처 도구
chrome.exe	9484	실행 중	user	02	119,700 K	Google Chrome
vmware-authd.exe	4896	실행 중	SYSTEM	02	1,972 K	VMware Authorization Service
Taskmgr.exe	18716	실행 중	user	01	21,944 K	작업 관리자
NHCAAgent.exe	4076	실행 중	SYSTEM	01	4,232 K	NHCAAgent
svchost.exe	8920	실행 중	user	01	5,040 K	Host Process for Windows Services
explorer.exe	9968	실행 중	user	00	92,084 K	Windows 탐색기
dwm.exe	1312	실행 중	DWM-1	00	145,344 K	데스크톱 창 관리자

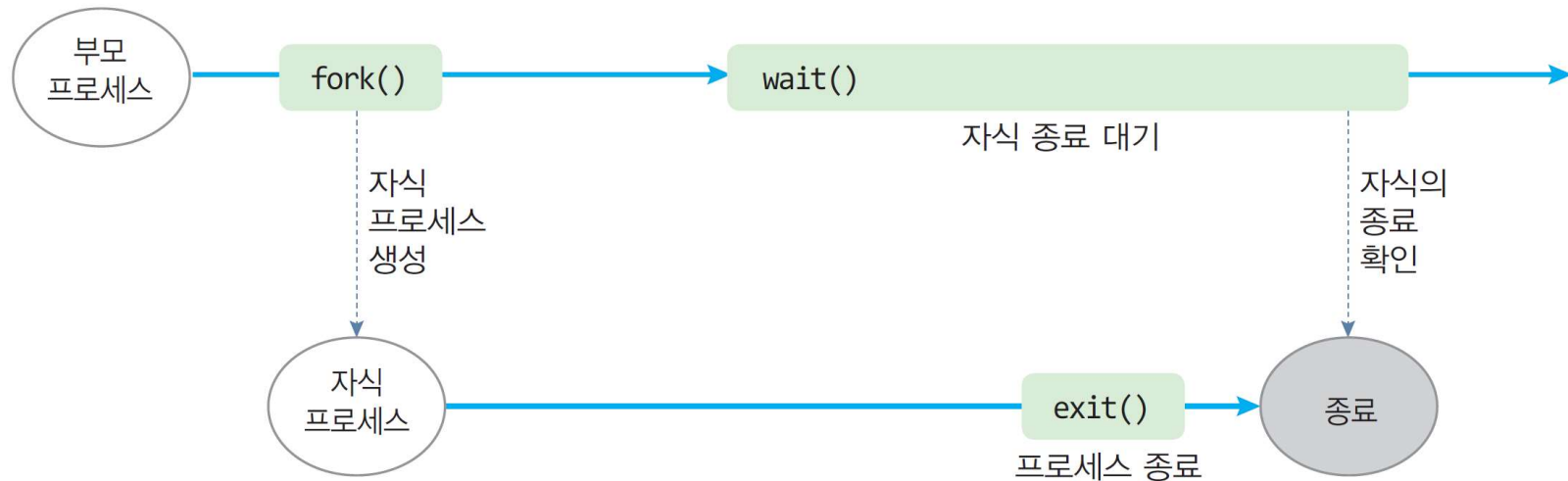
프로세스를 다루는 시스템 호출

37

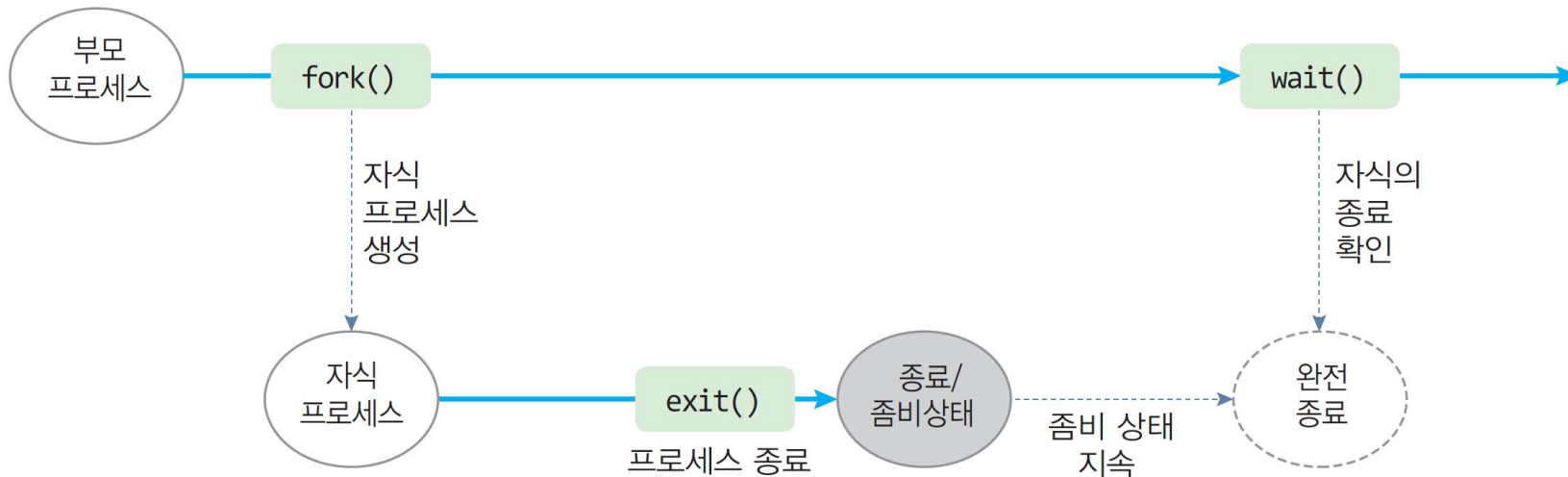
- fork()
 - ▣ 자식 프로세스를 생성하는 시스템 호출
- exit()
 - ▣ 현재 프로세스의 종료를 커널에 알리는 시스템 호출
 - ▣ 현재 프로세스의 종료를 처리하는 커널 코드 실행
- Wait()
 - ▣ 부모가 자식 프로세스의 종료를 기다리고 확인하는 시스템 호출

부모 프로세스와 자식 프로세스의 실행 관계

38



(a) 부모가 자식을 생성한 후 자식의 종료를 기다리는 경우



(b) 자식이 부모보다 먼저 종료한 경우

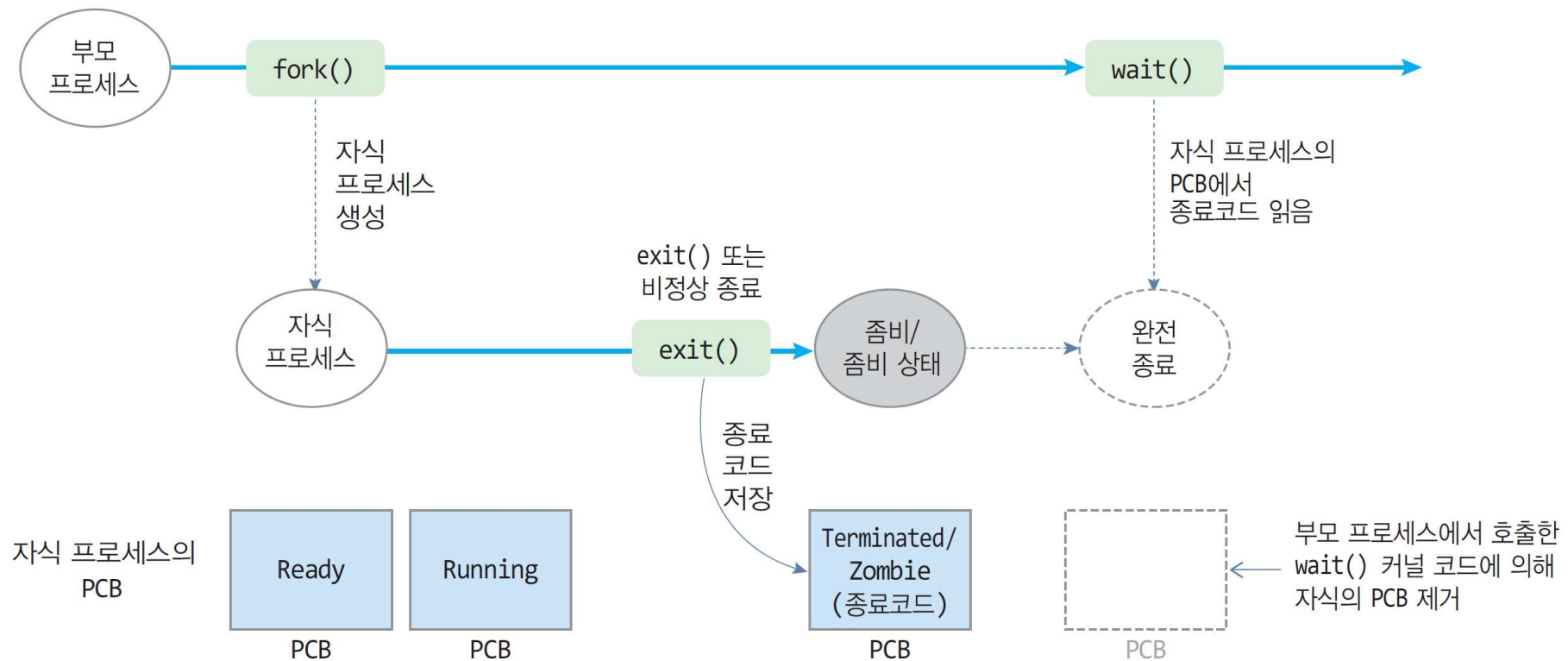
좀비 프로세스 - 종료 후 방치된 자식 프로세스

39

- 프로세스가 종료할 때,
 - ▣ PCB에, 종료코드(exit status) 저장
 - ▣ PCB에, 프로세스 상태를 Terminated라고 표시
 - ▣ 프로세스에게 할당된 모든 메모리 반환,
 - PCB와 프로세스 테이블의 항목은 제거되지 않음
- 부모 프로세스의 의무
 - ▣ wait() 시스템 호출을 통해 자식 프로세스의 종료 코드를 읽어야 함
 - ▣ 자식이 종료되면 부모에게 SIGCHLD 신호가 전송됨. 부모가 이 신호를 받았을 때 wait() 시스템 호출을 하도록 작성되어 있지 않다면 자식 프로세스는 계속 좀비 상태로 남아 있음
- 좀비 프로세스(zombie process)
 - ▣ 종료하였지만, 부모가 종료코드를 읽지 않은 상태의 프로세스
 - ▣ 프로세스 테이블에는 아직 남아 있으므로, 프로세스 목록을 출력할 때(ps 명령으로) 나타남
- 좀비 프로세스 제거 방법
 - ▣ 방법 1) 셸에서 부모 프로세스에게 SIGCHLD 신호 보내기
 - \$ kill -SIGCHLD 부모프로세스의PID'
 - 부모 프로세스의 SIGCHLD 핸들러가 wait() 호출하여 좀비 자식 제거
 - 부모의 SIGCHLD 핸들러가 wait() 호출하지 않으면 좀비는 제거되지 못함
 - ▣ 방법 2) 부모 프로세스 강제 종료
 - \$ kill -9 부모프로세스의PID'
 - 좀비는 init 프로세스의 자식이 되고,
 - init이 wait() 호출하여 좀비 프로세스 제거

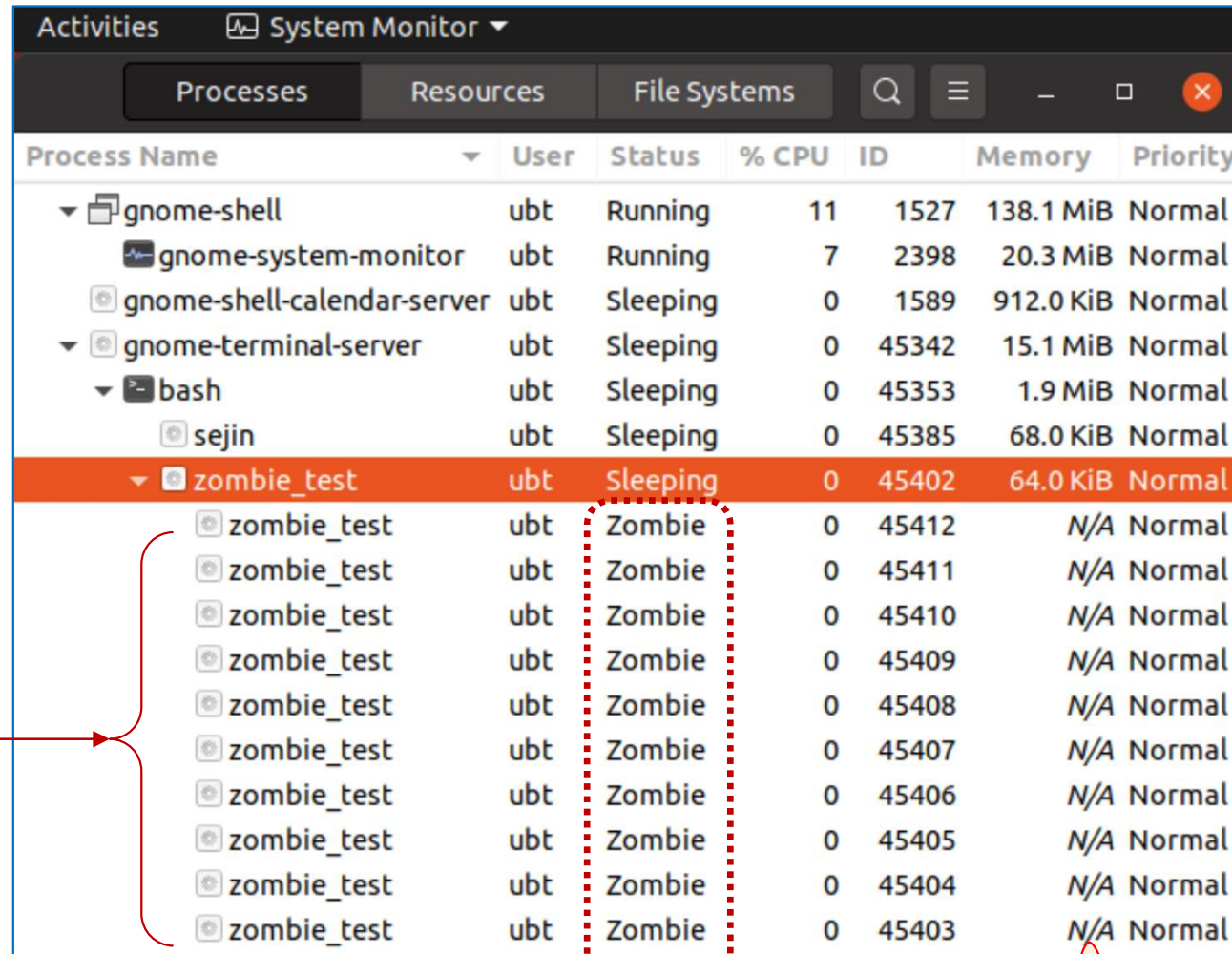
부모 프로세스에서 자식의 종료를 확인하는 wait() 시스템 호출

40



리눅스의 좀비 프로세스의 사례

41



Process Name	User	Status	% CPU	ID	Memory	Priority
gnome-shell	ubt	Running	11	1527	138.1 MiB	Normal
gnome-system-monitor	ubt	Running	7	2398	20.3 MiB	Normal
gnome-shell-calendar-server	ubt	Sleeping	0	1589	912.0 KiB	Normal
gnome-terminal-server	ubt	Sleeping	0	45342	15.1 MiB	Normal
bash	ubt	Sleeping	0	45353	1.9 MiB	Normal
sejin	ubt	Sleeping	0	45385	68.0 KiB	Normal
zombie_test	ubt	Sleeping	0	45402	64.0 KiB	Normal
zombie_test	ubt	Zombie	0	45412	N/A	Normal
zombie_test	ubt	Zombie	0	45411	N/A	Normal
zombie_test	ubt	Zombie	0	45410	N/A	Normal
zombie_test	ubt	Zombie	0	45409	N/A	Normal
zombie_test	ubt	Zombie	0	45408	N/A	Normal
zombie_test	ubt	Zombie	0	45407	N/A	Normal
zombie_test	ubt	Zombie	0	45406	N/A	Normal
zombie_test	ubt	Zombie	0	45405	N/A	Normal
zombie_test	ubt	Zombie	0	45404	N/A	Normal
zombie_test	ubt	Zombie	0	45403	N/A	Normal

우분투 리눅스에서
10개의 좀비 프로세스를 만든
사례를 System Monitor 프로
그램을 통해 관찰한 결과

테스트 프로그램으로 만든
10개의 좀비 프로세스

좀비 프로세스는 메모리가 제거되어 사용량
을 나타낼 수 없음

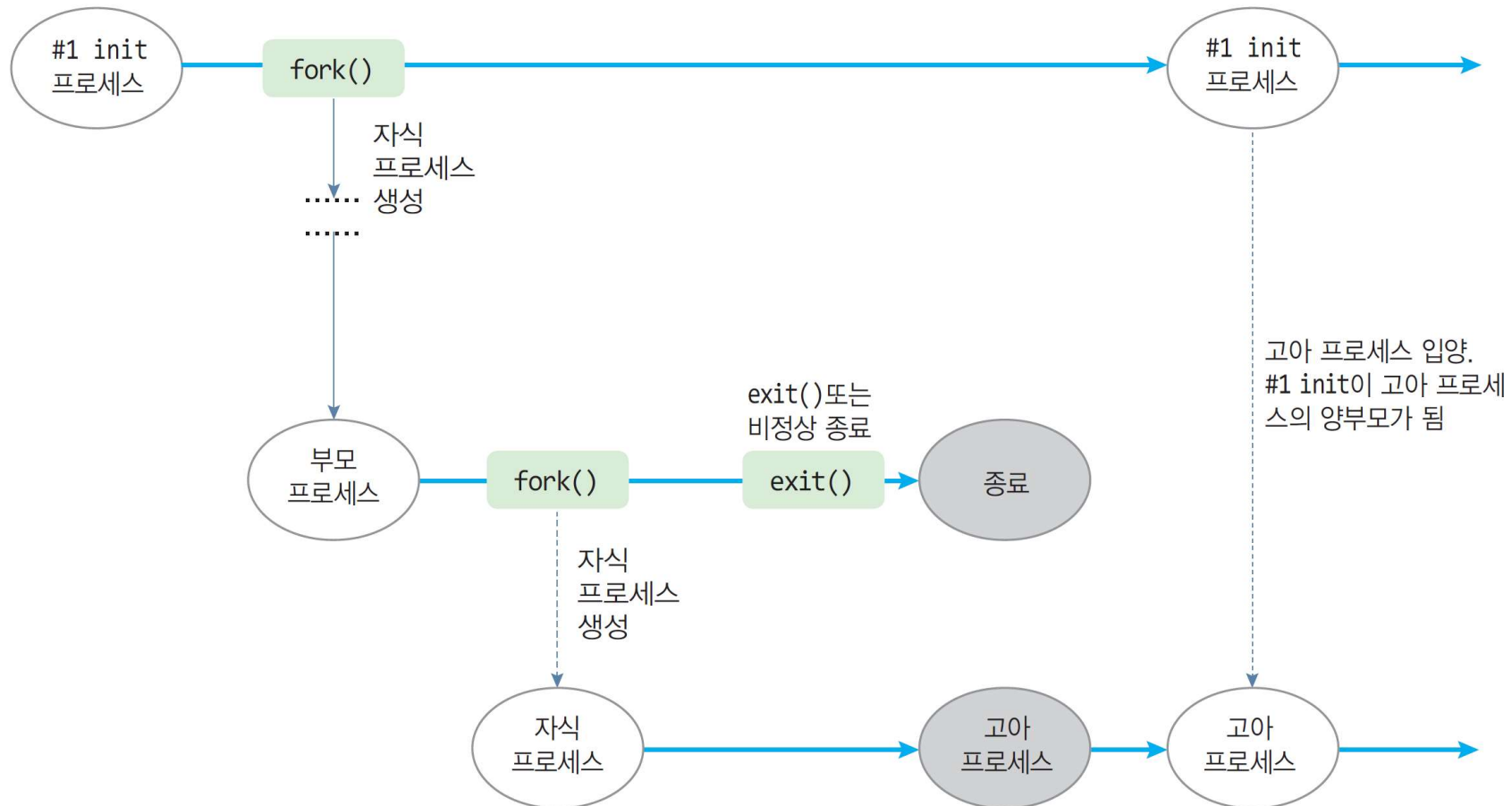
고아 프로세스와 입양

42

- 고아 프로세스(Orphan Process)
 - ▣ 부모가 먼저 종료한 자식 프로세스
- 부모 프로세스가 종료할 때
 - ▣ 일반적으로
 - 커널(exit() 시스템 호출 코드)은 자식 프로세스가 있는지 확인
 - 커널은 자식 프로세스(고아)를 init 프로세스에게 입양
 - ▣ 운영체제에 따라, 혹은 쉘의 경우
 - 모든 자식 프로세스 강제 종료시키기도 함

고아 프로세스가 #0 init 프로세스에 입양되는 과정

43



여러 종류의 프로세스

44

- 백그라운드 프로세스와 포그라운드 프로세스
 - ▣ 백그라운드 프로세스
 - 터미널에서 실행되었지만, 터미널 사용자와의 대화가 없는 채 실행되는 프로세스
 - 사용자와 대화없이 실행되는 프로세스
 - 사용자 입력을 필요로 하지 않는 프로세스
 - idle 상태로 잠을 자거나 디스크에 스왑된 상태의 프로세스
 - ▣ 포그라운드 프로세스
 - 실행되는 동안 터미널 사용자의 입력을 독점하는 프로세스
- CPU 집중 프로세스 vs. I/O 집중 프로세스
 - ▣ CPU 집중 프로세스(CPU intensive process)
 - 대부분의 시간을 계산 중심의 일(CPU 작업)을 하느라 보내는 프로세스
 - 배열 곱, 인공지능 연산, 이미지 처리 등의 작업
 - CPU 속도가 성능 좌우(CPU bound)
 - ▣ I/O 집중 프로세스(CPU intensive process)
 - 입출력 작업을 하느라 대부분의 시간을 보내는 프로세스
 - 네트워크 전송, 파일 입출력에 집중된 프로세스
 - 파일 서버, 웹 서버
 - 입출력 장치나 입출력 시스템의 속도가 성능 좌우(I/O bound)
 - ▣ 운영체제의 스케줄링 우선순위 : I/O 집중 프로세스 > CPU 집중 프로세스
 - I/O 작업하는 동안 다른 프로세스에게 CPU 할당 가능

4. 프로세스 제어

1. 프로세스 생성, `fork()`
2. 프로세스 오버레이, `exec()`
3. 자식 프로세스 종료 대기, `wait()`
4. 프로세스 종료, `exit()`
5. 좀비 프로세스 제어

프로세스 생성

46

- 컴퓨터 시스템에서 프로세스가 생성되는 5가지 경우
 - ▣ 시스템 부팅과정에서 필요한 프로세스 생성
 - ▣ 사용자의 로그인 후 사용자와 대화를 위한 프로세스 생성(bash 등 쉘)
 - ▣ 새로운 프로세스를 생성하도록 하는 사용자의 명령(vi a.c)
 - ▣ 배치 작업 실행 시(at, batch 명령)
 - ▣ 사용자 응용프로그램이 시스템 호출로 새 프로세스 생성
- 프로세스 생성
 - ▣ 프로세스가 프로세스를 생성
 - ▣ 시스템 호출을 통해서만 프로세스 생성
 - 커널 만이 프로세스를 생성하는 작업 가능
 - 리눅스 : fork() 시스템 호출
 - Windows : CreateProcess() 등 시스템 호출

프로세스 생성 과정

47

- 프로세스의 생성 과정
 - ▣ 새로운 PID 번호 할당
 - ▣ PCB 구조체 생성
 - ▣ 프로세스 테이블에서 새 항목 할당
 - ▣ 새로 할당된 프로세스 테이블 항목에 PCB 연결
 - ▣ 새로운 프로세스를 위한 메모리 공간 할당
 - 프로세스의 코드, 데이터, 스택, 힙 영역
 - 할당받은 메모리 공간에 프로세스의 코드와 데이터 적재
 - ▣ PCB에 프로세스 정보 기록
 - ▣ PCB에 프로세스 상태를 Ready로 표시하고, 준비 큐에 넣어서 차 후 스케줄되게 함

fork() 시스템 호출로 자식 프로세스 생성

48

□ fork() 시스템 호출

▣ 현재 프로세스를 복사하여 자식 프로세스 생성

■ `int pid = fork();`

- 자식 프로세스 생성
- 부모 프로세스의 모든 환경, 메모리, PCB 등을 복사
- 부모와 동일한 모양이지만, 독립된 주소 공간 소유
- (뒷부분의 쓰기 시 복사 참고)

■ 리턴 값

- 부모 프로세스에게는 자식 프로세스의 *PID* 리턴
- 자식 프로세스에게는 *0* 리턴

```
1  pid_t pid; // pid 변수 선언
2
3  pid = fork(); // 자식 프로세스 생성
4  if(pid > 0) {
5      /* 이곳에 부모 프로세스가 계속 실행할 코드 작성 */
6  }
7  else if(pid == 0) {
8      /* 이곳에 자식 프로세스가 실행할 코드 작성 */
9  }
10 else {
11     /* fork() 오류를 처리하는 코드 작성 */
12 }
```


탐구 3-5 fork()를 이용한 자식 프로세스 생성

49

forkex.c

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

int main() {
    pid_t pid;
    int i, sum=0;

    pid = fork(); // 자식프로세스 생성
    if(pid > 0) { // 부모 프로세스에 의해 실행되는 코드
        printf("부모프로세스: fork()의 리턴 값 = 자식프로세스 pid = %d\n", pid);
        printf("부모프로세스: pid = %d\n", getpid());
        wait(NULL); // 자식 프로세스가 종료할 때까지 대기
        printf("부모프로세스종료\n");
        return 0;
    }
    else if(pid == 0) { // 자식 프로세스에 의해 실행되는 코드
        printf("자식프로세스: fork()의 리턴 값 pid = %d\n", pid);
        printf("자식프로세스: pid = %d, 부모프로세스 pid = %d\n", getpid(), getppid());
        for (i=1; i<=100; i++)
            sum += i;
        printf("자식프로세스: sum = %d\n", sum);
        return 0;
    }
    else { // fork() 오류
        printf("fork 오류");
        return 0;
    }
}
```

```
$ gcc -o forkex forkex.c
```

```
$ ./forkex
```

```
부모프로세스: fork()의 리턴 값 = 자식프로세스 pid = 29138
```

```
부모프로세스: pid = 29137
```

```
자식프로세스: fork()의 리턴 값 pid = 0
```

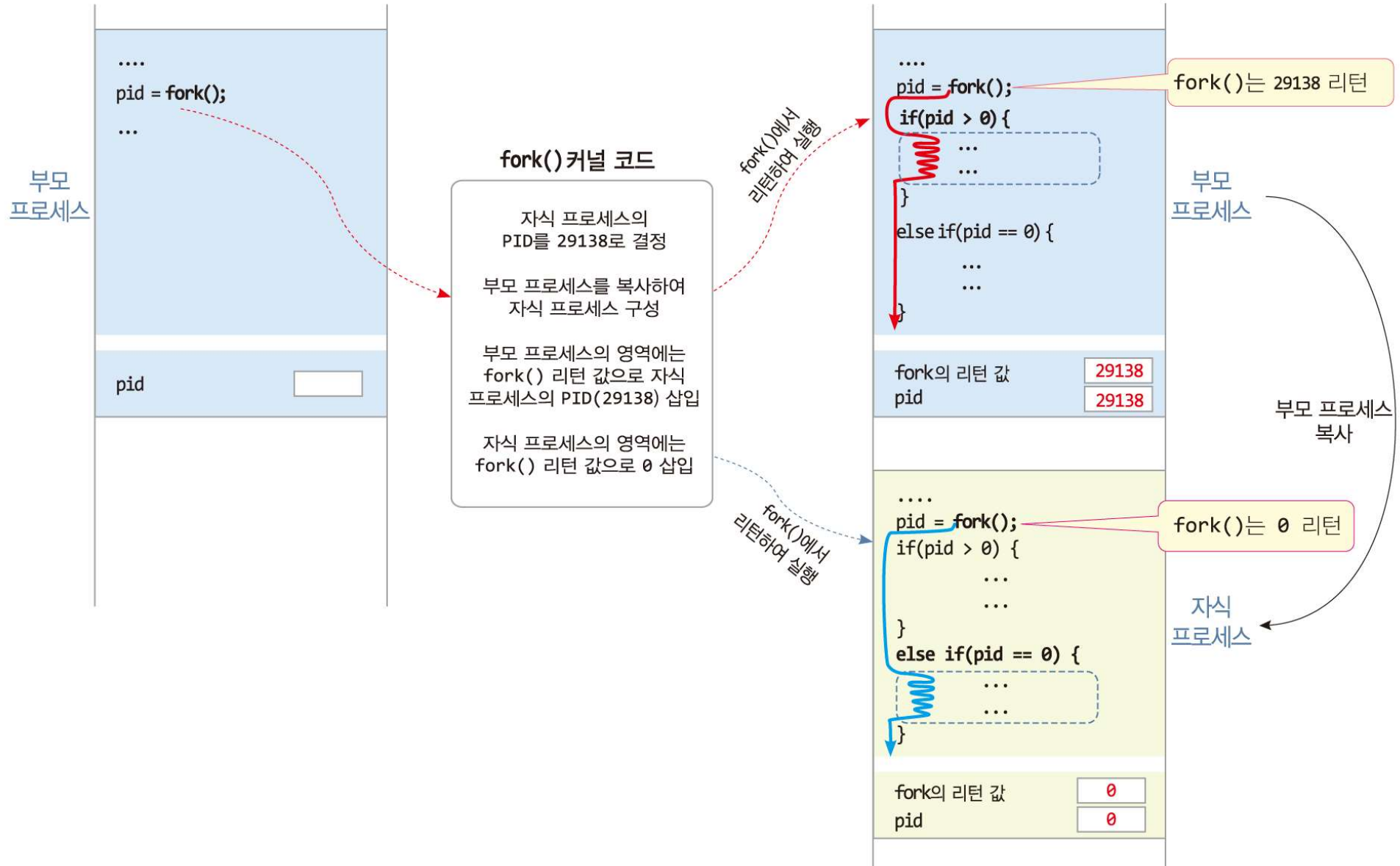
```
자식프로세스: pid = 29138, 부모프로세스 pid = 29137
```

```
자식프로세스: sum = 5050
```

```
부모프로세스종료
```

```
$
```

fork()의 실행 과정



(a) fork() 호출 전
부모 프로세스

(b) fork() 실행
자식 프로세스 생성

(c) 부모/자식 모두 fork()로부터의
리턴하여 각각 실행

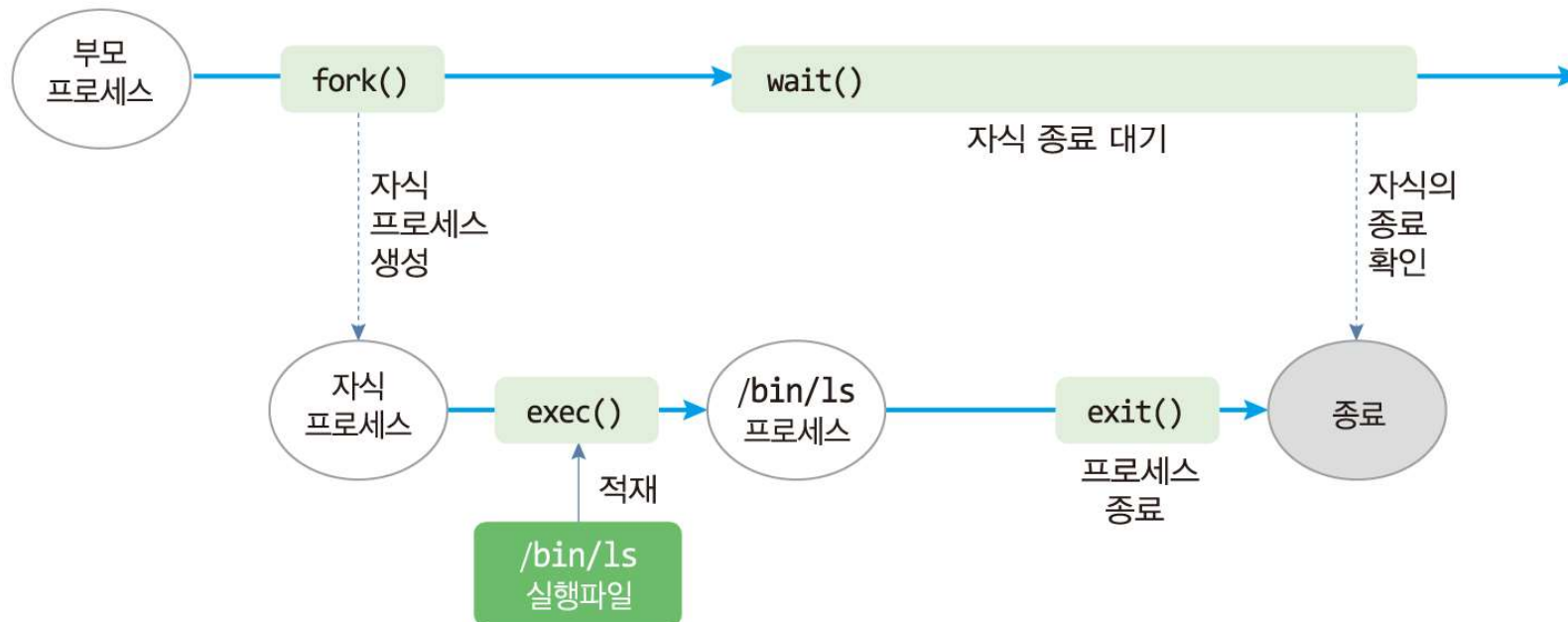
프로세스 오버레이, exec()

51

- 프로세스 오버레이(process overlay)
 - ▣ 현재 실행중인 프로세스의 주소 공간에 새로운 응용프로그램을 적재하여 실행시키는 기법
 - ▣ exec 패밀리 시스템 호출
 - execlp(), execv(), execvp() 시스템 호출들
 - 실행 파일을 적재하여 현재 프로세스의 메모리 공간에 단순히 덮어쓰고, 새로운 프로세스의 생성 과정을 거치지 않는다.
 - ▣ 프로세스의 PID 변경 없음
 - ▣ 프로세스의 코드, 데이터, 힙, 스택에 새로운 응용프로그램이 적재됨
 - ▣ fork()에 의해 생성된 자식 프로세스는 생성 후 바로 exec()을 실행하는 경우가 다반사임

exec()을 이용하여 /bin/ls 응용프로그램을 실행시키는 사례

52



탐구 3-6 fork()로 자식 프로세스 만들고 execlp()로 "ls -l" 명령을 오버레이하여 실행

53

execex.c

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

int main() {
    pid_t pid;

    pid = fork(); // 자식프로세스 생성

    if (pid > 0) { // 부모 프로세스 코드
        printf("부모프로세스: fork()의 리턴 값 = 자식프로세스 pid = %d\n", pid);
        printf("부모프로세스: 프로세스 pid = %d\n", getpid());
        wait(NULL); // 자식프로세스가 종료할 때까지 대기
        return 0;
    }
    else if (pid == 0) { // 자식 프로세스 코드
        printf("자식프로세스: fork()의 리턴 값 pid = %d\n", pid);
        printf("자식프로세스: pid = %d, 부모프로세스 pid = %d\n", getpid(), getppid());
        execlp("/bin/ls", "ls", "-l", NULL); // /bin/ls를 현재프로세스에 오버레이하여 실행
    }
    else { // fork() 오류
        printf("fork 오류");
        return 0;
    }
}
```

```
$ gcc -o execex execex.c
```

```
$ ./execex
```

```
부모프로세스: fork()의 리턴 값 = 자식프로세스 pid = 29566
```

```
부모프로세스: 프로세스 pid = 29565
```

```
자식프로세스: fork()의 리턴 값 pid = 0
```

```
자식프로세스: pid = 29566, 부모프로세스 pid = 29565
```

```
합계 32
```

```
-rwxrwxr-x 1 han00 han00 9016 4월 23 23:21 execex
```

```
-rw-rw-r-- 1 han00 han00 883 4월 23 23:21 execex.c
```

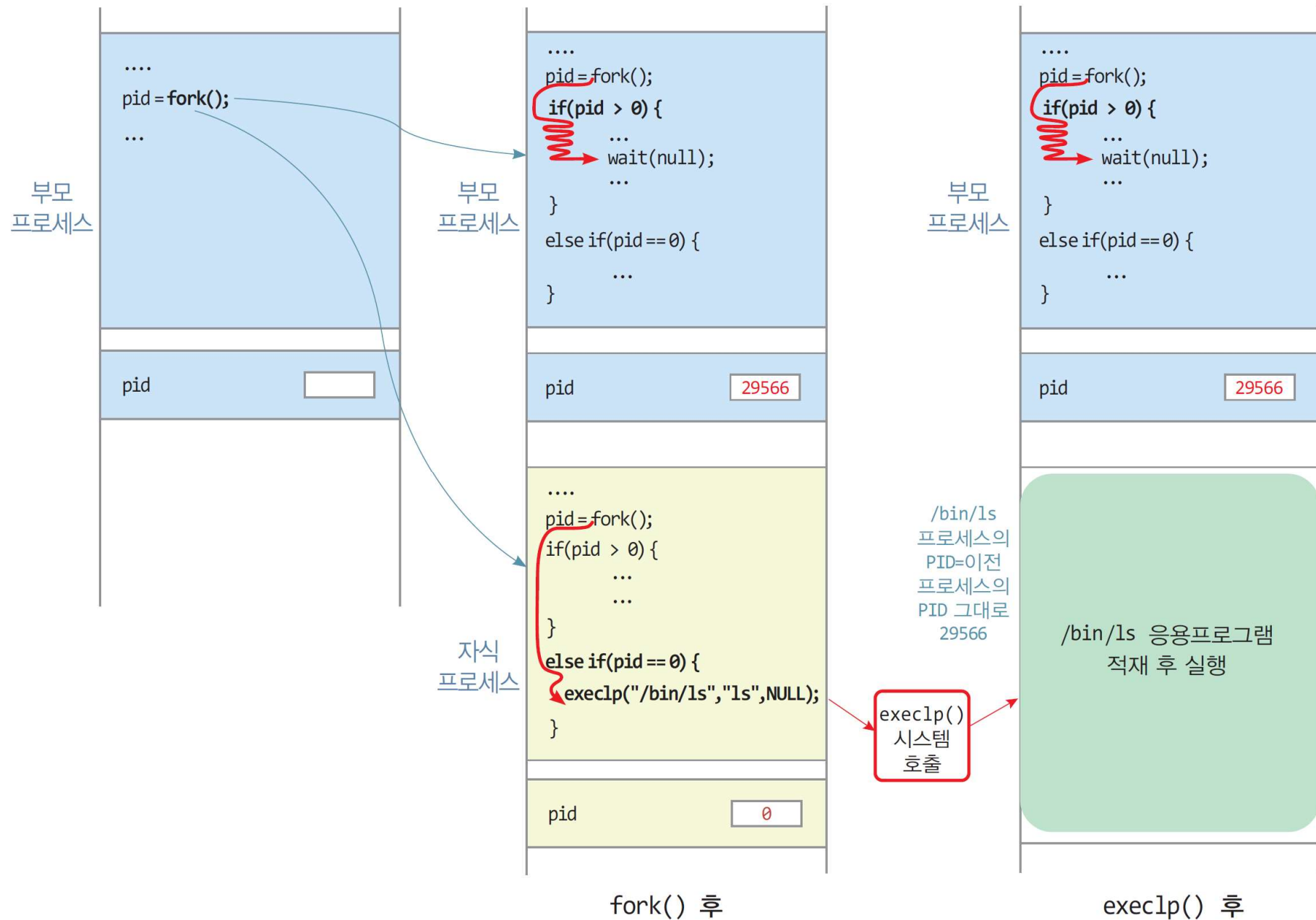
```
-rwxrwxr-x 1 han00 han00 9016 4월 23 16:20 forkex
```

```
-rw-rw-r-- 1 han00 han00 969 4월 23 16:20 forkex.c
```

```
$
```

"ls -l"의
실행 결과

탐구 3-6의 실행 과정



프로세스 종료와 프로세스 종료 대기

55

- 프로세스 종료
 - ▣ `exit()` 시스템 호출
 - ▣ C 프로그램의 `main()`에서 리턴
 - `exit()` 시스템 호출이 진행됨
- 종료 코드
 - ▣ 부모 프로세스에게 전달하는 값
 - `main()` 함수의 리턴 값; `return` 종료코드;
 - `exit(종료코드)`
- `exit()` 시스템 호출로 프로세스 종료 과정
 - (1) 프로세스의 모든 자원 반환
 - 코드, 데이터, 스택, 힙 등의 모든 메모리 자원을 반환
 - 열어 놓은 파일이나 소켓 등 닫음
 - (2) PCB에 프로세스 상태를 Terminated로 변경, PCB에 종료 코드 저장
 - (3) 자식 프로세스들이 있으면 이들을 init 프로세스에게 입양
 - (4) 부모 프로세스에게 SIGCHLD 신호 전송
 - 부모가 SIGCHLD 신호 핸들러를 작성하고 여기서 `wait()` 시스템 호출을 이용하여 자식의 종료 코드 읽기 실행
 - 혹은 언젠가 부모가 자식의 죽음 처리. 그동안 자식은 좀비 상태에 있음

종료 코드의 범위와 의미

56

- 종료 코드(exit code)
 - ▣ 프로세스가 종료한 상태나 이유를 부모에게 전달하기 위한 것
 - ▣ POSIX 표준에서 0~255사이의 1바이트 숫자
 - 정상 종료는 0
 - 1~255 - 개발자가 종료 이유를 임의로 정해 사용

- 종료 코드 사용 시 유의할 점

- ▣ main이나 exit()에서 255 이상의 값을 사용할 때 유의

```
int main() {  
    return 300; // return 44;와 같음  
}  
void func() {  
    exit(300); // exit(44)와 같음  
}
```

- ▣ -1을 리턴하는 경우(return -1, 혹은exit(-1))
 - -1 -> 0xff -> 양의 정수로 255. 그러므로 종료 코드로 255가 전달

탐구 3-7 wait()로 자식 프로세스 종료 대기

57

child.c

```
#include <stdio.h>

int main() {
    printf("I am a child\n");
    return 100;
}
```

```
$ gcc -o child child.c
$ gcc -o waitex waitex.c
$ ./waitex
부모프로세스: 자식의 종료를 기다림
I am a child
부모프로세스: child의 종료 코드=100
$
```

waitex.c

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

int main() {
    pid_t pid;
    int status;

    pid = fork(); // 자식프로세스 생성

    if (pid > 0) { // 부모 프로세스 코드
        printf("부모프로세스: 자식의 종료를 기다림\n");
        wait(&status); // 자식프로세스 종료 대기. status에 종료 코드 받음
        printf("부모프로세스: child의 종료 코드=%d\n", WEXITSTATUS(status));
        return 0;
    }
    else if (pid == 0) { // 자식 프로세스 코드
        execlp("./child", "child", NULL); // child를 자식프로세스로 실행
    }
    else { // fork() 오류
        printf("fork 오류");
        return 0;
    }
}
```

프로세스 종료와 좀비 프로세스

58

□ 프로세스 종료

▣ 두 종류

- 1) C 언어에서 main() 함수의 종료나 exit()을 호출한 정상 종료
- 2) 다른 프로세스에 의해 강제 종료(kill)

▣ 프로세스가 종료되면

- 차지하고 있던 메모리와 자원 모두 반환
- PCB는 프로세스 테이블에서 제거되지 않음
- 프로세스 상태 : Terminated

▣ 부모 프로세스가 wait() 시스템 호출을 통해, 죽은 자식이 남긴 종료 코드를 읽게 되면, 자식 프로세스의 PCB가 완전히 제거

□ 좀비 프로세스

- ▣ 종료할 때 리턴한 정보(main() 함수에서 리턴값, 종료 코드)를 부모 프로세스가 읽지 않았을 때, 죽었지만 PCB만 남아 완전히 제거되지 못한 상태

탐구 3-8 좀비 프로세스 만들고 관찰하기

59

zombieex.c

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

```
int main() {
    pid_t pid, zompid;
    int status;
```

```
    pid = fork();
```

```
    if(pid > 0) { // 부모 프로세스 코드
```

```
        sleep(10); // 10초 동안 잠자기
```

```
        zompid = wait(&status); // 자식프로세스 종료 대기
```

```
        printf("부모프로세스: 자식 PID=%d, 종료 코드=%d\n", zompid, WEXITSTATUS(status));
        return 0;
    }
```

```
    else if(pid == 0) { // 자식프로세스 코드
```

```
        printf("자식프로세스: %d 종료합니다.\n", getpid());
```

```
        exit(100); // 자식이 종료하여 좀비 프로세스가 됨. 종료 코드 100 전달
```

```
    }
```

```
    else { // fork() 오류
```

```
        printf("fork 오류");
```

```
        return 0;
    }
```

```
}
```

부모프로세스가 10초 동안 잠자는 동안, 자식프로세스가 종료하여 좀비 프로세스가 됨. 그 사이에 셸에서 ps -l 명령을 입력하여 좀비 프로세스 관찰

status에 종료 코드가 들어 있으며, zompid에는 종료한 자식프로세스의 PID가 들어 있음

탐구 3-8 실행

60

```
$ gcc -o zombieex zombieex.c
```

```
$
```

```
$ ./zombieex&
```

ps -l 명령을 입력하기 위해 백그라운드로 실행

```
[1] 30748
```

```
자식프로세스: 30749 종료합니다.
```

자식 프로세스 30749. 자식프로세스는 종료 후 좀비가 됨

```
$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	30580	30579	0	80	0	-	5932	wait	pts/8	00:00:00	bash
0	S	1000	30748	30580	0	80	0	-	1055	hrttime	pts/8	00:00:00	zombieex
1	Z	1000	30749	30748	0	80	0	-	0	-	pts/8	00:00:00	zombieex <defunct>
0	R	1000	30750	30580	0	80	0	-	7549	-	pts/8	00:00:00	ps

좀비 상태

부모 프로세스 30748

좀비 프로세스 30749

```
부모프로세스: 자식 PID=30749, 종료 코드=100
```

할당된 물리 메모리의 크기 0

```
[1]+  완료
```

```
zombieex
```

```
$
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
1	Z	1000	30749	30748	0	80	0	-	0	-	pts/8	00:00:00	zombieex <defunct>

좀비 상태

프로세스 ID

부모 프로세스 ID

현재 메모리 할당량 0

좀비프로세스