

프로세스와 스레드

무단 배포 및 공유를 금합니다

쉽게 배우는 운영체제, 한빛출판

프로세스

학습목표

- 프로세스가 생성된 후 어떤 상태 변화를 거치는지 알아본다.
- 프로세스 제어 블록의 구성과 문맥 교환 시 동작 과정을 이해한다.
- 프로세스의 생성과 복사, 전환 과정을 이해한다.
- 스레드의 개념을 이해하고 멀티스레드 시스템의 장점을 알아본다.

1-1 프로세스의 개념

■ 프로그램

- 저장장치에 저장되어 있는 정적인 상태

■ 프로세스

- 실행을 위해 메모리에 올라온 동적인 상태



그림 3-1 레시피와 요리에 비유한 프로그램과 프로세스

1-2 요리사 모형에의 비유

■ 코스 요리 메뉴와 주문서

<p>코스 요리 A 1인당 3만 원</p> <p>전 채: 샐러드/수프 메 인: 안심/등심 디저트: 녹차/커피</p>	<p>코스 요리 B 1인당 2.5만 원</p> <p>전 채: 족/수프 메 인: 닭튀김/새우튀김 디저트: 과일/커피</p>	<p>코스 요리 C 1인당 2.1만 원</p> <p>전 채: 족/샐러드 메 인: 새우튀김/안심 디저트: 커피/케이크</p>
---	--	---

그림 3-2 코스 요리 메뉴

<p>-일련번호: 16 -테이블 번호: 4 -전체 인원: 1 -코스: C • 샐러드 • 안심: 웰던 • 케이크: 딸기</p>	<p>-일련번호: 15 -테이블 번호: 3 -전체 인원: 1 -코스: C • 족: 채소 • 안심: 미디엄 • 케이크: 초콜릿</p>	<p>-일련번호: 14 -테이블 번호: 2 -전체 인원: 2 -코스: A • 수프: 양파 • 등심: 미디엄 • 녹차</p>	<p>-일련번호: 13 -테이블 번호: 2 -전체 인원: 2 -코스: B • 수프: 토마토 • 새우튀김: 바삭하게 • 과일</p>	<p>-일련번호: 12 -테이블 번호: 1 -전체 인원: 1 -코스: A • 샐러드 • 안심: 웰던 • 커피: 진하게</p>
---	---	--	--	---

그림 3-3 주문서의 예

1-2 요리사 모형에의 비유

■ 일괄 작업 방식의 요리

- 레스토랑에 테이블이 하나만 있는 것
- 요리사는 주문서를 받은 순서대로 요리
- 현재 손님의 식사가 끝나야 다음 손님을 받을 수 있어 작업 효율이 떨어짐
- 주문서가 도착한 순서대로 요리를 하기 위해 '주문 목록'을 사용(주문 목록은 큐(queue)로 처리)

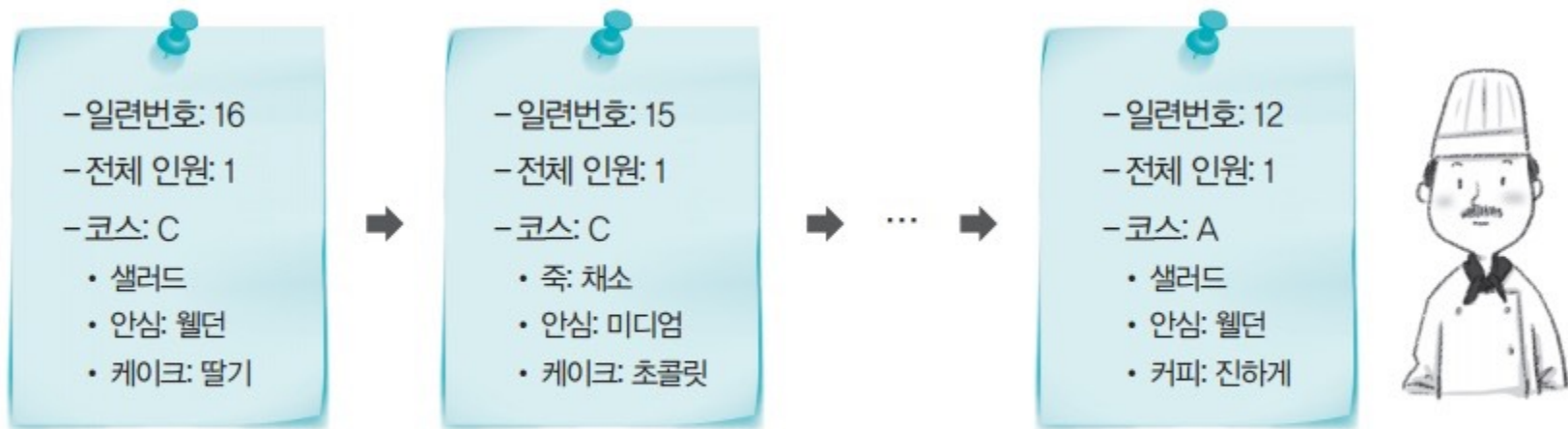


그림 3-4 일괄 작업 방식의 요리

1-2 요리사 모형에의 비유

■ 시분할 방식의 요리

- 요리사 1명이 시간을 적당히 배분하여 여러 가지 요리를 동시에 하는 방식
- 요리사는 주문 목록에 있는 주문서 중 하나를 가져다가 요리함
- 모든 요리가 제공되면 주문 목록에서 삭제

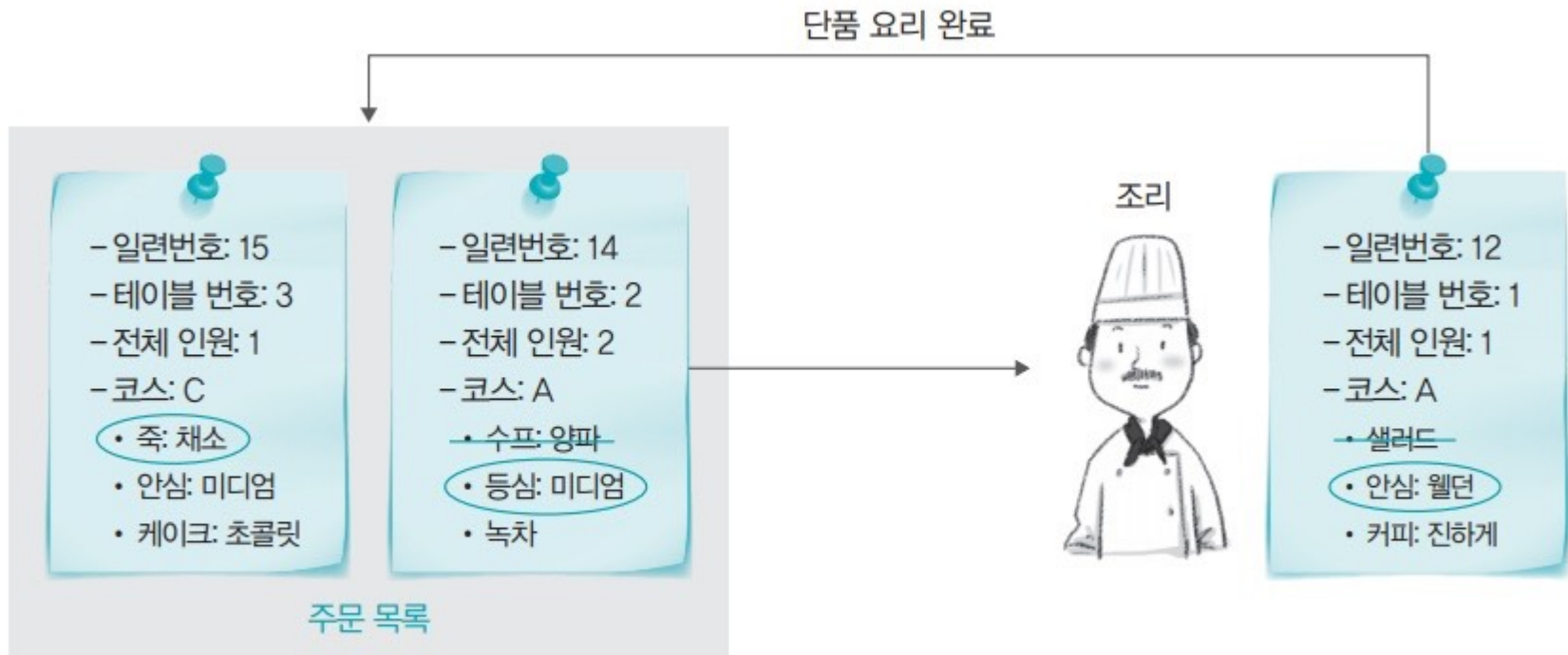


그림 3-5 시분할 방식의 요리

1-3 프로그램에서 프로세스로의 전환

- **프로세스 제어 블록 Process Control Block, PCB**
 - 운영체제가 해당 프로세스를 위해 관리하는 자료 구조
 - 프로세스 구분자 : 각 프로세스를 구분하는 구분자
 - 메모리 관련 정보 : 프로세스의 메모리 위치 정보
 - 각종 중간값 : 프로세스가 사용했던 중간값

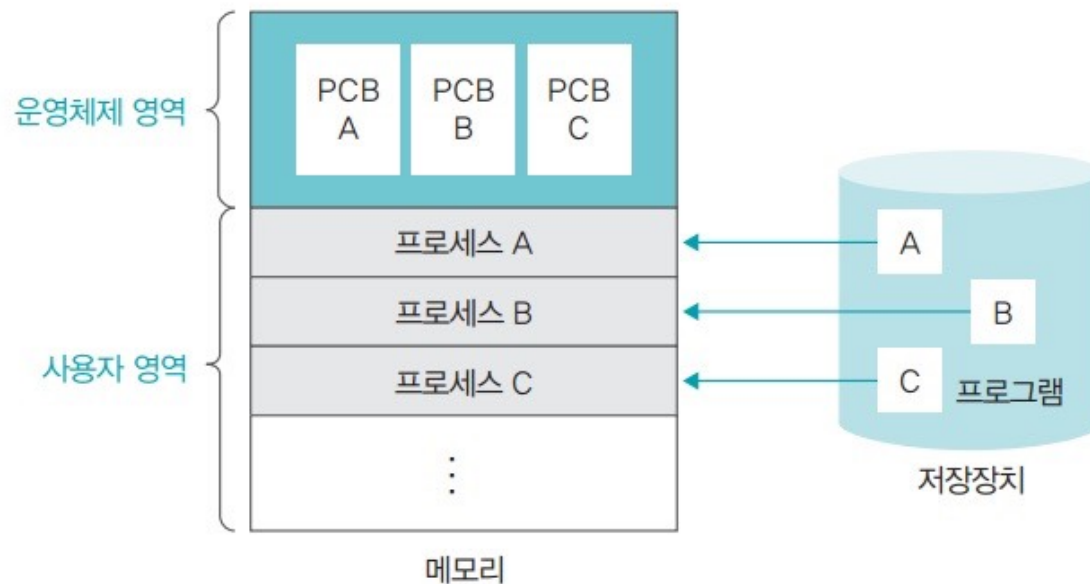


그림 3-7 프로그램이 메모리에 올라와 프로세스가 되는 과정

1-3 프로그램에서 프로세스로의 전환

■ 프로세스와 프로그램의 관계

- 프로그램이 프로세스가 된다는 것은 운영체제로부터 프로세스 제어 블록을 얻는다는 뜻
- 프로세스가 종료된다는 것은 해당 프로세스 제어 블록이 폐기된다는 뜻

정의 3-1 프로세스와 프로그램의 관계

프로세스 = 프로그램 + 프로세스 제어 블록

프로그램 = 프로세스 - 프로세스 제어 블록

1-4 프로세스의 상태

■ 프로세스의 네 가지 상태

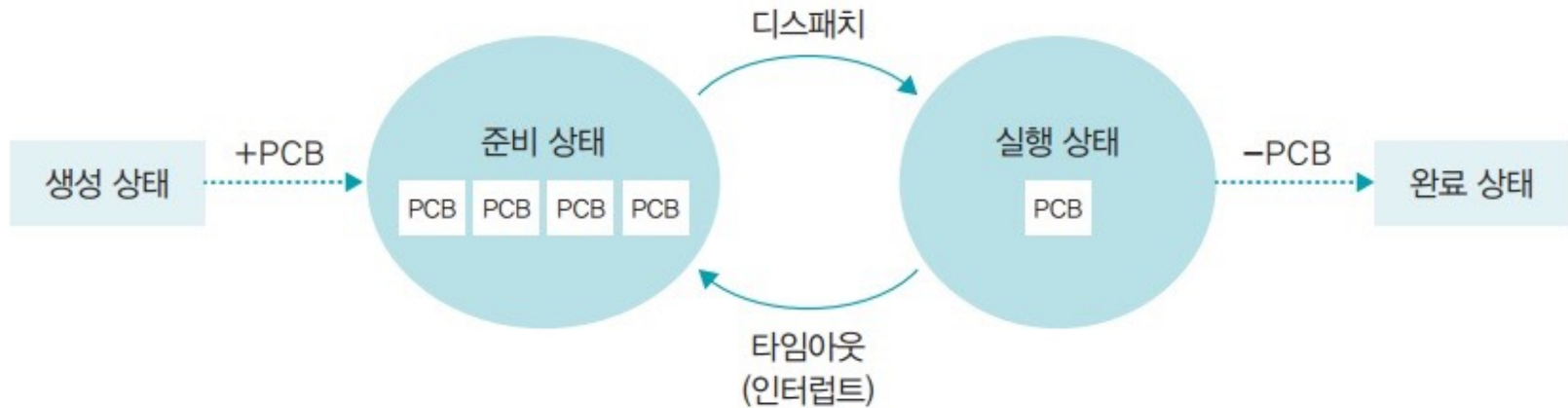


그림 3-8 간단한 프로세스의 상태

- **생성 상태** : 프로세스가 메모리에 올라와 실행 준비를 완료한 상태
- **준비 상태** : 생성된 프로세스가 CPU를 얻을 때까지 기다리는 상태
- **실행 상태** : 준비 상태에 있는 프로세스 중 하나가 CPU를 얻어 실제 작업을 수행하는 상태
- **완료 상태** : 실행 상태의 프로세스가 주어진 시간 동안 작업을 마치면 진입하는 상태(프로세스 제어 블록이 사라진 상태)

1-4 프로세스의 상태

■ 프로세스의 네 가지 상태

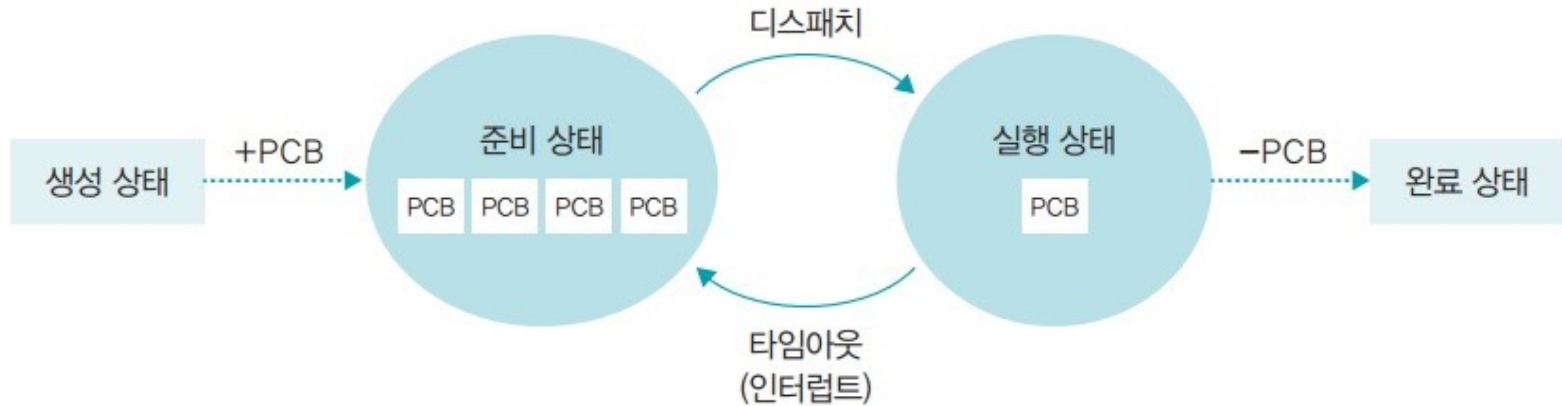


그림 3-8 간단한 프로세스의 상태

- **디스패치** : 준비 상태의 프로세스 중 하나를 골라 실행 상태로 바꾸는 CPU 스케줄러의 작업
- **타임아웃** : 프로세스가 자신에게 주어진 하나의 타임 슬라이스 동안 작업을 끝내지 못하면 다시 준비 상태로 돌아가는 것

1-4 프로세스의 상태

■ 프로세스의 다섯 가지 상태

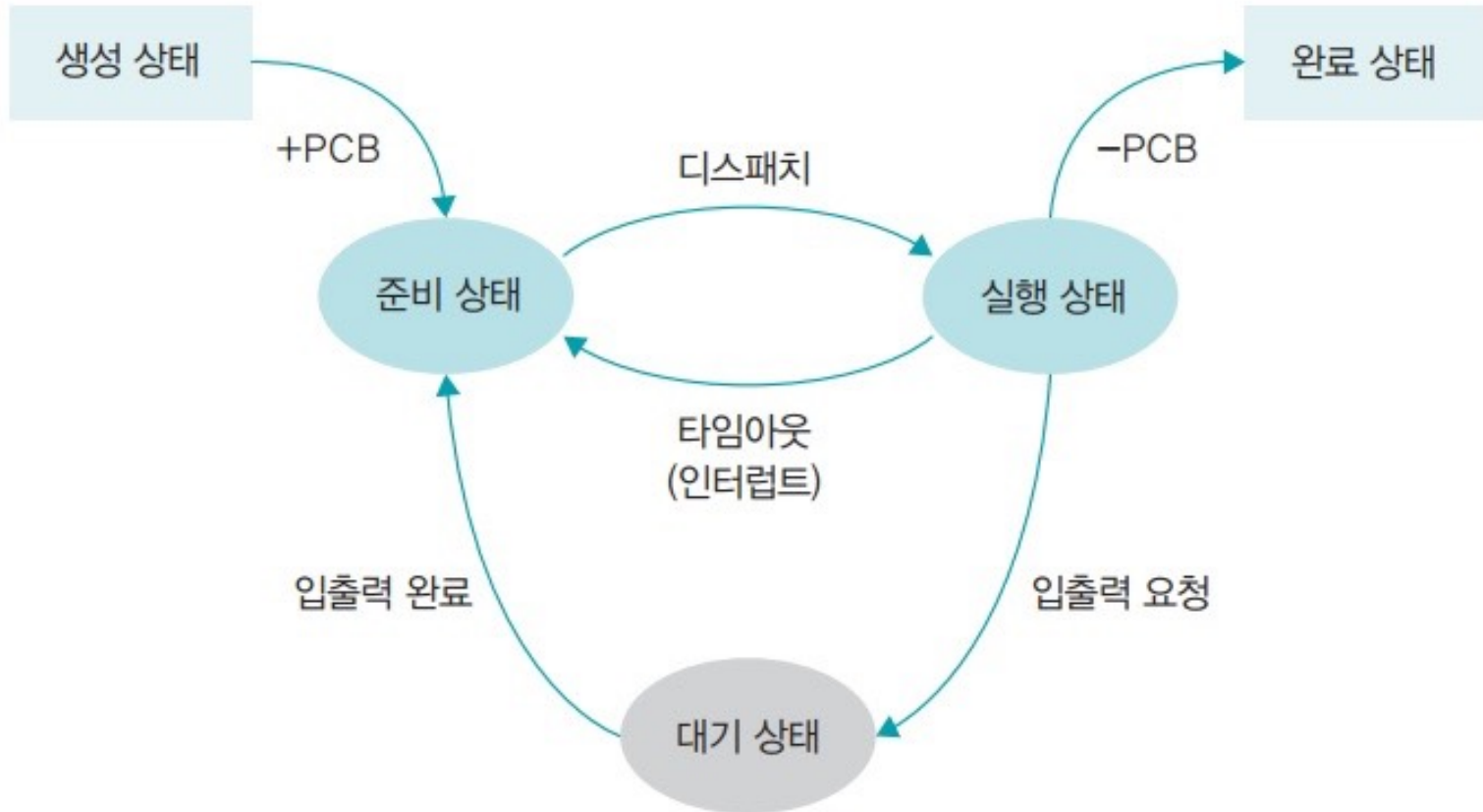


그림 3-9 대기 상태를 추가한 프로세스의 상태

1-4 프로세스의 상태

■ 생성 상태

- 프로그램이 메모리에 올라오고 운영체제로부터 프로세스 제어 블록을 할당받은 상태
- 생성된 프로세스는 바로 실행되는 것이 아니라 준비 상태에서 자기 순서를 기다리며, 프로세스 제어 블록도 같이 준비 상태로 옮겨짐

■ 준비 상태

- 실행 대기 중인 모든 프로세스가 자기 순서를 기다리는 상태
- 프로세스 제어 블록은 준비 큐에서 기다리며 CPU 스케줄러에 의해 관리
- CPU 스케줄러는 준비 상태에서 큐를 몇 개 운영할지, 큐에 있는 어떤 프로세스의 프로세스 제어 블록을 실행 상태로 보낼지 결정
- CPU 스케줄러가 어떤 프로세스 제어 블록을 선택하는 작업은 `dispatch(PID)` 명령으로 처리
- CPU 스케줄러가 `dispatch(PID)`를 실행하면 해당 프로세스가 준비 상태에서 실행 상태로 바뀌어 작업이 이루어짐

1-4 프로세스의 상태

■ 실행 상태

- 프로세스가 CPU를 할당받아 실행되는 상태
- 실행 상태에 있는 프로세스는 자신에게 주어진 시간, 즉 타임 슬라이스 동안만 작업할 수 있음
- 그 시간을 다 사용하면 `timeout(PID)`가 실행되어 실행 상태에서 준비 상태로 옮김
- 실행 상태 동안 작업이 완료되면 `exit(PID)`가 실행되어 프로세스가 정상 종료
- 실행 상태에 있는 프로세스가 입출력을 요청하면 CPU는 입출력 관리자에게 입출력을 요청하고 `block(PID)`를 실행
- `block(PID)`는 입출력이 완료될 때까지 작업을 진행할 수 없기 때문에 해당 프로세스를 대기 상태로 옮기고 CPU 스케줄러는 새로운 프로세스를 실행 상태로 가져옴

1-4 프로세스의 상태

■ 대기 상태

- 실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태
- 대기 상태의 프로세스는 입출력장치별로 마련된 큐에서 기다리다가 완료되면 인터럽트가 발생하고, 대기 상태에 있는 여러 프로세스 중 해당 인터럽트로 깨어날 프로세스를 찾는데 이것이 wakeup(PID)
- wakeup(PID)로 해당 프로세스의 프로세스 제어 블록이 준비 상태로 이동

■ 완료 상태

- 프로세스가 종료되는 상태
- 코드와 사용했던 데이터를 메모리에서 삭제하고 프로세스 제어 블록을 폐기
- 정상적인 종료는 간단히 `exit()`로 처리
- 오류나 다른 프로세스에 의해 비정상적으로 종료되는 강제 종료를 만나면 디버깅하기 위해 종료 직전의 메모리 상태를 저장장치로 옮기는데 이를 코어 덤프(core dump)라고 함

1-4 프로세스의 상태

표 3-1 프로세스의 상태와 관련 작업

상태	설명	작업
생성 상태	프로그램을 메모리에 가져와 실행 준비가 완료된 상태이다.	메모리 할당, 프로세스 제어 블록 생성
준비 상태	실행을 기다리는 모든 프로세스가 자기 차례를 기다리는 상태이다. 실행될 프로세스를 CPU 스케줄러가 선택한다.	dispatch(PID): 준비→실행
실행 상태	선택된 프로세스가 타임 슬라이스를 얻어 CPU를 사용하는 상태이다. 프로세스 사이의 문맥 교환이 일어난다.	timeout(PID): 실행→준비 exit(PID): 실행→완료 block(PID): 실행→대기
대기 상태	실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태이다. 입출력이 완료되면 준비 상태로 간다.	wakeup(PID): 대기→준비
완료 상태	프로세스가 종료된 상태이다. 사용하던 모든 데이터가 정리된다. 정상 종료인 exit와 비정상 종료인 abort를 포함한다.	메모리 삭제, 프로세스 제어 블록 삭제

1-4 프로세스의 상태

■ 휴식 상태

- 프로세스가 작업을 일시적으로 쉬고 있는 상태
- 유닉스에서 프로그램을 실행하는 동중에 [Ctrl]+[Z] 키를 누르면 볼 수 있음
- 종료 상태가 아니기 때문에 원할 때 다시 시작할 수 있는 상태

```
zoch@ubuntu:~$ sleep 100
^Z
[1]+  Stopped                  sleep 100
zoch@ubuntu:~$ jobs
[1]+  Stopped                  sleep 100
zoch@ubuntu:~$ bg
[1]+ sleep 100 &
zoch@ubuntu:~$ jobs
[1]+  Running                  sleep 100 &
```

그림 3-10 유닉스에서 프로세스 정지 상태 화면

1-4 프로세스의 상태

■ 보류 상태

- 프로세스가 메모리에서 잠시 쫓겨난 상태
- 프로세스는 다음과 같은 경우에 보류 상태가 됨
 - 메모리가 꽉 차서 일부 프로세스를 메모리 밖으로 내보낼 때
 - 프로그램에 오류가 있어서 실행을 미루어야 할 때
 - 바이러스와 같이 악의적인 공격을 하는 프로세스라고 판단될 때
 - 매우 긴 주기로 반복되는 프로세스라 메모리 밖으로 쫓아내도 큰 문제가 없을 때
 - 입출력을 기다리는 프로세스의 입출력이 계속 지연될 때

1-4 프로세스의 상태

■ 보류 상태를 포함한 프로세스의 상태

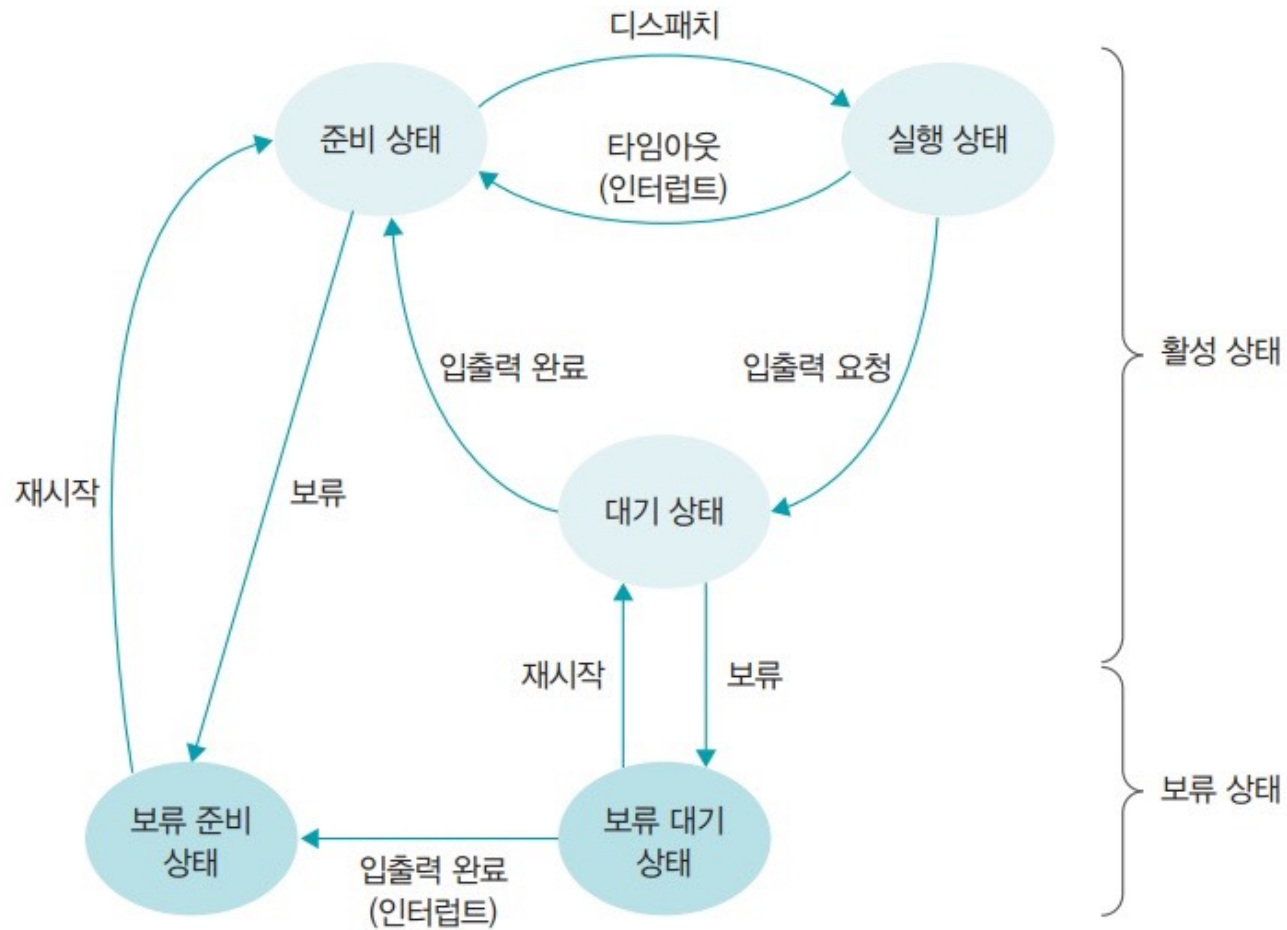


그림 3-11 보류 상태를 포함한 프로세스의 상태

2-1 프로세스 제어 블록

■ 프로세스 제어 블록(PCB)

- 프로세스를 실행하는 데 필요한 중요한 정보를 보관하는 자료 구조
- 프로세스는 고유의 프로세스 제어 블록을 가짐
- 프로세스 생성 시 만들어져서 프로세스가 실행을 완료하면 폐기

■ 프로세스 제어 블록의 구성

포인터	프로세스 상태
프로세스 구분자	
프로그램 카운터	
프로세스 우선순위	
각종 레지스터 정보	
메모리 관리 정보	
할당된 자원 정보	
계정 정보	
PPID와 CPID	
⋮	

그림 3-12 프로세스 제어 블록의 구성

2-1 프로세스 제어 블록

■ 프로세스 제어 블록의 구성(1)

- **포인터** : 준비 상태나 대기 상태의 큐를 구현할 때 사용
- **프로세스 상태** : 프로세스가 현재 어떤 상태에 있는지를 나타내는 정보
- **프로세스 구분자** : 운영체제 내에 있는 여러 프로세스를 구현하기 위한 구분자
- **프로그램 카운터** : 다음에 실행될 명령어의 위치를 가리키는 프로그램 카운터의 값
- **프로세스 우선순위** : 프로세스의 실행 순서를 결정하는 우선순위
- **각종 레지스터 정보** : 프로세스가 실행되는 중에 사용하던 레지스터의 값

포인터	프로세스 상태
프로세스 구분자	
프로그램 카운터	
프로세스 우선순위	
각종 레지스터 정보	
메모리 관리 정보	
할당된 자원 정보	
계정 정보	
PPID와 CPID	
⋮	

그림 3-12 프로세스 제어 블록의 구성

2-1 프로세스 제어 블록

■ 프로세스 제어 블록의 구성(2)

- **메모리 관리 정보** : 프로세스가 메모리의 어디에 있는지 나타내는 메모리 위치 정보, 메모리 보호를 위해 사용하는 경계 레지스터 값과 한계 레지스터 값 등
- **할당된 자원 정보** : 프로세스를 실행하기 위해 사용하는 입출력 자원이나 오픈 파일 등에 대한 정보
- **계정 정보** : 계정 번호, CPU 할당 시간, CPU 사용 시간 등
- **부모 프로세스 구분자와 자식 프로세스 구분자** : 부모 프로세스를 가리키는 PPID와 자식 프로세스를 가리키는 CPID 정보

포인터	프로세스 상태
프로세스 구분자	
프로그램 카운터	
프로세스 우선순위	
각종 레지스터 정보	
메모리 관리 정보	
할당된 자원 정보	
계정 정보	
PPID와 CPID	
⋮	

그림 3-12 프로세스 제어 블록의 구성

2-1 프로세스 제어 블록

■ 포인터

- 대기 상태에는 같은 입출력을 요구한 프로세스끼리 연결할 때 포인터 사용

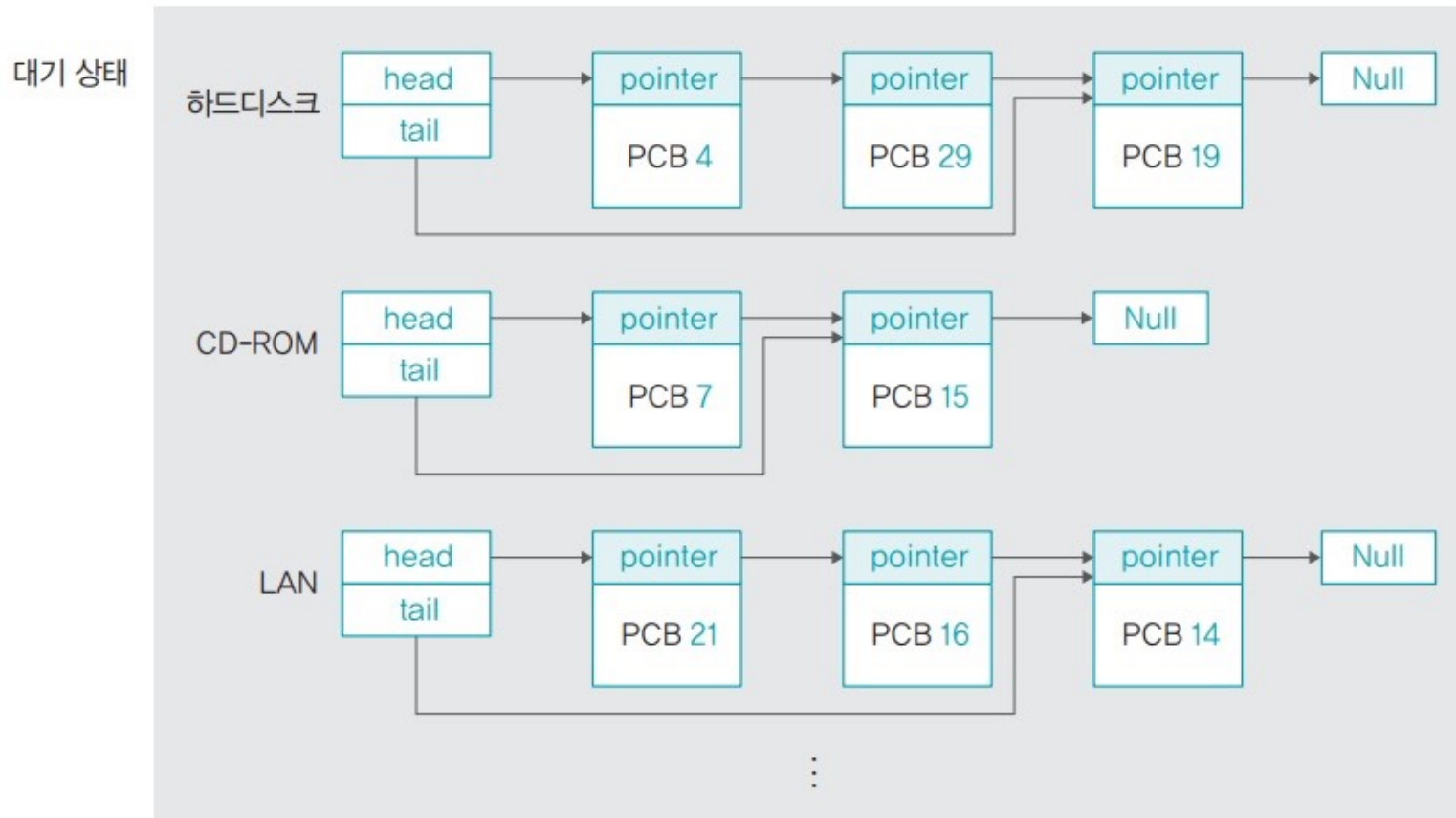


그림 3-13 대기 상태의 대기 큐

2-2 문맥 교환

■ 요리 작업의 전환

- 주문서를 바꾸는 것과 동시에 작업 환경을 바꾸는 것

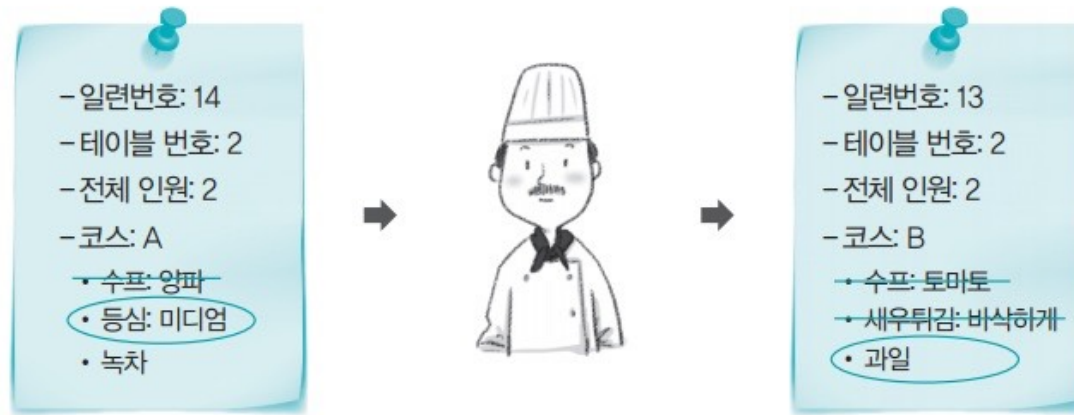


그림 3-14 요리 작업의 전환 과정

■ 문맥 교환

- CPU를 차지하던 프로세스가 나가고 새로운 프로세스를 받아들이는 작업
- 실행 상태에서 나가는 프로세스 제어 블록에는 지금까지의 작업 내용을 저장하고, 반대로 실행 상태로 들어오는 프로세스 제어 블록의 내용으로 CPU가 다시 세팅

2-2 문맥 교환

■ 문맥 교환 절차

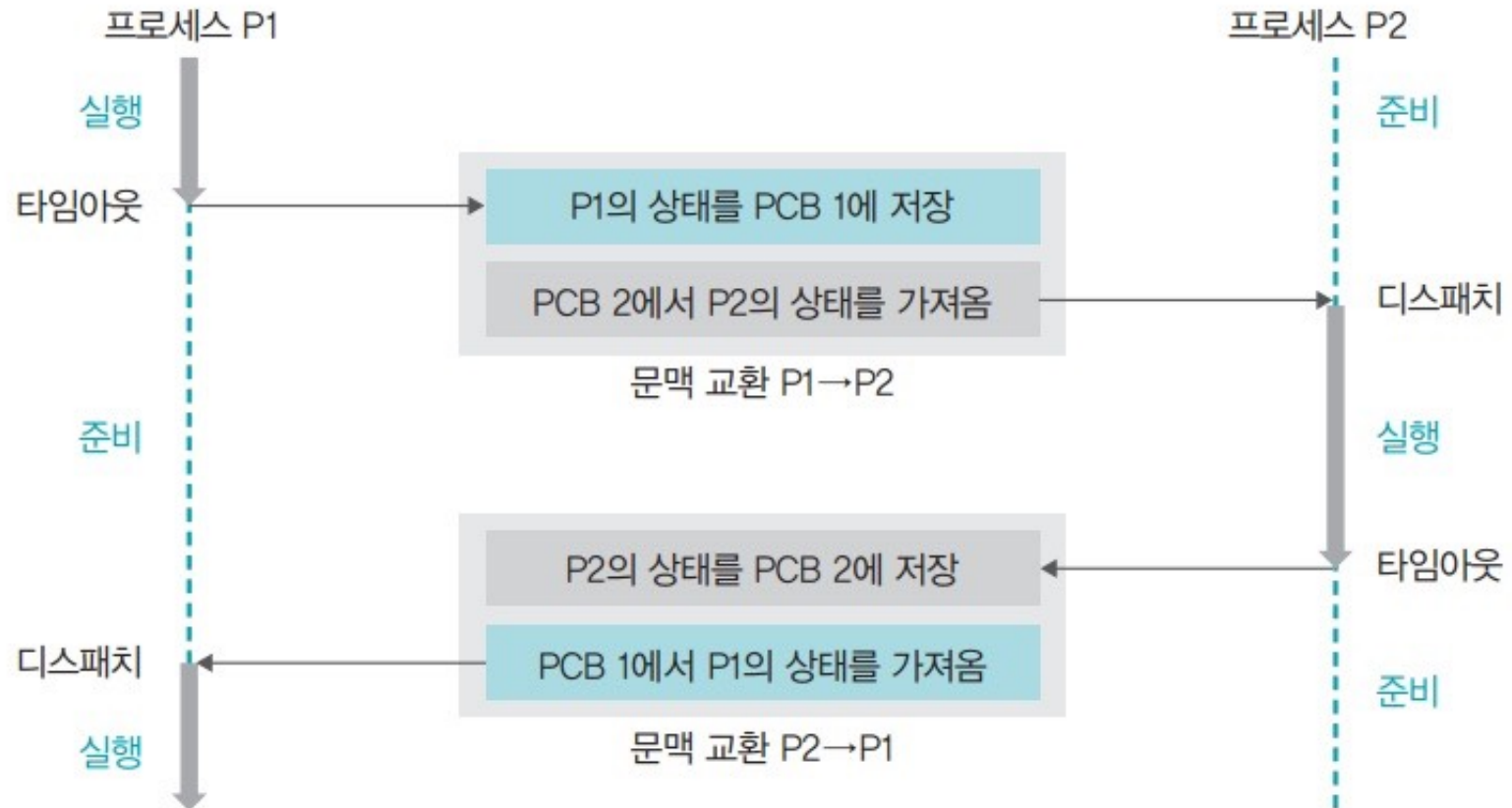


그림 3-15 문맥 교환 과정

3-1 프로세스의 구조

■ 프로세스의 구조

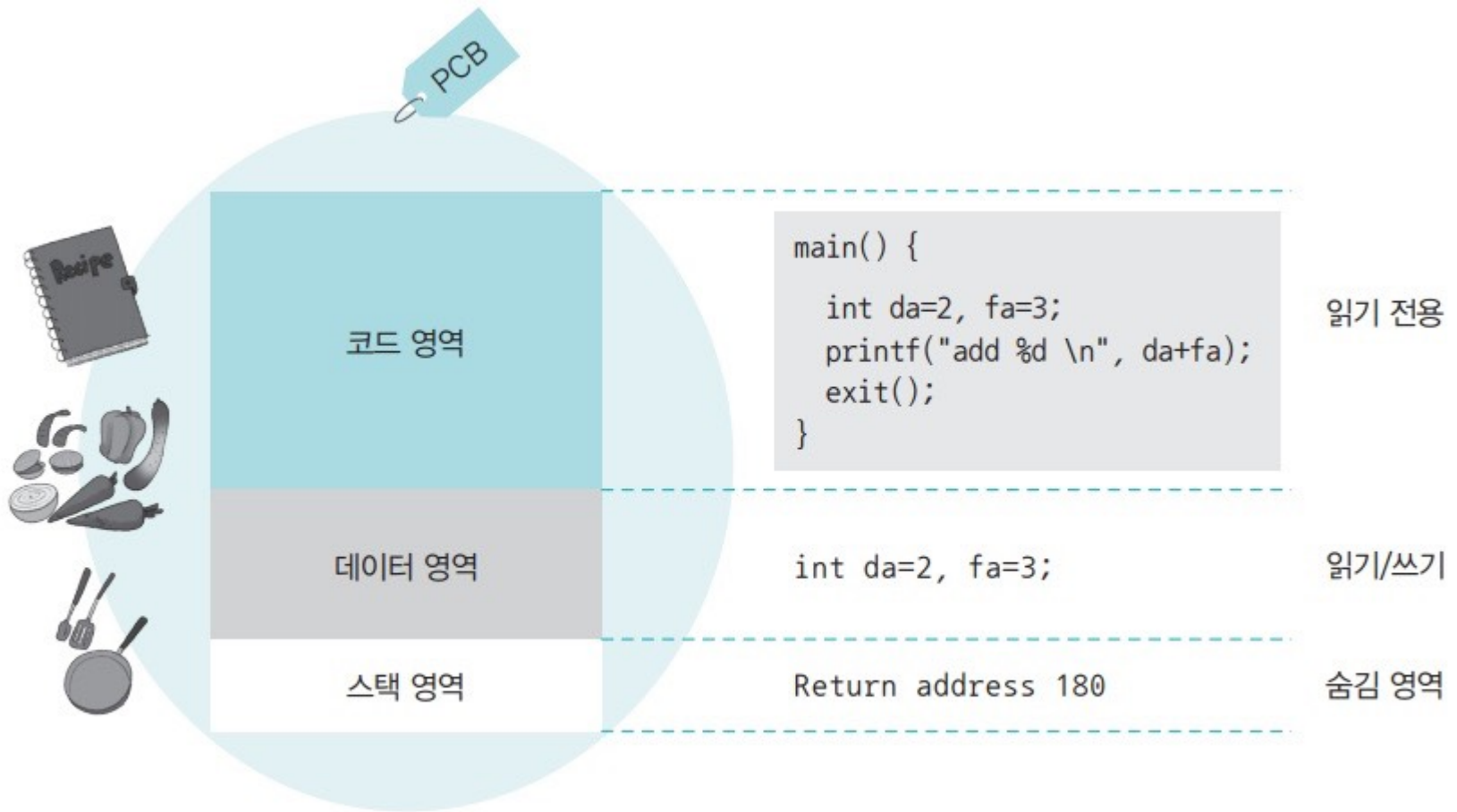


그림 3-16 프로세스의 구조

3-1 프로세스의 구조

■ 코드 영역

- 프로그램의 본문이 기술된 곳
- 프로그래머가 작성한 코드가 탑재되며 탑재된 코드는 읽기 전용으로 처리됨

■ 데이터 영역

- 코드가 실행되면서 사용하는 변수나 파일 등의 각종 데이터를 모아놓은 곳
- 데이터는 변하는 값이기 때문에 이곳의 내용은 기본적으로 읽기와 쓰기가 가능

■ 스택 영역

- 운영체제가 프로세스를 실행하기 위해 부수적으로 필요한 데이터를 모아놓은 곳
- 프로세스 내에서 함수를 호출하면 함수를 수행하고 원래 프로그램으로 되돌아올 위치를 이 영역에 저장
- 운영체제가 사용자의 프로세스를 작동하기 위해 유지하는 영역이므로 사용자에게는 보이지 않음

프로세스 생성

3-2 프로세스의 생성과 복사

■ `fork()` 시스템 호출의 개념

- 실행 중인 프로세스로부터 새로운 프로세스를 복사하는 함수
- 실행 중인 프로세스와 똑같은 프로세스가 하나 더 만들어짐

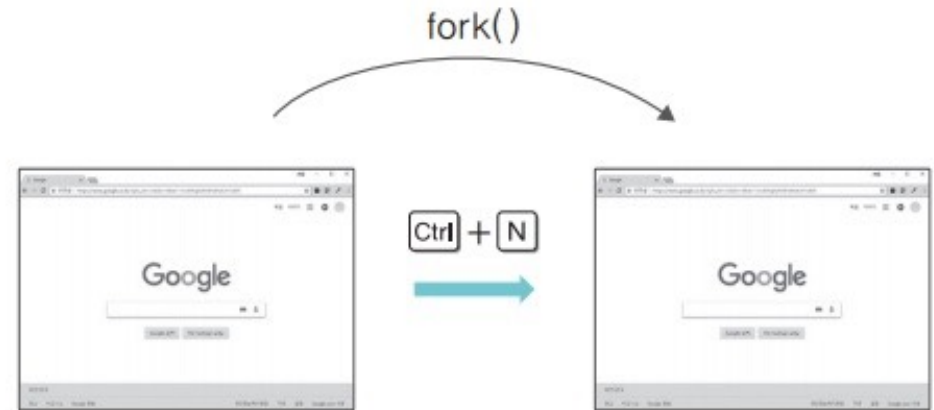
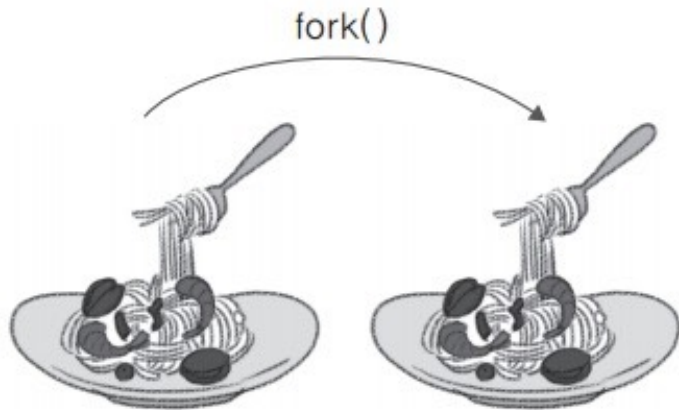


그림 3-17 `fork()` 시스템 호출의 개념

3-2 프로세스의 생성과 복사

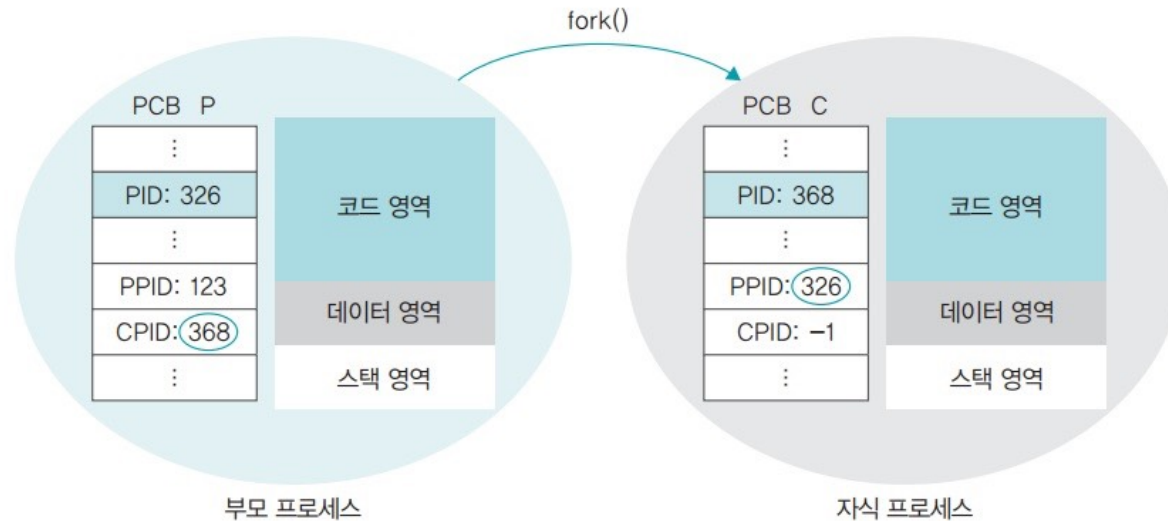
■ `fork()` 시스템 호출의 개념

정의 3-2 `fork()` 시스템 호출

`fork()` 시스템 호출은 실행 중인 프로세스를 복사하는 함수이다. 이때 실행하던 프로세스는 부모 프로세스, 새로 생긴 프로세스는 자식 프로세스로서 부모-자식 관계가 된다.

3-2 프로세스의 생성과 복사

■ fork() 시스템 호출의 동작 과정



- `fork()` 시스템 호출을 하면 프로세스 제어 블록을 포함한 부모 프로세스 영역의 대부분이 자식 프로세스에 복사되어 똑같은 프로세스가 만들어짐
- 단, 프로세스 제어 블록의 내용 중 다음이 변경됨
 - 프로세스 구분자
 - 메모리 관련 정보
 - 부모 프로세스 구분자와 자식 프로세스 구분자

3-2 프로세스의 생성과 복사

■ **fork() 시스템 호출의 장점**

- 프로세스의 생성 속도가 빠름
- 추가 작업 없이 자원을 상속할 수 있음
- 시스템 관리를 효율적으로 할 수 있음

■ **fork() 시스템 호출의 예 (그림 3-19)**

- 부모 프로세스의 코드가 실행되어 fork() 문을 만나면 똑같은 내용의 자식 프로세스를 하나 생성
- 이때 fork() 문은 부모 프로세스에 0보다 큰 값을 반환하고 자식 프로세스에 0을 반환
- 만약 0보다 작은 값을 반환하면 자식 프로세스가 생성되지 않은 것으로 여겨 'Error'를 출력

3-2 프로세스의 생성과 복사

■ fork() 시스템 호출의 예

```
#include <stdio.h>
#include <unistd.h>
```

```
void main()
{   int pid;
```

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

프로세스 복사

```
    if(pid<0) { printf("Error");
               exit(-1); }
```

```
    else if(pid==0) { printf("Child");
                     exit(0); }
```

```
    else { printf("Parent");
           exit(0); }
```

```
}
```

부모 프로세스

```
#include <stdio.h>
#include <unistd.h>
```

```
void main()
{   int pid;
```

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

```
    if(pid<0) { printf("Error");
               exit(-1); }
```

```
    else if(pid==0) { printf("Child");
                     exit(0); }
```

```
    else { printf("Parent");
           exit(0); }
```

```
}
```

자식 프로세스

그림 3-19 fork() 시스템 호출 코드의 예

3-3 프로세스의 전환

■ exec() 시스템 호출의 개념

- 기존의 프로세스를 새로운 프로세스로 전환(재사용)하는 함수
 - fork(): 새로운 프로세스를 복사하는 시스템 호출
 - exec(): 프로세스는 그대로 둔 채 내용만 바꾸는 시스템 호출

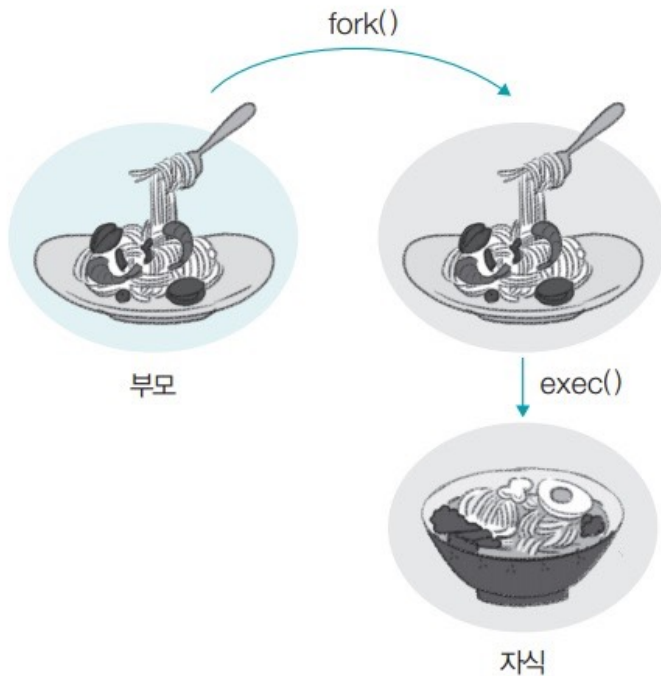


그림 3-20 exec() 시스템 호출의 개념

3-3 프로세스의 전환

정의 3-3 exec() 시스템 호출

exec() 시스템 호출은 이미 만들어진 프로세스의 구조를 재활용하는 것이다.

■ exec() 시스템 호출의 동작 과정 (그림 3-21)

- exec() 시스템 호출을 하면 코드 영역에 있는 기존의 내용을 지우고 새로운 코드로 바꿔버림
- 데이터 영역이 새로운 변수로 채워지고 스택 영역이 리셋
- 프로세스 제어 블록의 내용 중 프로세스 구분자, 부모 프로세스 구분자, 자식 프로세스 구분자, 메모리 관련 사항 등은 변하지 않지만 프로그램 카운터 레지스터 값을 비롯한 각종 레지스터와 사용한 파일 정보가 모두 리셋

3-3 프로세스의 전환

■ exec() 시스템 호출의 동작 과정

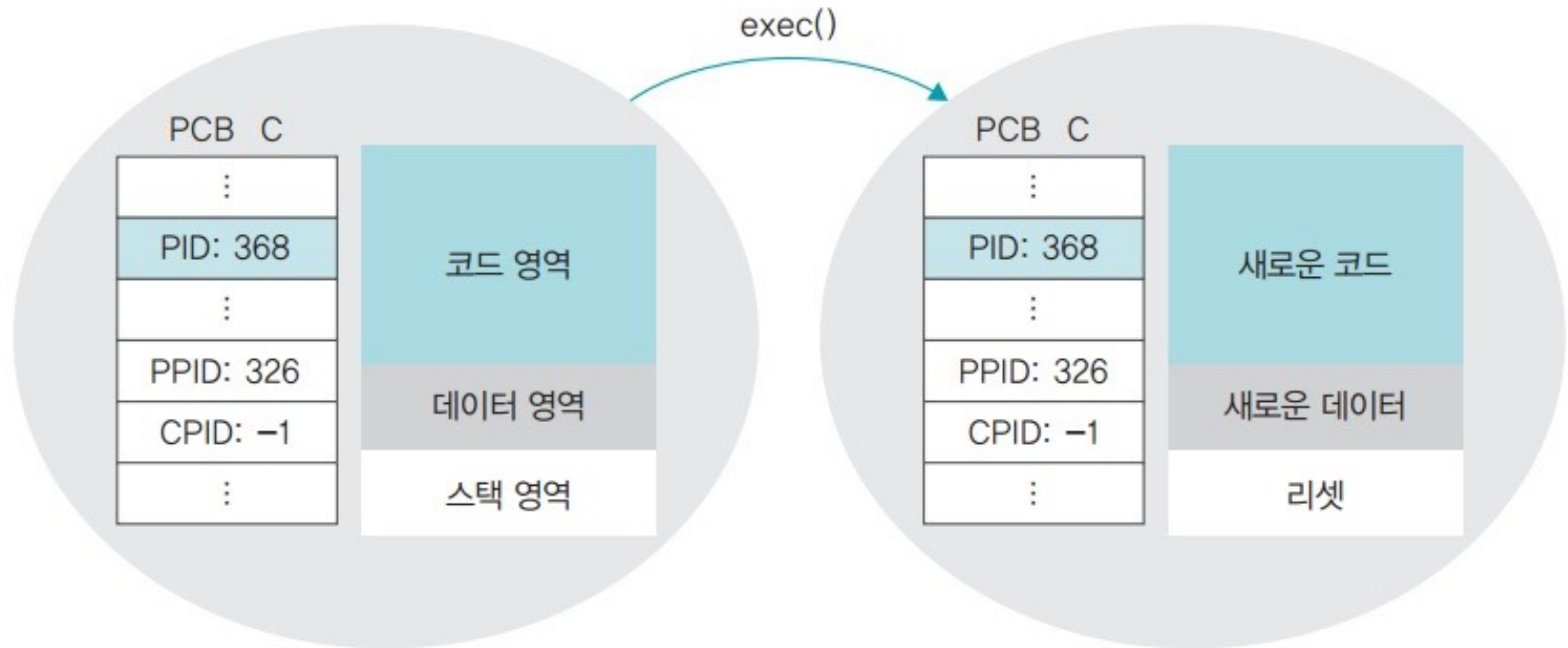


그림 3-21 exec() 시스템 호출 후 프로세스의 변화

3-3 프로세스의 전환

- **exec() 시스템 호출의 예 (그림 3-22)**
 - 부모 프로세스의 fork() 문을 실행하여 자식 프로세스를 생성하고, wait() 문을 실행하여 자식 프로세스가 끝날 때까지 기다림
 - 새로 생성된 자식 프로세스는 부모 프로세스의 코드와 같음
 - exec() 시스템 호출을 사용하여 새로운 프로세스로 전환하더라도 프로세스 제어 블록의 각종 프로세스 구분자(PID, PPID, CPID)가 변경되지 않기 때문에, 프로세스가 종료된 후 부모 프로세스로 돌아올 수 있음

3-3 프로세스의 전환

■ exec() 시스템 호출의 예

```
#include <stdio.h>
#include <unistd.h>

void main()
{   int pid;

    pid=fork();

    if(pid<0) { printf("Error");
               exit(-1); }

    else if(pid==0) { /* child process */
                     execlp("mplayer", "mplayer", NULL);
                     exit(0); }

    else { wait(NULL);
           printf("mplayer Terminated");
           exit(0); }
}
```

부모 프로세스

```
/* mplayer - music player */
:
void main()
{
:
:
mplayer
실행 코드
:
}
```

자식 프로세스(exec() 실행)

그림 3-22 exec() 시스템 호출 코드의 예

프로세스 계층

3-4 프로세스의 계층 구조

■ 유닉스의 프로세스 계층 구조

- 유닉스의 모든 프로세스는 init 프로세스의 자식이 되어 트리 구조를 이룸

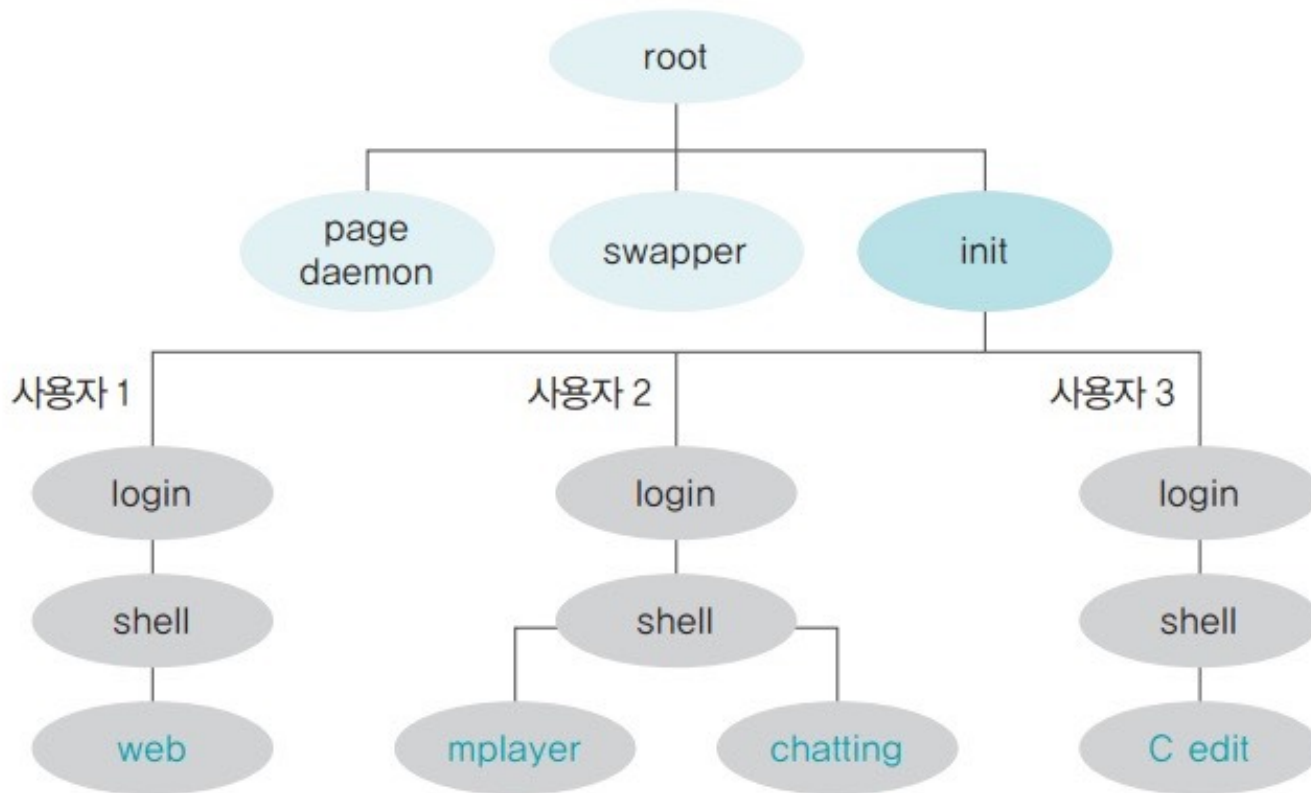


그림 3-23 유닉스의 프로세스 계층 구조

3-4 프로세스의 계층 구조

■ 프로세스 계층 구조의 장점

- 여러 작업을 동시에 처리할 수 있다.

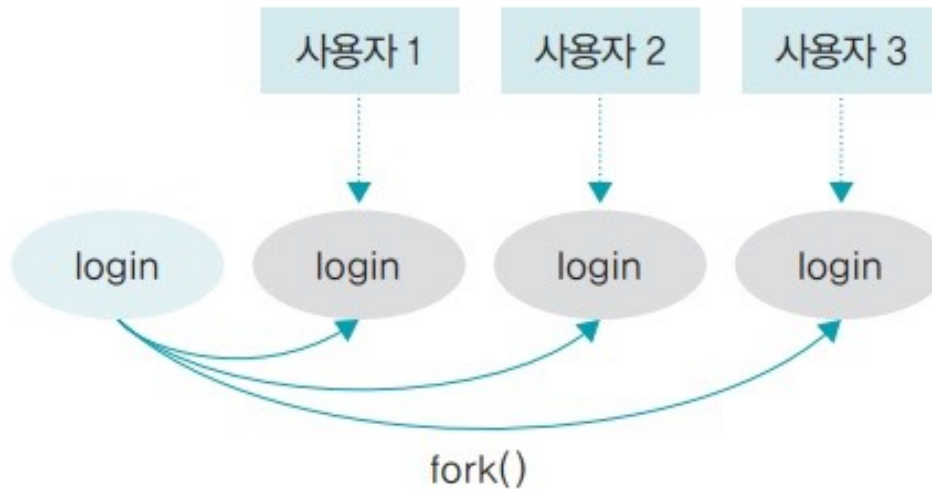


그림 3-24 여러 사용자를 동시 처리

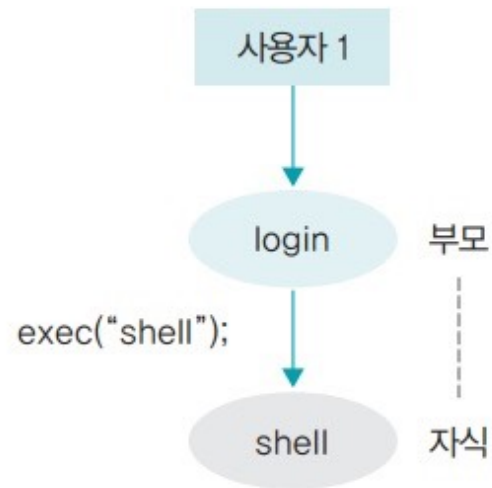
3-4 프로세스의 계층 구조

■ 프로세스 계층 구조의 장점

- 프로세스의 재사용이 용이하다.



그림 3-25 프로세스의 재사용



- 자원 회수가 쉽다.

- 프로세스를 계층 구조로 만들면 프로세스 간의 책임 관계가 분명해져서 시스템을 관리하기가 수월

3-4 프로세스의 계층 구조

■ 미아 프로세스

- 프로세스가 종료된 후에도 비정상적으로 남아 있는 프로세스
- C 언어의 `exit()` 또는 `return()` 문은 자식 프로세스가 작업이 끝났음을 부모 프로세스에 알리는 것으로 미아 프로세스 발생을 미연에 방지함

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

그림 3-26 exit 함수

스레드 개념

4-1 스레드의 개념

■ 스레드의 정의

- CPU 스케줄러가 CPU에 전달하는 일 하나
- CPU가 처리하는 작업의 단위는 프로세스로부터 전달받은 스레드
 - 운영체제 입장에서의 작업 단위는 프로세스
 - CPU 입장에서의 작업 단위는 스레드

정의 3-4 스레드

프로세스의 코드에 정의된 절차에 따라 CPU에 작업 요청을 하는 실행 단위이다.

4-1 스레드의 개념

■ 프로세스와 스레드의 차이

- 프로세스끼리는 약하게 연결되어 있는 반면 스레드끼리는 강하게 연결되어 있음

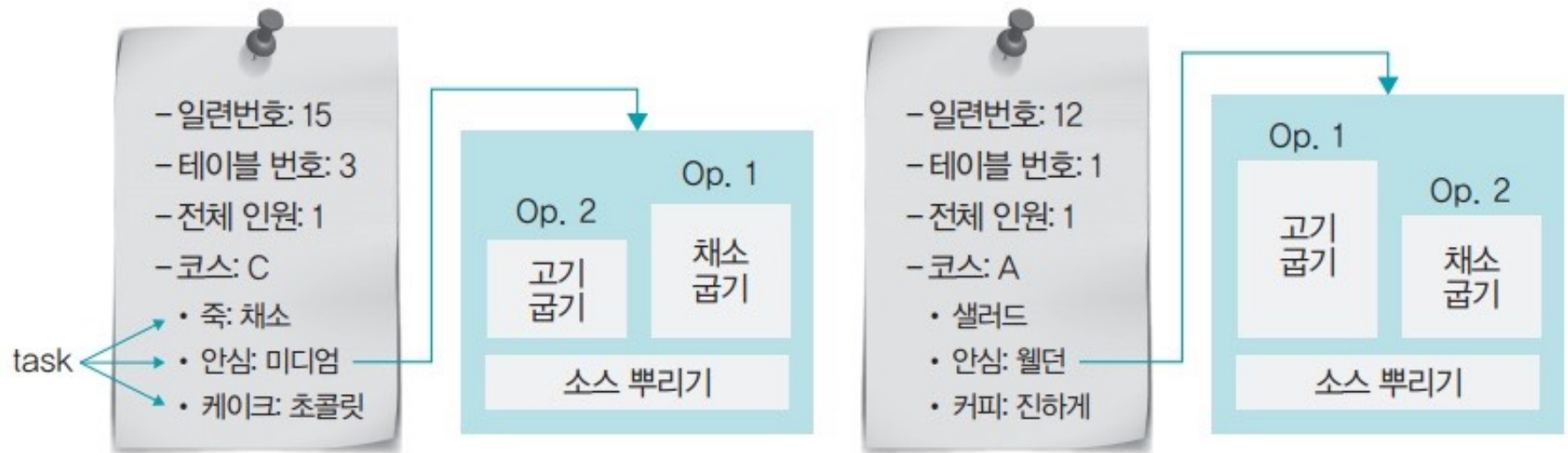
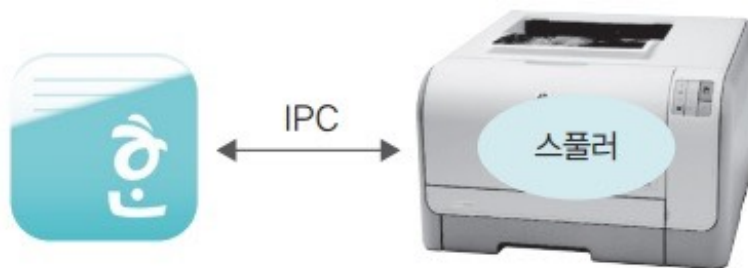


그림 3-27 작업과 일의 차이

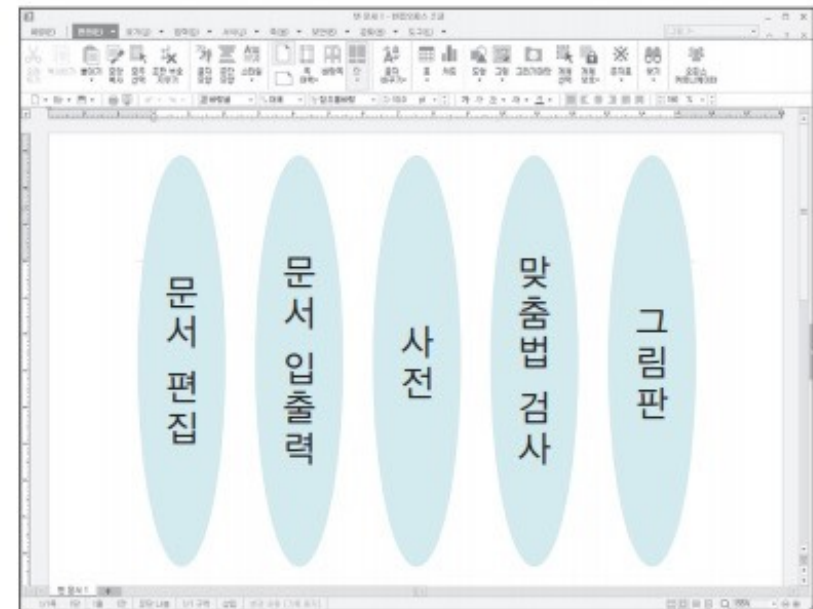
4-1 스레드의 개념

■ 멀티태스킹과 멀티스레드의 차이

- **멀티태스킹** : 여러 개의 프로세스로 구성된 것
- **멀티스레드** : 하나의 프로세스에 여러 개의 스레드로 구성된 것



(a) 멀티태스킹



(b) 멀티스레드

그림 3-28 멀티태스킹과 멀티스레드의 차이

4-1 스레드의 개념

■ 멀티스레드

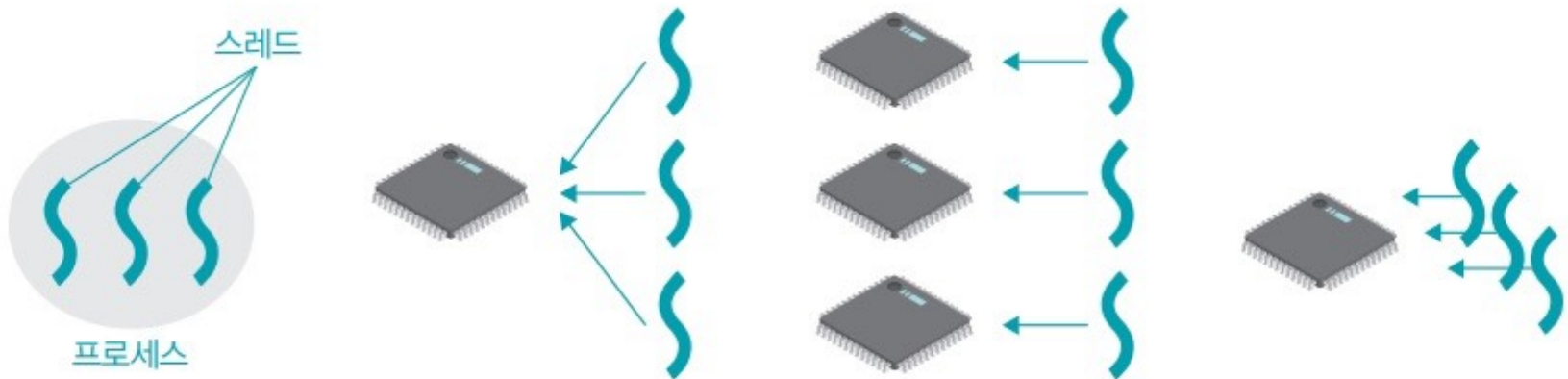
- 프로세스 내 작업을 여러 개의 스레드로 분할함으로써 작업의 부담을 줄이는 프로세스 운영 기법

■ 멀티태스킹

- 운영체제가 CPU에 작업을 줄 때 시간을 잘게 나누어 배분하는 기법

■ 멀티프로세싱

- CPU를 여러 개 사용하여 여러 개의 스레드를 동시에 처리하는 작업 환경



(a) 멀티스레드

(b) 멀티태스킹(시간 공유)

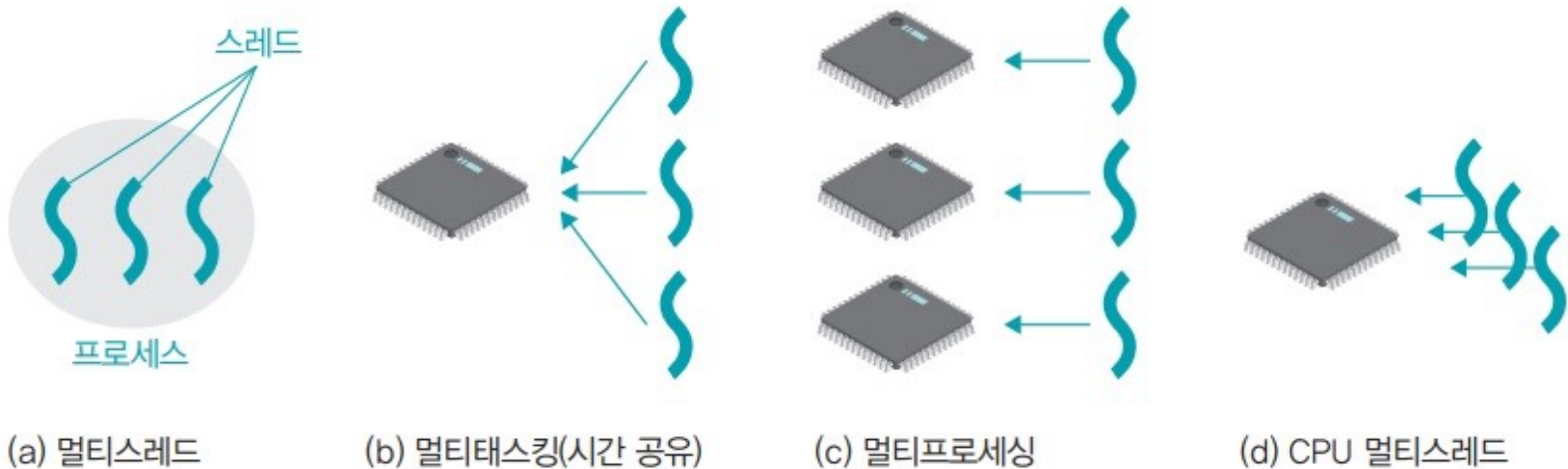
(c) 멀티프로세싱

(d) CPU 멀티스레드

4-1 스레드의 개념

■ CPU 멀티스레드

- 한 번에 하나씩 처리해야 하는 스레드를 파이프라인 기법을 이용하여 동시에 여러 스레드를 처리하도록 만든 병렬 처리 기법
 - **멀티스레드** : 운영체제가 소프트웨어적으로 프로세스를 작은 단위의 스레드로 분할하여 운영하는 기법
 - **CPU 멀티스레드** : 하드웨어적인 방법으로 하나의 CPU에서 여러 스레드를 동시에 처리하는 병렬 처리 기법



4-2 멀티스레드의 구조와 예

■ 멀티태스킹의 낭비 요소

- `fork()` 시스템 호출로 프로세스를 복사하면 코드 영역과 데이터 영역의 일부가 메모리에 중복되어 존재하며, 부모-자식 관계이지만 서로 독립적인 프로세스이므로 이러한 낭비 요소를 제거할 수 없음

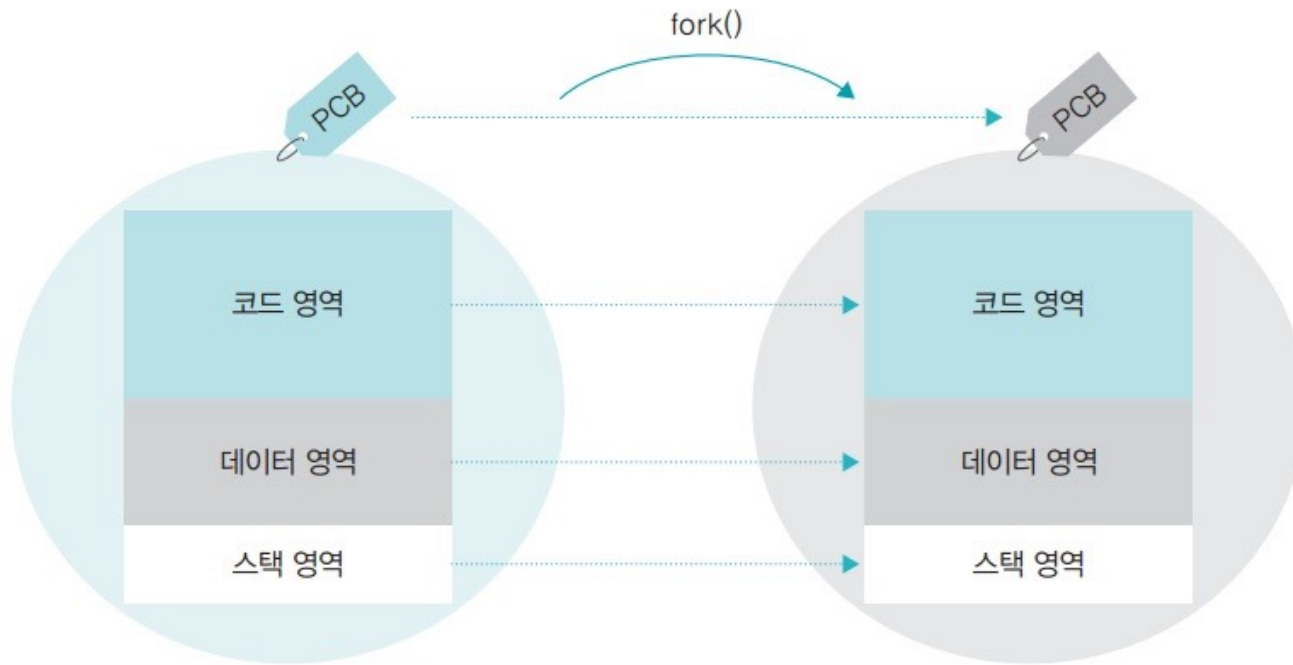


그림 3-30 멀티태스킹의 낭비 요소

4-2 멀티스레드의 구조와 예

■ 멀티태스킹과 멀티스레드의 차이

- fork() 시스템 호출로 여러 개의 프로세스를 만들면 필요 없는 정적 영역이 여러 개가 됨
- 멀티스레드는 코드, 파일 등의 자원을 공유함으로써 자원의 낭비를 막고 효율성 향상

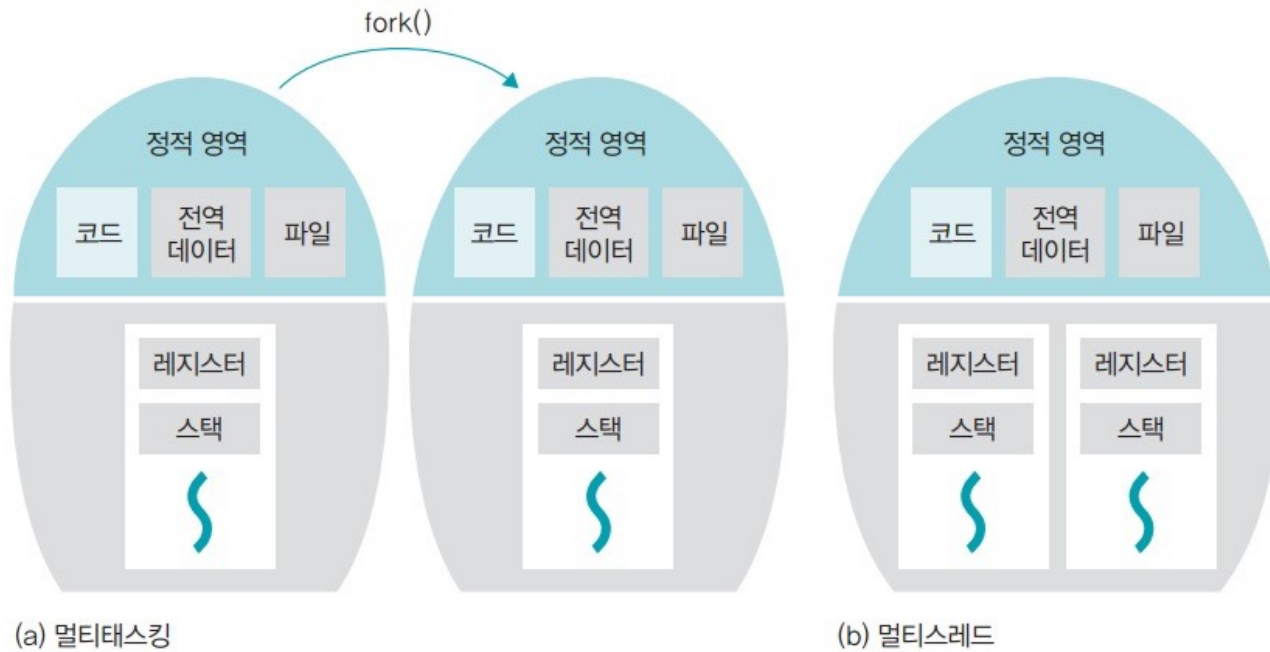


그림 3-31 멀티태스킹과 멀티스레드의 구조

4-2 멀티스레드의 구조와 예

■ 자바 스레드 코드의 예

- main{ } 위쪽에 있는 class TH_test extends Thread{ }는 스레드 객체로, 이는 TH_test 객체를 확장하여 스레드를 만들
- TH_test 객체의 run() 부분을 스레드로 만들어 실행하며 'Th_Test'를 백 번 출력

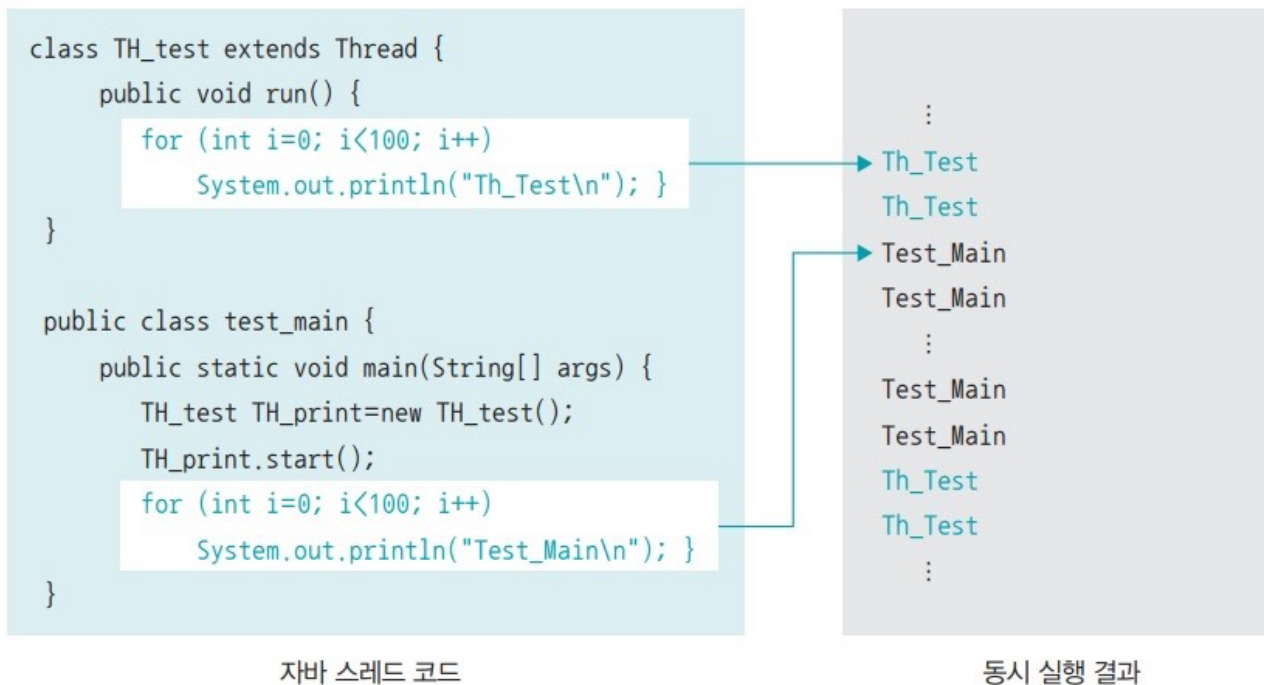


그림 3-32 자바 스레드 코드의 예

4-2 멀티스레드의 구조와 예

■ fork() 시스템 호출로 작성한 코드 예

- 프로세스 제어 블록, 코드, 데이터 등이 모두 2배가 됨으로써 스레드를 사용하는 것보다 낭비가 심함

```
include <stdio.h>
include <unistd.h>
```

```
int main()
```

```
    int i=0, pid;
```

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

```
    if(pid<0) { printf("Error");
                exit(-1); }
```

```
    else if(pid==0) { for(i=0; i<100; i++)
                        printf("Th_Test\n");
                      exit(0); }
```

```
    else { for(i=0; i<100; i++) /* parent */
            printf("Test_Main\n");
            exit(0); }
```

부모 프로세스

```
#include <stdio.h>
#include <unistd.h>
```

```
int main()
```

```
{    int i=0, pid;
```

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

```
    if(pid<0) { printf("Error");
                exit(-1); }
```

```
    else if(pid==0) { for(i=0; i<100; i++)
                        printf("Th_Test\n");
                      exit(0); }
```

```
    else { for(i=0; i<100; i++) /* parent */
            printf("Test_Main\n");
            exit(0); }
```

```
}
```

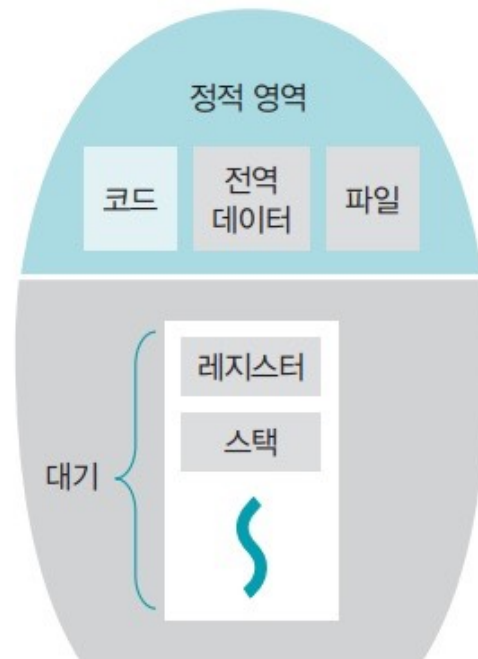
자식 프로세스

| 3-33 멀티태스킹 코드의 예

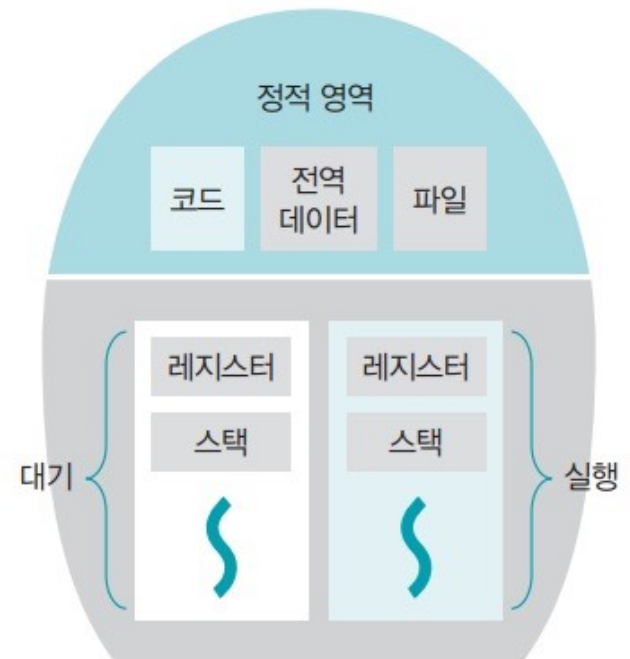
4-3 멀티스레드의 장단점

■ 멀티스레드의 장점

- 응답성 향상
- 자원 공유
- 효율성 향상
- 다중 CPU 지원



(a) 단일 스레드



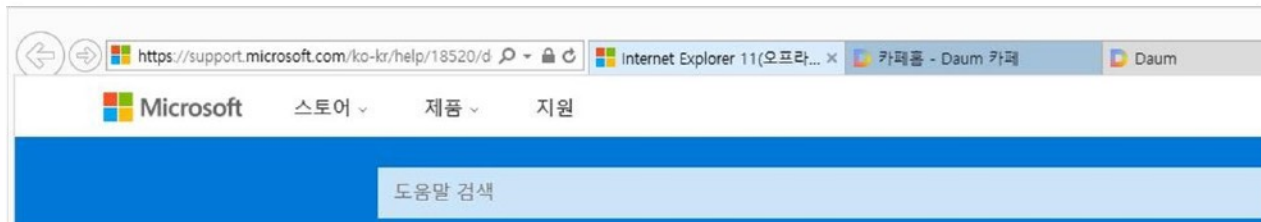
(b) 멀티스레드

그림 3-34 단일 스레드와 멀티스레드의 구조

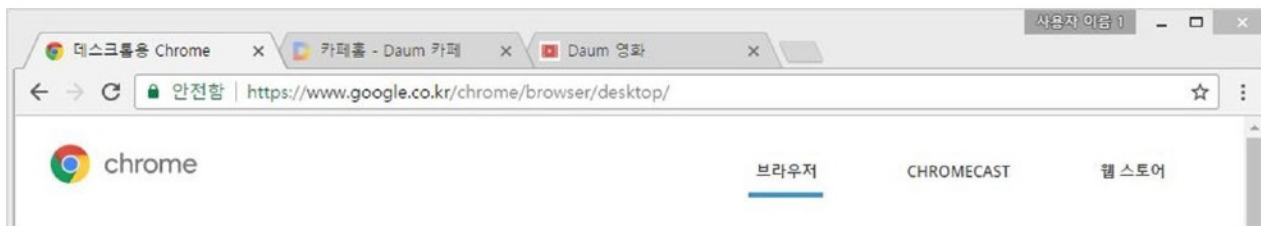
4-3 멀티스레드의 장단점

■ 멀티스레드의 단점

- 모든 스레드가 자원을 공유하기 때문에 한 스레드에 문제가 생기면 전체 프로세스에 영향을 미침
- 인터넷 익스플로러에서 여러 개의 화면을 동시에 띄웠는데 그중 하나에 문제가 생기면 인터넷 익스플로러 전체가 종료



(a) 멀티스레드를 이용하는 인터넷 익스플로러



(b) 멀티태스킹을 이용하는 크롬

그림 3-35 멀티탭 기능 구현 방식의 차이

4-4 멀티스레드 모델

- **커널 스레드와 사용자 스레드**
 - **커널 스레드** : 커널이 직접 생성하고 관리하는 스레드
 - **사용자 스레드** : 라이브러리에 의해 구현된 일반적인 스레드

4-4 멀티스레드 모델

■ 사용자 레벨 스레드

- 사용자 프로세스 내에 여러 개의 스레드가 커널의 스레드 하나와 연결(1 to N 모델)
- 라이브러리가 직접 스케줄링을 하고 작업에 필요한 정보를 처리하기 때문에 문맥 교환이 필요 없음
- 커널 스레드가 입출력 작업을 위해 대기 상태에 들어가면 모든 사용자 스레드가 같이 대기하게 됨
- 한 프로세스의 타임 슬라이스를
여러 스레드가 공유하기 때문에
여러 개의 CPU를 동시에 사용할
수 없음

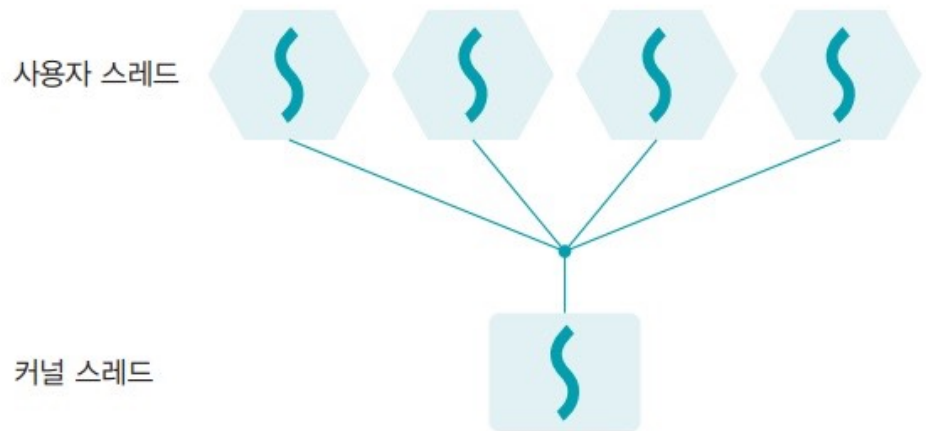


그림 3-36 사용자 레벨 스레드(1 to N 모델)

4-4 멀티스레드 모델

■ 커널 레벨 스레드

- 하나의 사용자 스레드가 하나의 커널 스레드와 연결(1 to 1 모델)
- 독립적으로 스케줄링이 되므로 특정 스레드가 대기 상태에 들어가도 다른 스레드는 작업을 계속할 수 있음
- 커널 레벨에서 모든 작업을 지원하기 때문에 멀티 CPU를 사용할 수 있음
- 하나의 스레드가 대기 상태에 있어도 다른 스레드는 작업을 계속할 수 있음
- 커널의 기능을 사용하므로 보안에 강하고 안정적으로 작동
- 문맥 교환할 때 오버헤드 때문에 느리게 작동

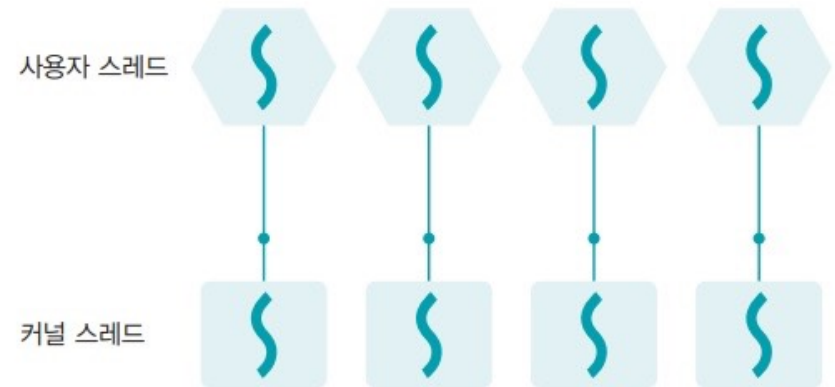


그림 3-37 커널 레벨 스레드(1 to 1 모델)

4-4 멀티스레드 모델

■ 멀티레벨 스레드

- 사용자 레벨 스레드와 커널 레벨 스레드를 혼합한 방식(M to N 모델)
- 커널 스레드가 대기 상태에 들어가면 다른 커널 스레드가 대신 작업을 하여 사용자 레벨 스레드보다 유연하게 작업을 처리할 수 있음
- 커널 레벨 스레드를 같이 사용하기 때문에 여전히 문맥 교환 시 오버헤드가 있어 사용자 레벨 스레드만큼 빠르지 않음
- 빠르게 움직여야 하는 스레드는 사용자 레벨 스레드로 작동하고, 안정적으로 움직여야 하는 스레드는 커널 레벨 스레드로 작동

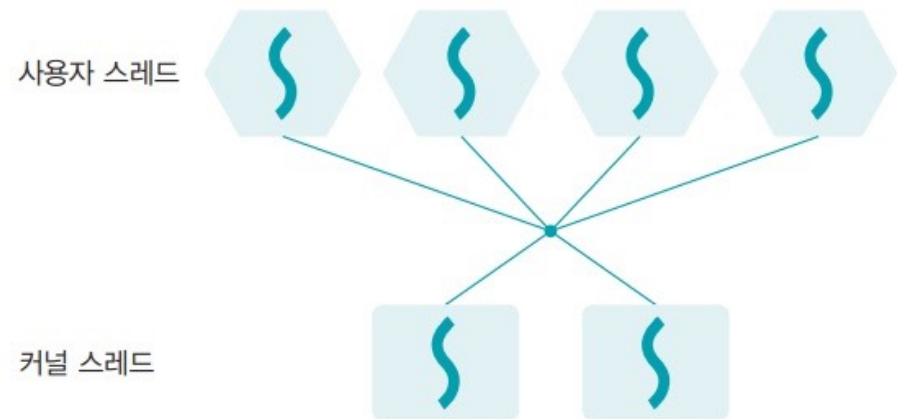


그림 3-38 멀티레벨 스레드(M to N 모델)

동적할당 영역

심화학습 5-1 프로세스의 동적 할당 영역

■ 프로세스의 구조

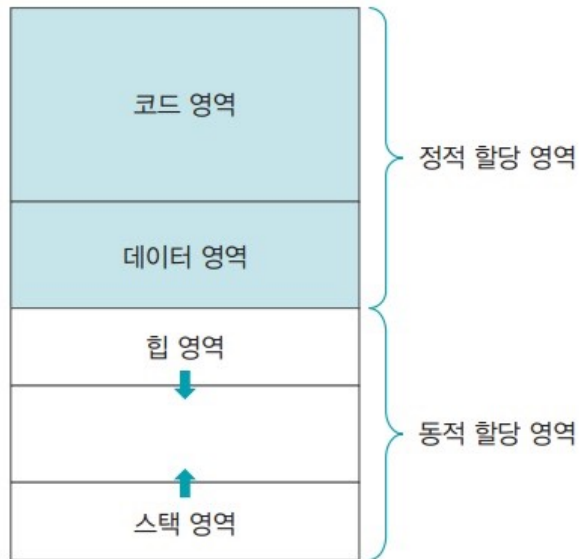


그림 3-39 프로세스의 상세 구조

- **코드 영역** : 프로그램의 본체가 있는 곳
- **데이터 영역** : 프로그램이 사용하려고 정의한 변수와 데이터가 있는 곳
- **스택 영역과 힙 영역** : 프로세스가 실행되는 동안 만들어지는 영역으로, 그 크기가 늘어났다 줄어든
기도 하는 동적 할당 영역

5-1 프로세스의 동적 할당 영역

■ 스택 영역

- 스레드가 작동하는 동안 추가되거나 삭제되는 동적 할당 영역
- 스레드가 진행됨에 따라 커지기도 하고 작아지기도 함

```

int main() {
    int a=1, b=2;
    printf("main %d %d\n", a b);
    add(a, b);
    ❶ → exit(); }

add(int c, int d) {
    ❷ → mul(c, d);
    printf("add %d\n", c+d); }

mul(int e, int f) {
    printf("mul %d\n", e*f); }
  
```

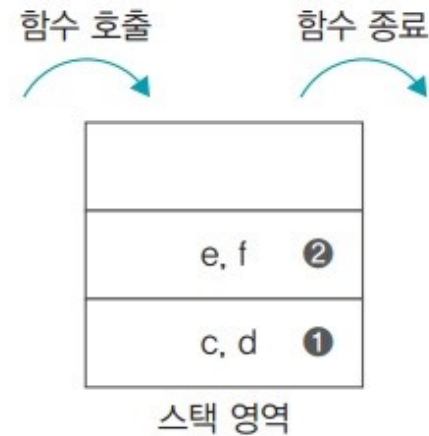


그림 3-40 함수 호출과 스택

5-1 프로세스의 동적 할당 영역

■ 힙 영역

- 프로그램이 실행되는 동안 할당되는 변수 영역
- 포인터, malloc() 함수, calloc() 함수 등은 메모리를 효율적으로 사용하기 위해 만들어진 것으로 어쩌다 한 번 쓰는 큰 배열을 처음부터 선언하고 끝까지 놔두는 일이 없어야 함

```
main() {  
    int sarr[50];  
    int *darr;  
  
    darr=(int*)malloc(sizeof(int)*50);  
  
    free(darr);  
}
```

그림 3-41 malloc() 함수 코드의 예

5-2 exit ()와 wait () 시스템 호출

■ exit() 시스템 호출

- 작업의 종료를 알려주는 시스템 호출
- exit() 함수를 선언함으로써 부모 프로세스는 자식 프로세스가 사용하던 자원을 빨리 거둬 갈 수 있음
- exit() 함수는 전달하는 인자를 확인하여 자식 프로세스가 어떤 상태로 종료되었는지를 알려주는데, 인자가 0이면 정상 종료이고 -1이면 비정상 종료

```
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{   int pid;

    pid=fork();

    if(pid<0) { printf("Error");
                exit(-1); }

    else if(pid==0) { printf("Child");
                      exit(0); }

    else { printf("Parent");
           exit(0); }
}
```

부모 프로세스

```
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{   int pid;

    pid=fork();

    if(pid<0) { printf("Error");
                exit(-1); }

    else if(pid==0) { printf("Child");
                      exit(0); }

    else { printf("Parent");
           exit(0); }
}
```

자식 프로세스

그림 3-44 fork() 코드의 예

5-3 프로세스의 동적 할당 영역

■ wait() 시스템 호출

- 자식 프로세스가 끝나기를 기다렸다가 자식 프로세스가 종료되면 다음 문장을 실행하는 시스템 호출
- 부모 프로세스와 자식 프로세스 간 동기화에도 사용

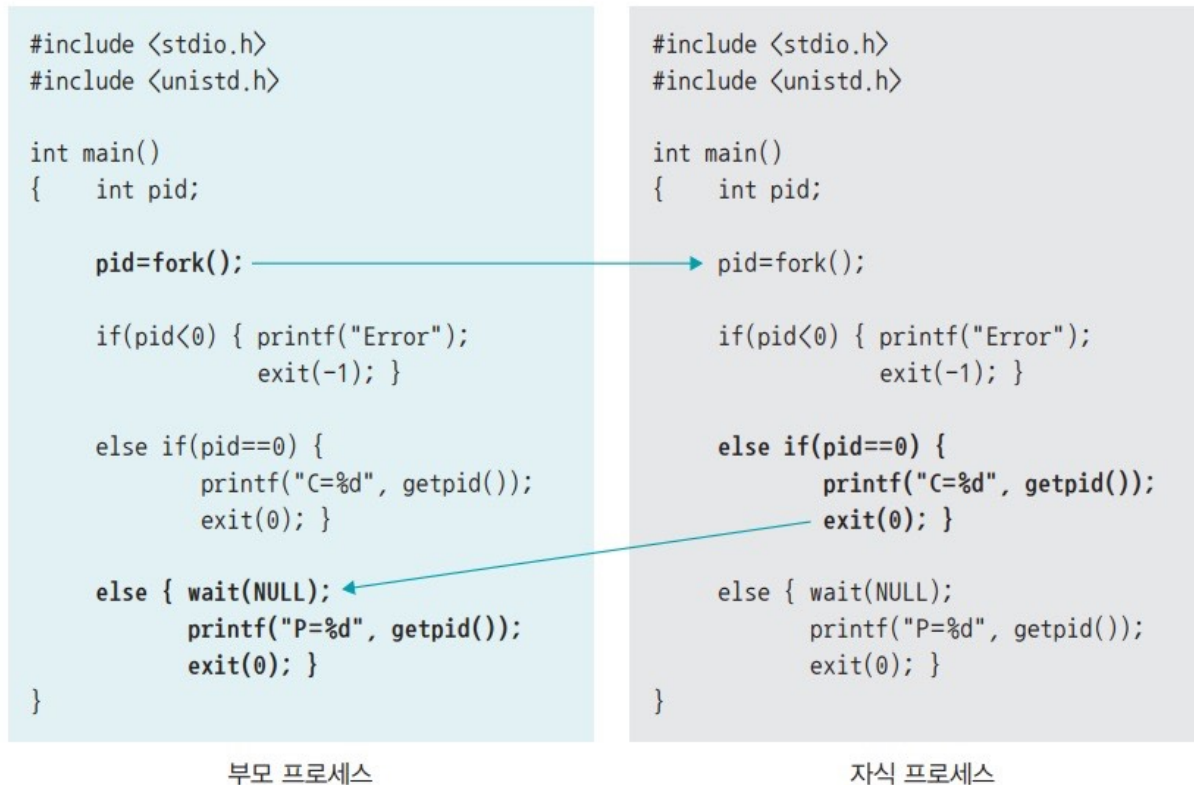


그림 3-45 wait() 시스템 호출이 포함된 fork() 코드

5-3 프로세스의 동적 할당 영역

■ Wait() 함수의 응용

- 전면 프로세스에서는 셸이 wait() 함수를 사용하기 때문에 자식 프로세스가 끝날 때까지 다음 명령어를 입력받을 수 없음
- 후면 프로세스에서는 셸이 wait() 함수를 사용하지 않기 때문에 다음 명령어를 받아들일 수 있음

```
01 sleep 100    // 전면 프로세스  
02 sleep 100&   // 후면 프로세스
```



Thank You
