스마트시스템 운영체제
(LD01600)

김준철

정보시스템공학과

greensday@sungshin.ac.kr

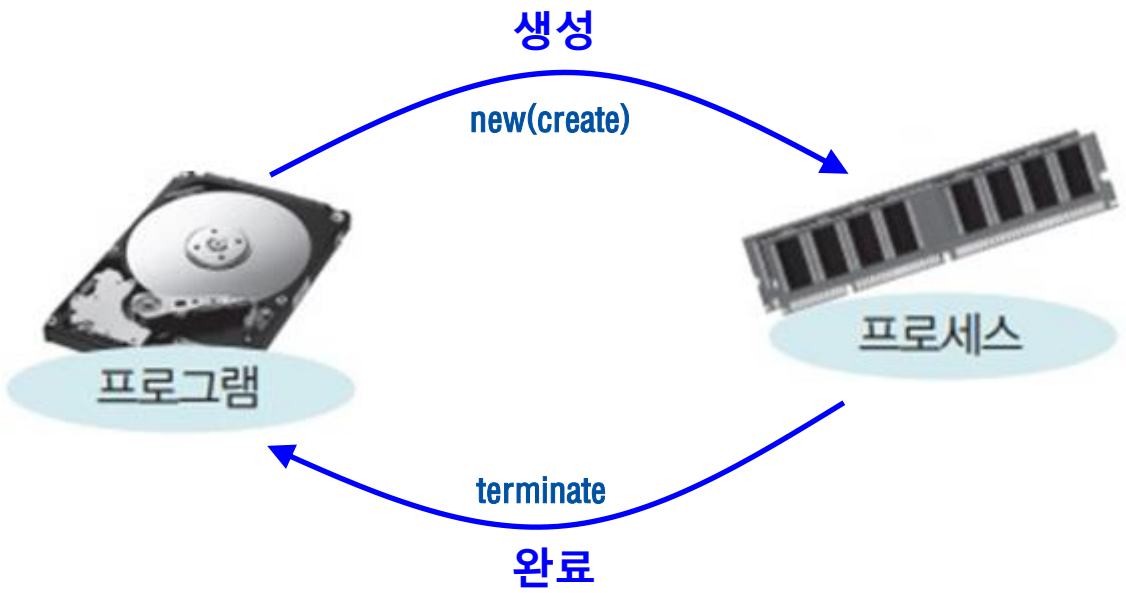
3주차 강의

	주차	강의 목차
9.2	1	과목소개 / 운영체제 개요
9.9	2	컴퓨터 시스템 구조
9.16	3	**Process와 thread1**
9.22	4	Process와 thread2, CPU스케쥴링1
휴강(9.30) (추석)		
10.7	5	CPU스케쥴링2
10.14	6	Process 동기화
10.21	7	교착 상태
10.28	8	중간고사
11.4	9	물리 메모리 관리
11.11	10	가상메모리 기초
11.18	11	가상메모리 관리
11.25	12	입출력시스템1
12.2	13	입출력시스템2, 파일시스템1
12.9	14	파일시스템2
12.16	15	기말고사

ch.03 Process와 thread1

Process의 개념

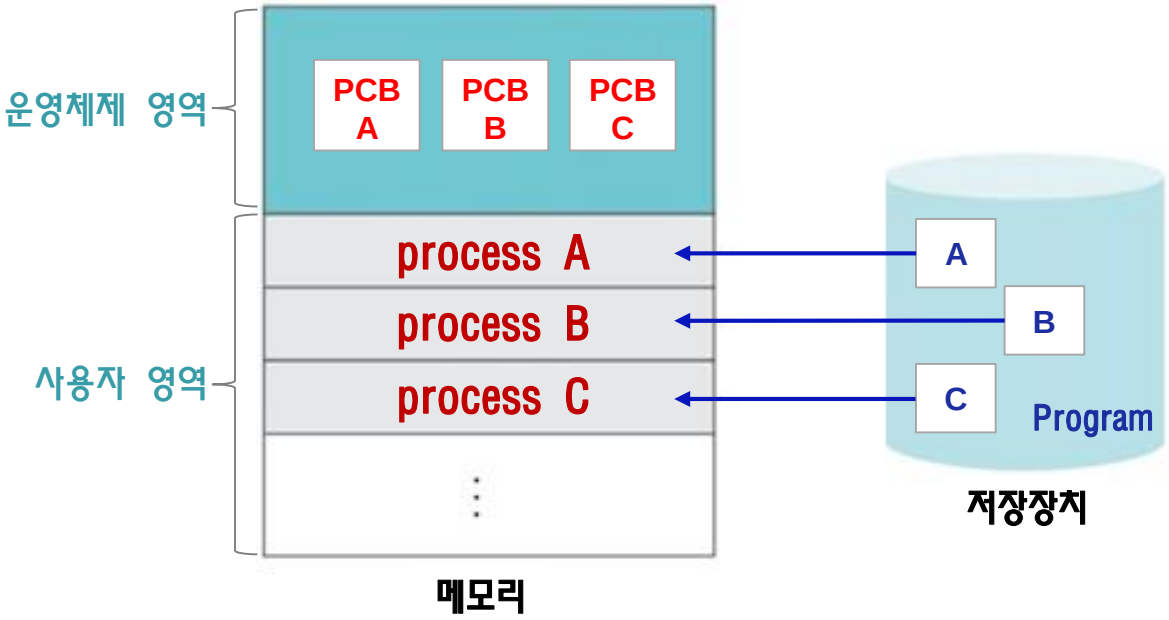
- Program
  - 저장장치에 저장되어 있는 정적인 상태
- Process
  - 실행을 위해 메모리에 올라온 동적인 상태



[그림] Program과 Process

Program에서 Process로의 전환

- Process control Block(PCB, process 제어 블록)
  - 운영체제가 해당 process를 관리하기 위한 자료 구조
    - **Process ID(PID, process 구분자)** : 각 process를 구분하는 구분자
    - **메모리 관련 정보** : process의 메모리 위치 정보 (base, limit 정보)
    - **각종 중간값** : process가 사용했던 중간값
      (process 상태, PC, registers 정보, CPU동작 시간, 열어본 file들 정보)



[그림] Program이 메모리에 올라와 Process가 되는 과정

Program에서 Process로의 전환

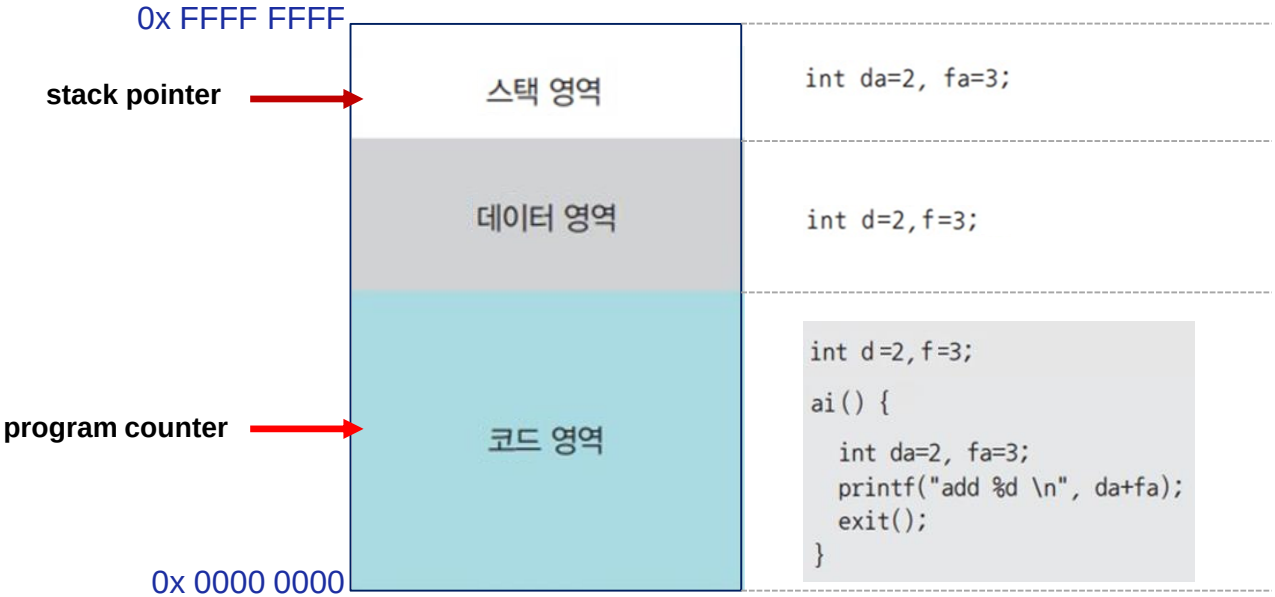
- Process와 Program의 관계
  - Program이 **Process가 된다는 것**은 운영체제로부터 **PCB를 얻는다**는 뜻
  - **Process가 종료된다는 것**은 **해당 PCB가 폐기된다**는 뜻

[정의] Process와 Program의 관계

```
process = program + PCB
program = process - PCB
```

Process의 구조

- Process의 구조



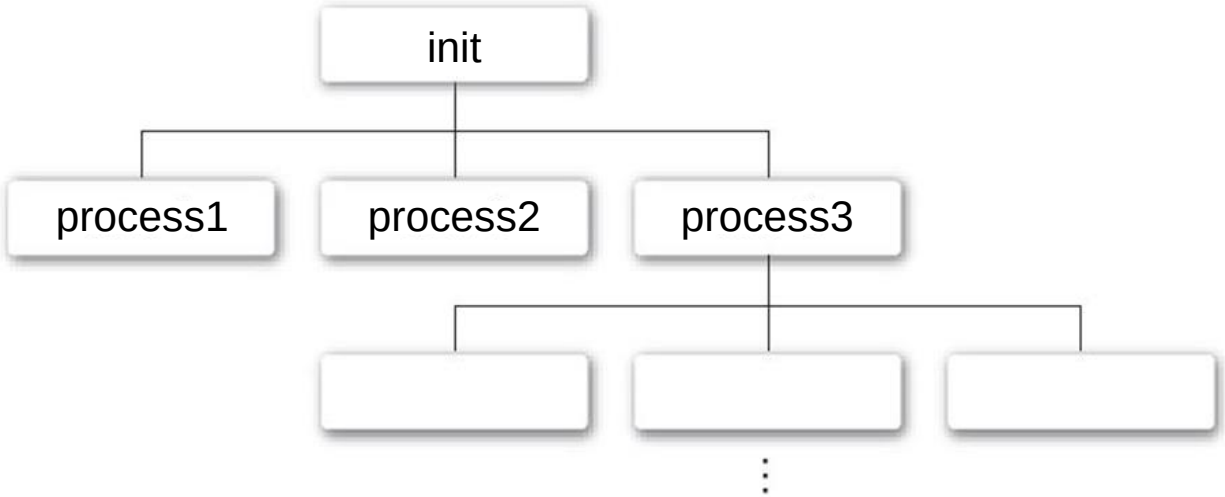
[그림] Process의 메모리 구조

Process의 구조

- **코드(code) 영역**
  - **Program의 본문**이 기술된 곳
  - 프로그래머가 작성한 **코드**가 탑재되며 탑재된 코드는 **읽기 전용**으로 처리됨
- **데이터(data) 영역**
  - 코드가 실행되면서 사용하는 **전역변수 등의 데이터를 모아 놓은 곳**
  - **데이터는 변하는 값**이므로 data영역의 내용은 기본적으로 **읽기와 쓰기가 가능**
- **스택(stack) 영역**
  - 운영체제가 process를 실행하기 위해 부수적으로 필요한 **지역 변수등의 일시적인 데이터를 모아놓은 곳**
  - process 내에서 **함수를 호출**하면 함수를 수행하고 원래 program으로 되돌아올 위치를 이 영역에 저장
  - 운영체제가 사용자의 process를 작동하기 위해 유지하는 영역

Process의 생성 및 계층구조

- **Process의 생성 및 계층 구조**
  - 처음 process는 booting시 OS가 생성 (**init process**)
  - Process 실행 중 **process 생성 system call**을 이용하여 **새로운 process 생성**
    fork()
  - Process 생성 순서를 저장, **부모-자식 관계** 유지하여 **계층적 생성**
  - 생성하는 process – **parent process**부모 process
    생성되는 process – **child process**자식 process



[그림] Process의 계층 구조의 예

Process의 생성과 복사

- fork( ) system call의 개념
  - 실행 중인 process로부터 새로운 process를 복사하는 함수
  - 실행 중인 process와 똑같은 process가 하나 더 만들어짐

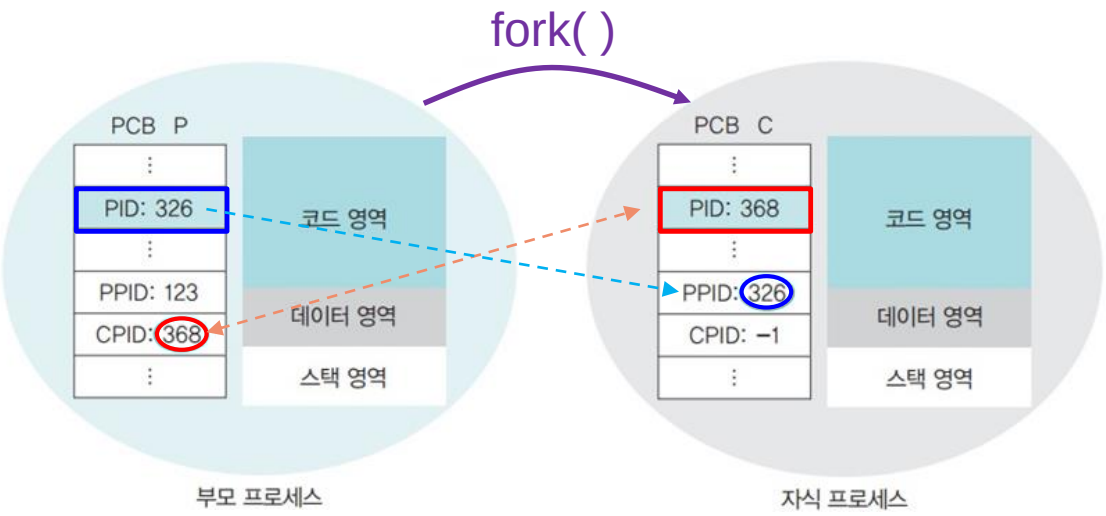
[그림] fork() system call의 개념

[정의] fork() system call

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

Process의 생성과 복사

- fork( ) system call의 동작 과정



[그림] fork() system call 후 Process의 변화

fork( ) system call:
Process control Block을 포함한 parent Process 영역의 대부분이 child Process에 복사되어 똑같은 Process가 만들어짐

* PCB 내용 변경
- PID(Process 구분자)
  - parent Process 구분자
  - child Process 구분자
- 메모리 관련 정보

- fork( ) system call의 장점
  - process의 생성 속도가 빠름
  - 추가 작업 없이 자원을 상속할 수 있음
  - 시스템 관리를 효율적으로 할 수 있음 (child process의 종료 시 자원 처리)

Process의 생성과 복사

- fork( ) system call의 예

```
#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); }
}
```

child process의 PID(CPID)

process복사

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); }
}
```

0

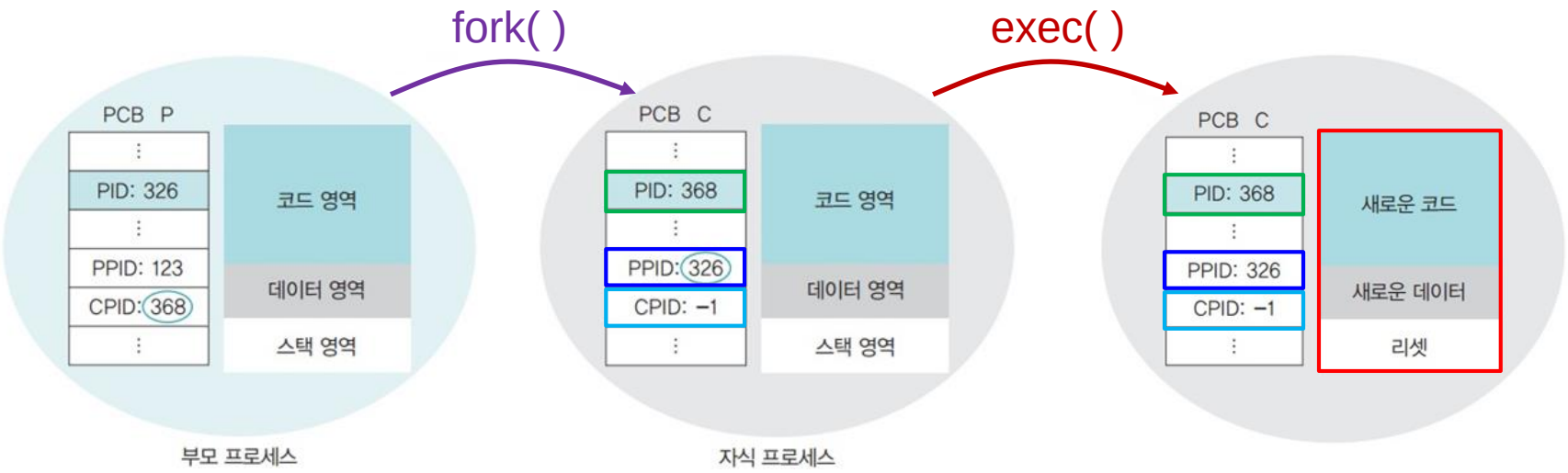
자식 프로세스

[그림] fork() system call 코드의 예

- parent process의 코드가 실행되어 fork( ) 문을 만나면 똑같은 내용의 child process를 하나 생성
- 이때 fork( ) 문은 parent process에 child process의 PID를 반환하고 child process에 0을 반환
- 만약 0보다 작은 값(-1)을 반환하면 child process가 생성되지 않은 것으로 여겨 'Error'를 출력

Process의 전환

- **exec( ) system call의 개념**
  - 기존의 Process를 **새로운 Process로 전환**(재사용)하는 함수
    - fork( ): 새로운 Process를 복사하는 system call
    - exec( ): process는 그대로 둔 채 내용만 바꾸는 system call



[그림] exec() system call 후 Process의 변화

- exec( ) system call을 하면 코드 영역에 있는 **기존의 내용을 지우고 새로운 코드로 바꿈**
- 데이터 영역이 새로운 변수로 채워지고 스택 영역이 리셋
- Process control Block의 내용 중 PID, PPID, CPID, 메모리 관련 사항 등은 변하지 않지만 Program counter 값을 비롯한 각종 register와 사용한 파일 정보가 모두 리셋

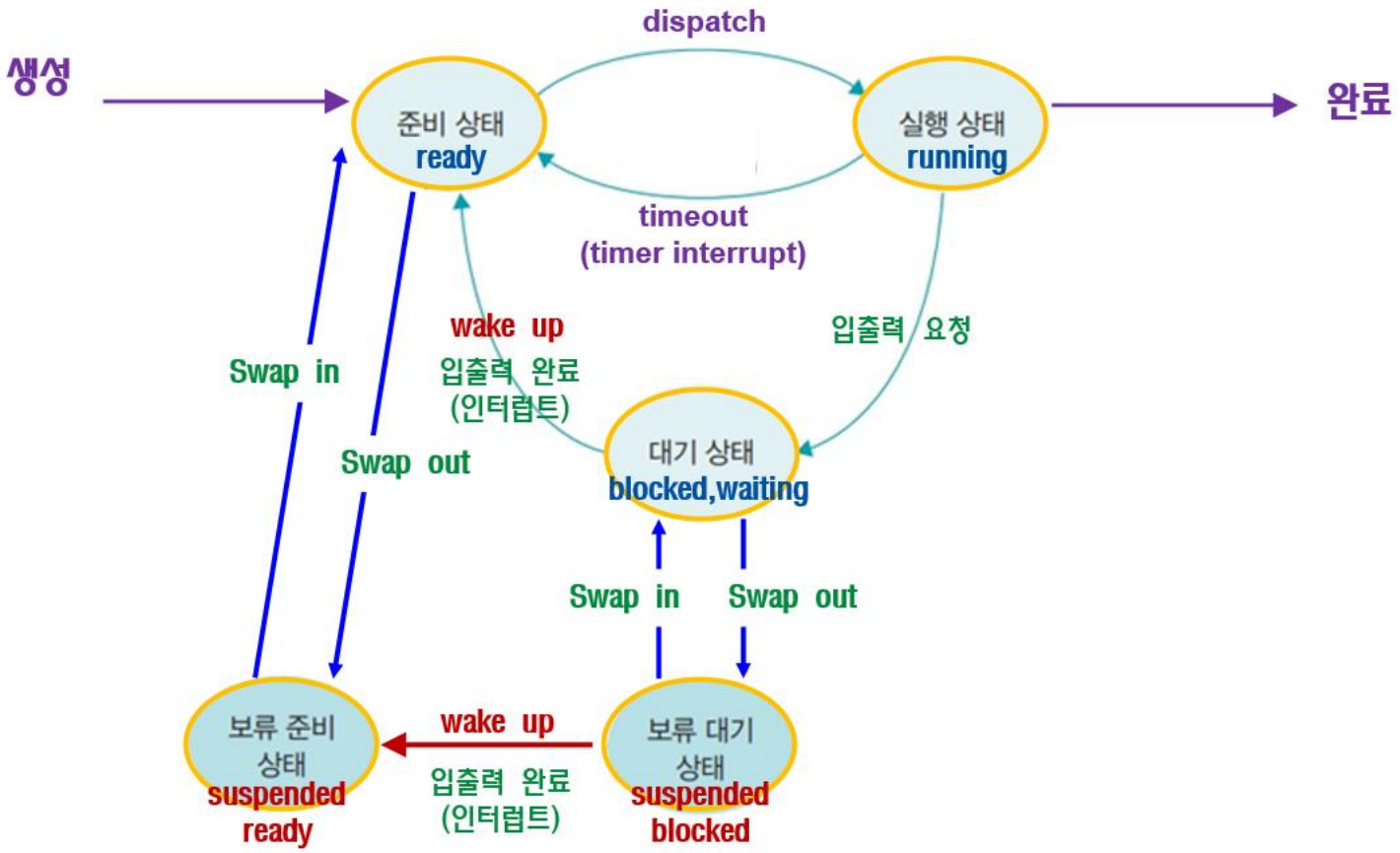
Zombie Process와 Orphan Process

- **Zombie Process** **Process가 종료된 후에도 OS의 process table에 남아있는 process**
  - 운영 체제는 모든 process를 관리하는 table을 가지고 있음
  - **process가 종료**되기 될 때
    - 종료 하려는 process가 상위(parent) process로 data collection 신호(SIGCLD, SIGCLD 등)를 전송함
    - 상위(parent) process는 child process의 data collection 작업을 한 후 child process는 종료됨
    - Data collection이 실패하면 process가 process table에 남아 있음
- **Orphan Process**
  - process가 종료될 때 상대적으로 하위(child) process들의 처리 방법
    - init process(1번 process)에 의해 하위 process들이 입양됨
    - 입양되지 못한 하위의 process가 종료되려고 하면 data collection을 해 줄 parent process가 없는 상태이므로 zombie process가 됨
  - C 언어의 main함수 하단에 exit( ) 또는 return( ) 문의 사용은 child process가 작업이 끝났음을 parent process에 알리는 것

Process 상태 5가지

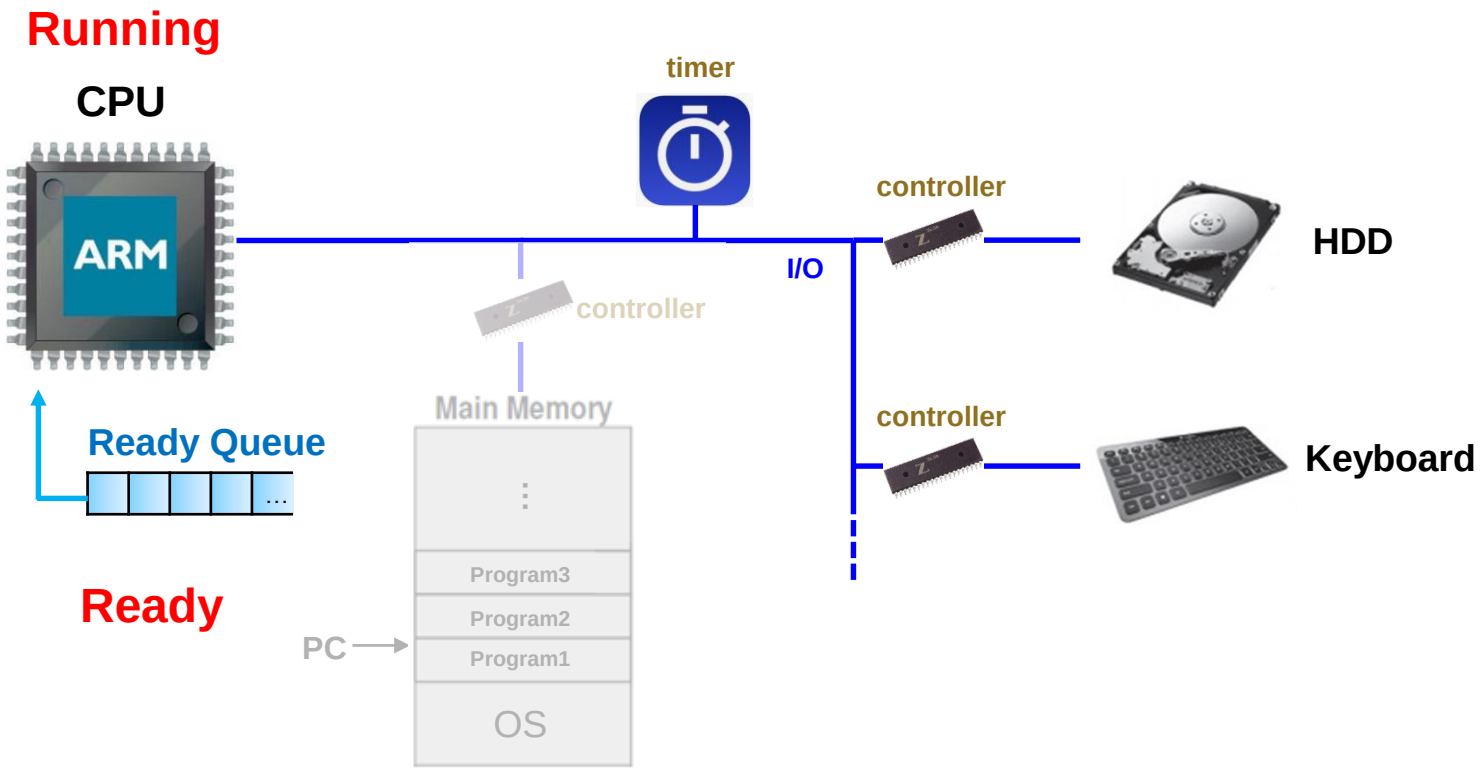
- Process 상태 5가지 :

Ready, Running, Blocked, suspended ready, suspended blocked



[그림] Process의 5가지 상태

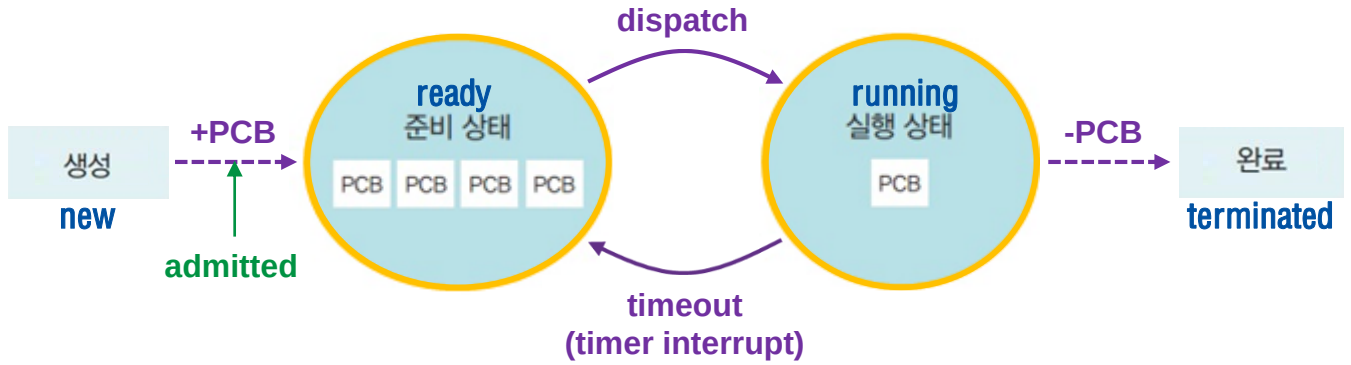
Process 상태 – Ready, Running



* 그림 상의 Queue는 실제 OS 내부(memory 내부)에 있음

Process 상태 설명 – Ready, Running

- Process의 ready, running 상태



[그림] 간단한 process의 상태

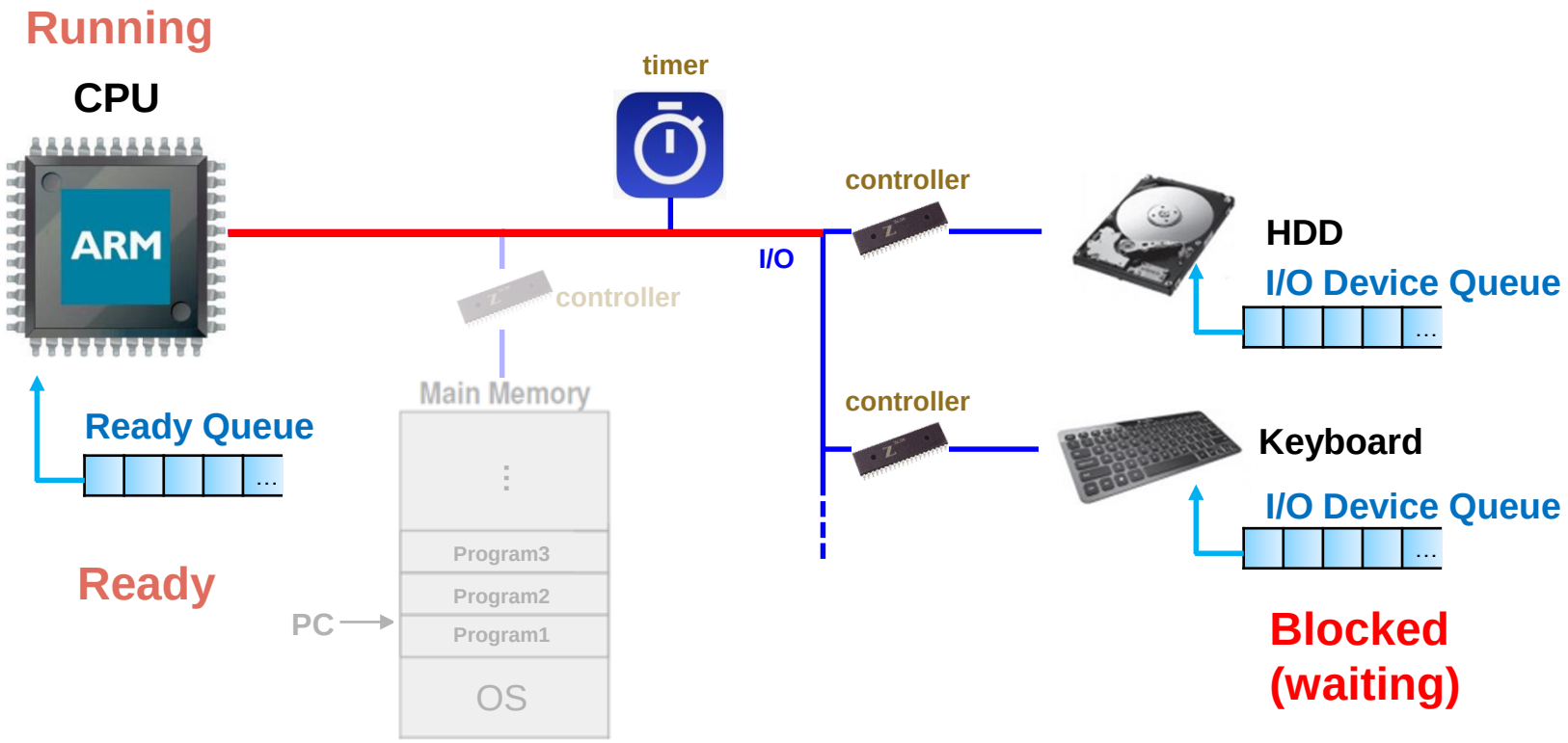
* **new(생성)**: Process가 메모리에 올라와 **실행 준비를 하는 과정** (OS에 process 등록 중)

1. **ready(준비 상태)**: 생성된 Process가 CPU를 얻을 때까지 기다리는 상태
2. **running(실행 상태)**: 준비 상태에 있는 Process 중 하나가 CPU를 얻어 실제 작업을 수행하는 상태

* **terminated(완료)**: 실행 상태의 Process가 **작업을 끝낸 것**
(Process Control Block이 사라짐- process 아님)

- **dispatch(디스패치)** : CPU scheduler가 ready상태의 process 중 하나를 선택하면 dispatcher가 해당 process에게 CPU제어권을 넘겨서 실행 상태로 바꾸는 작업
- **timeout(타임아웃)** : process가 자신에게 주어진 하나의 타임 슬라이스time slice (타임 퀀텀time quantum) 동안 작업을 끝내지 못하면 다시 준비 상태로 돌아가는 것

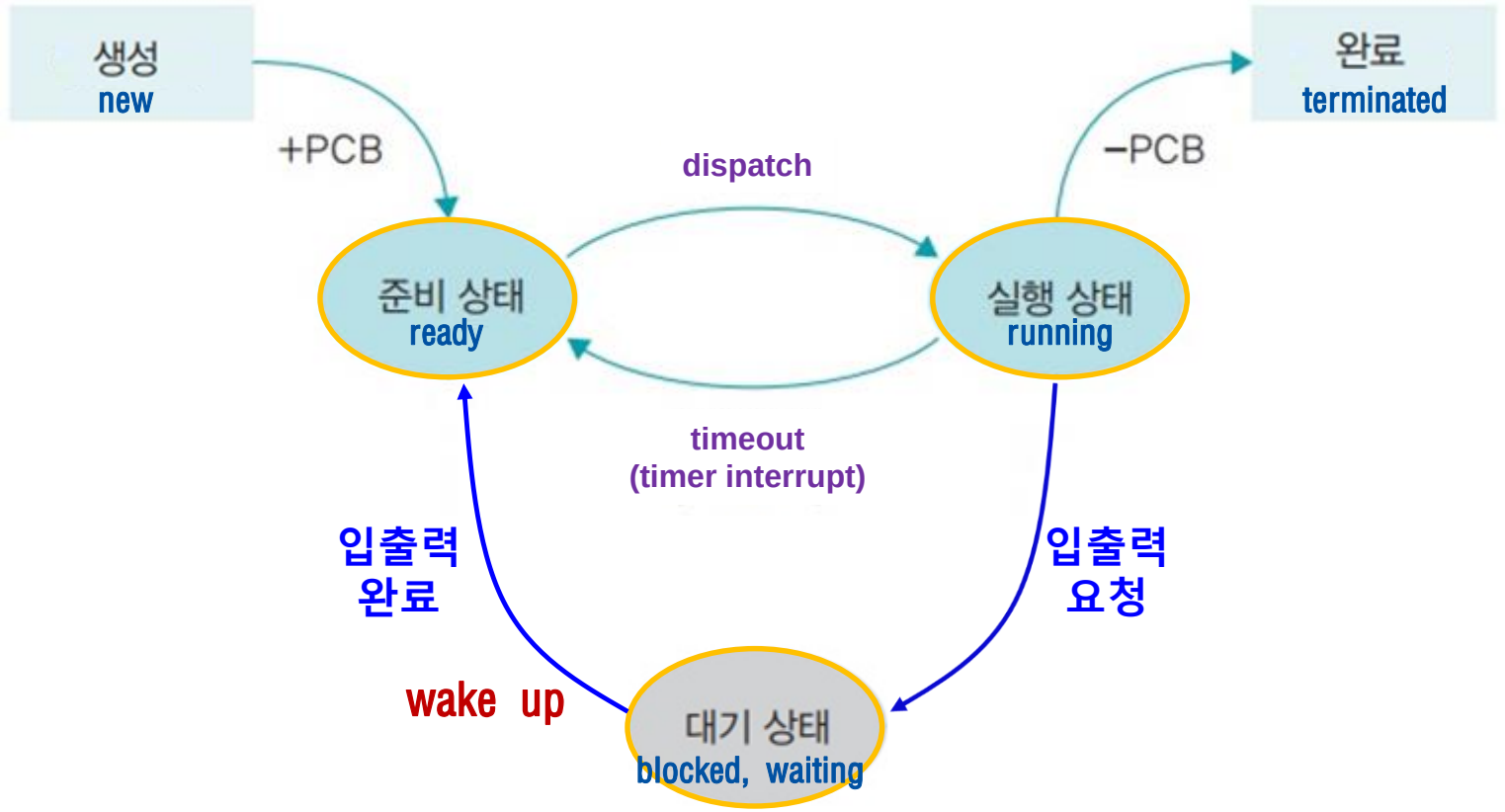
Process 상태 – Ready, Running, Blocked



* 그림 상의 Queue는 실제 OS 내부(memory 내부)에 있음

Process 상태 – Ready, Running, Blocked

- Process의 상태 – 대기 상태를 추가한 Process 상태



[그림] Process의 5가지 상태

Process 상태 설명 – Ready, Running, Blocked

* new (생성)

- Program이 메모리에 올라오고 운영체제로부터 **PCB을 할당** 받아서 실행 준비를 하는 과정
- Process가 생성이 되면 **준비 상태**에서 자기 순서를 기다리며, **PCB도 같이 준비 상태**로 옮겨짐

메모리 할당, PCB 생성

1. ready(준비) 상태

- 실행 대기 중인 모든 Process가 **자기 순서를 기다리는(실행 가능한) 상태**
- PCB는 **Ready Queue(준비 큐)**에서 기다리며 **CPU 스케줄러**에 의해 관리
- **CPU 스케줄러**는 준비 상태에서 큐를 몇 개 운영할지, 큐에 있는 어떤 Process의 PCB를 실행 상태로 보낼지 결정
- CPU 스케줄러가 **어떤 PCB를 선택하는 작업**은 **dispatch명령** 으로 처리
- CPU 스케줄러가 dispatch를 실행하면 **dispatcher가 해당 Process가 준비 상태에서 실행 상태로** 바뀌어 작업을 함

dispatch(PID)준비 → 실행

Process 상태

2. running(실행) 상태

- Process가 **CPU를 할당 받아 실행되는 상태**
- 실행 상태에 있는 Process는 자신에게 주어진 시간, 즉 **time slice (타임 슬라이스)** 동안만 작업할 수 있음
- **time slice**가 지나면 **timeout**이 실행되어 **실행 상태에서 준비 상태로** 옮김
- 실행 상태 동안 **작업이 완료되면 exit가 실행**되어 Process가 정상 종료
- 실행 상태에 있는 Process가 **입출력을 요청하면** CPU는 I/O management에게 입출력을 요청하고 **blocked를 실행**
  - 입출력이 완료될 때까지 작업을 진행할 수 없기 때문에 해당 Process를 **blocked(대기) 상태로 옮기고 CPU 스케줄러는 새로운 Process를** 선택하여 실행준비를 시킴

timeout(PID) : 실행 → 준비
exit(PID) : 실행 → 완료
blocked(PID) : 실행 → 대기

Process 상태

3. blocked(waiting,대기) 상태

- 실행 상태에 있는 Process가 **입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태**
- 대기 상태의 Process는 **입출력 장치 별로 마련된 큐에서 기다리다가 완료되면 인터럽트가 발생**하고, 대기 상태에 있는 **여러 Process 중** 해당 인터럽트로 깨어날 Process를 찾는데 이것이 wakeup
- Wakeup 으로 **해당 Process의 PCB가 준비 상태로 이동**

wakeup(PID) : 대기 → 준비

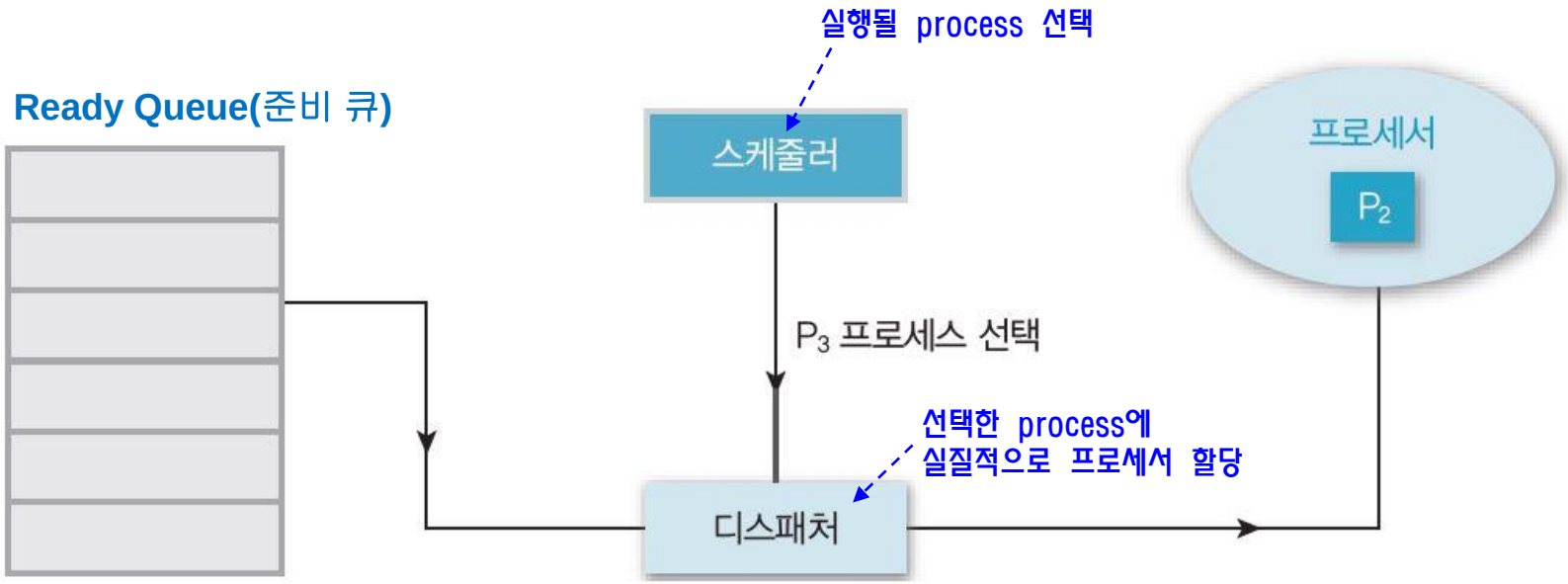
* terminated (완료)

- Process가 종료되는 상태
- **코드와 사용했던 데이터를 메모리에서 삭제**하고 **PCB를 폐기**
- 정상적인 종료는 간단히 **exit( )**로 처리
- 오류나 다른 Process에 의해 **비정상적으로 종료되는 강제 종료**를 만나면 디버깅하기 위해 종료 직전의 메모리 상태를 저장장치로 옮기는데 이를 **코어 덤프(core dump)**라고 함

메모리 삭제, PCB삭제

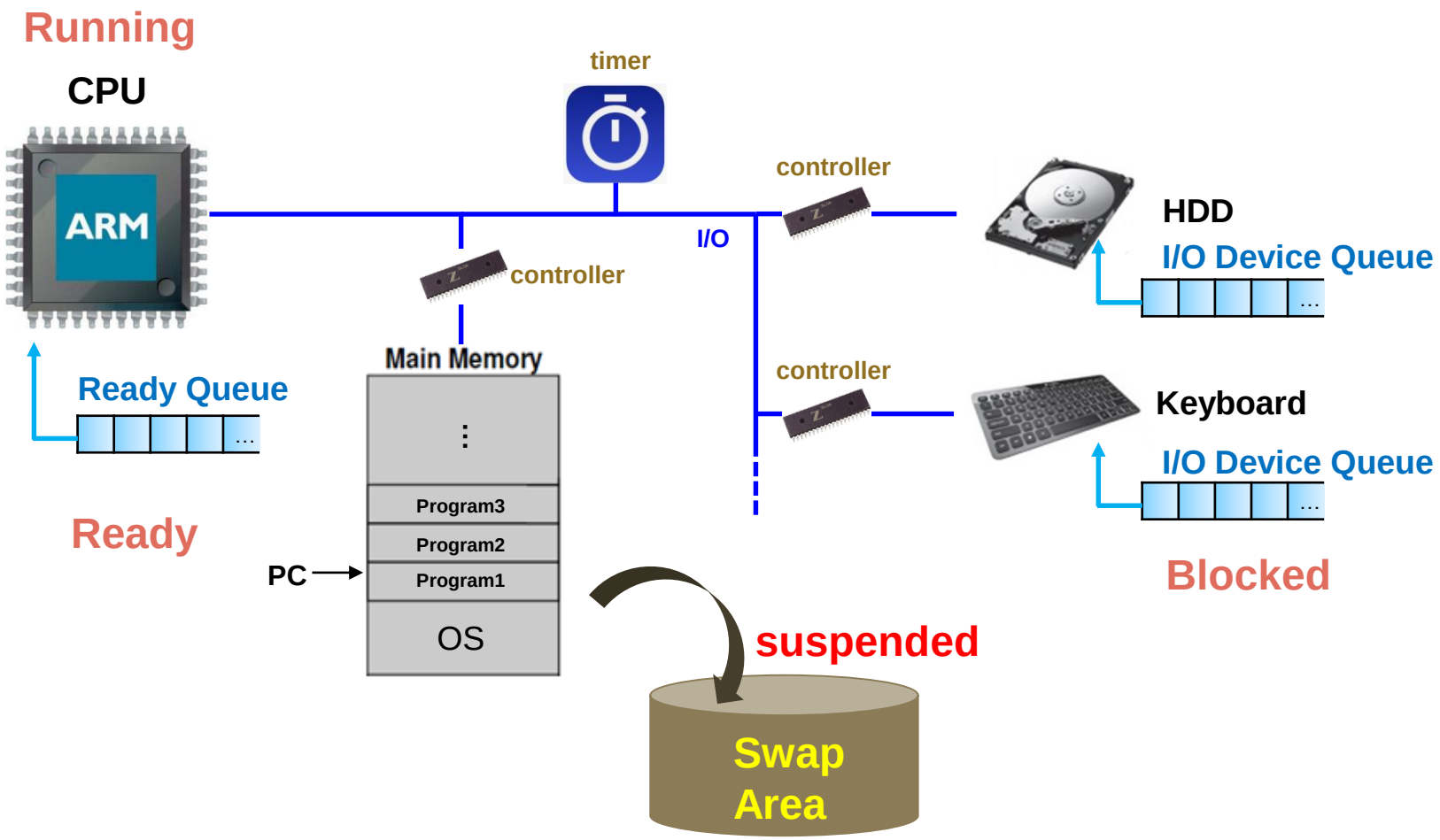
dispatcher

- blocked(waiting,대기) 상태의 Process는 **대기 원인이 제거 되면 ready(준비) 상태**가 됨
- dispatcher(디스패처)가 ready 상태의 Process에 **프로세서를 할당하면 running(실행) 상태**가 됨



[그림] dispatcher가 p3 process에게 프로세서를 할당하는 예

Process 상태 – Ready, Running, Blocked, suspended

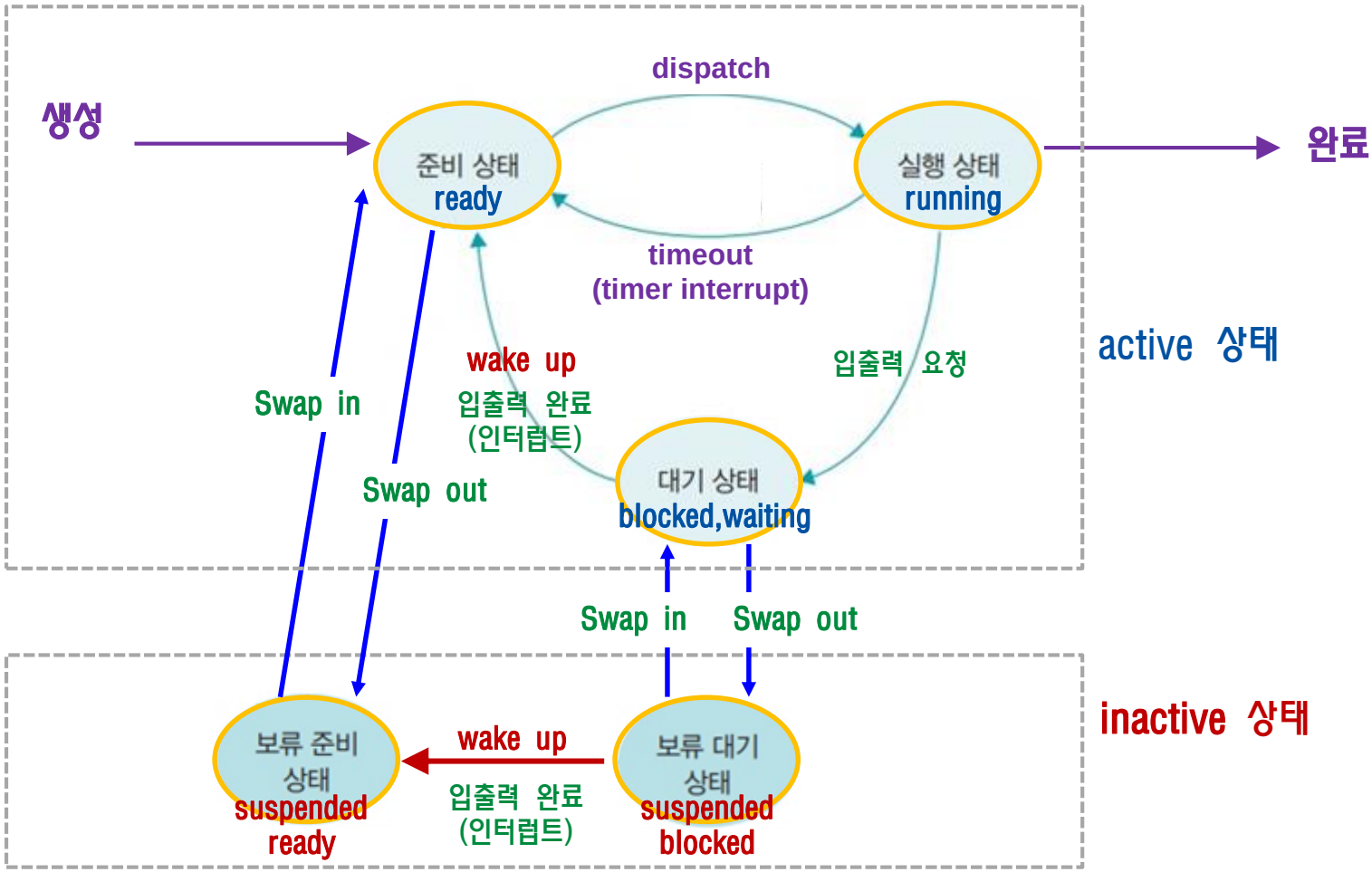


Process 상태 설명 – suspended

- Suspended(보류) 상태
  - process가 **메모리에서 잠시 쫓겨난 상태**
  - process는 다음과 같은 경우에 suspended 상태가 됨
    - **메모리가 꽉 차서 일부 process를 메모리 밖으로 내보낼 때**
    - **입출력을 기다리는 process의 입출력이 계속 지연될 때**
    - **매우 긴 주기로 반복되는 process라 메모리 밖으로 쫓아내도 문제가 없을 때**
    - program에 오류가 있어서 실행을 미루어야 할 때
    - 바이러스와 같이 악의적인 공격을 하는 process라고 판단될 때

Process 상태 – Ready, Running, Blocked, suspended

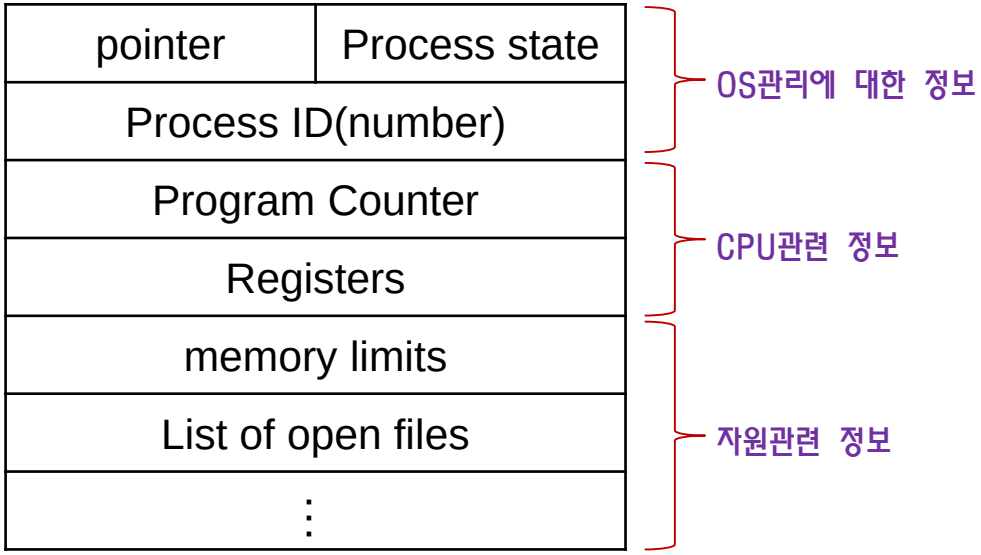
- suspended 상태를 포함한 Process의 5가지 상태



[그림] Process의 상태 5가지

Process Control Block (PCB, Process 제어 블록)

- Process Control Block(PCB)
  - Process를 실행하는 데 필요한 중요한 정보를 보관하는 것
  - Process는 고유의 PCB를 가짐
  - Process 생성 시 만들어져서 Process가 실행을 완료하면 폐기
- Process Control Block의 구성



Process control Block

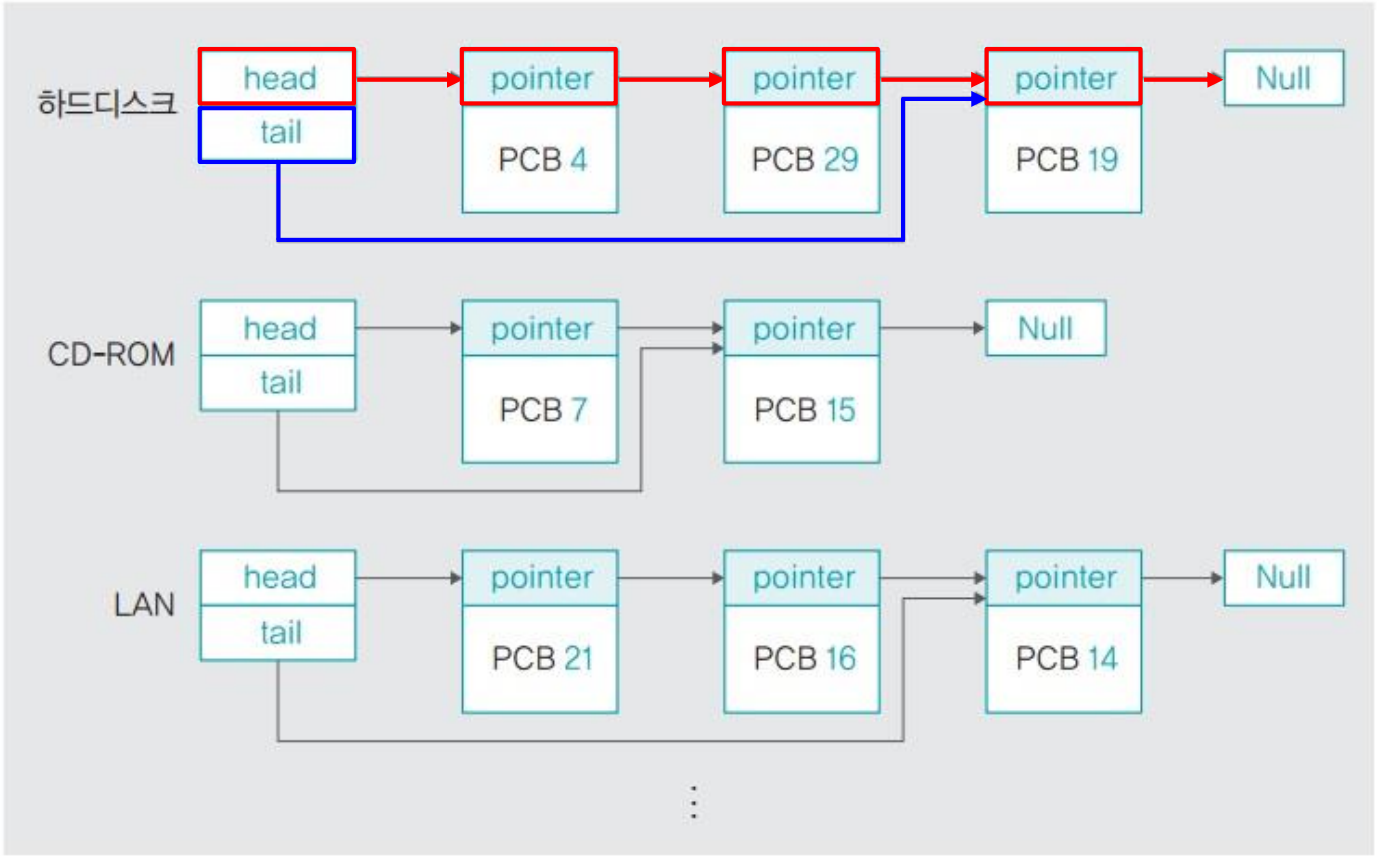
pointer	Process state
Process ID(number)	
Program Counter	
Registers	
memory limits	
List of open files	
⋮	

- Process control Block의 구성
  - pointer : **Queue를 구현**할 때 사용
  - **Process state** : Process가 **현재 어떤 상태에 있는지**를 나타내는 정보(준비, 실행 …)
  - **Process ID** (PID) : 운영체제 내에 있는 **여러 Process를 서로 구분**하기 위한 구분자
  - **Program Counter** : **다음에 실행될 명령어의 위치**를 가리키는 값
  - **Registers** : Process가 실행되는 중에 **사용하던 레지스터의 값** (정보)
  - **Memory limits** : Process가 메모리의 어디에 있는지 나타내는 메모리 위치 정보, 메모리 보호를 위해 사용하는 **경계 레지스터 값과 한계 레지스터 값** 등
  - **List of open files** : Process를 실행하기 위해 사용하는 **입출력 자원이나 오픈 파일 등에 대한 정보**
  - **기타 정보** : 계정 번호, CPU 사용 시간 등

Process control Block

- 포인터(pointer)
  - blocked상태에는 같은 입출력을 요구한 process들을 서로 연결을 위해서 pointer 사용

blocked 상태 (waiting, 대기)



[그림] blocked 상태의 Queue

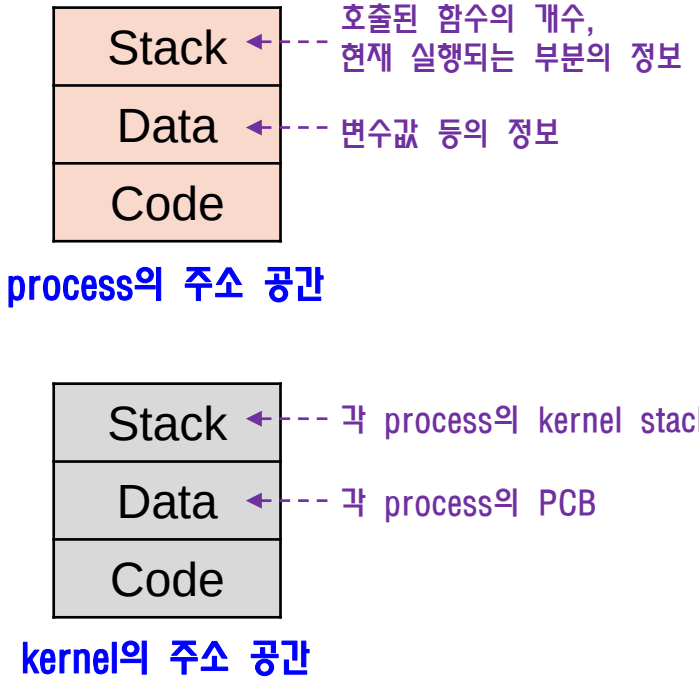
Process의 문맥 (Context) Process의 흐름, 현재 상태에 대한 정보

- Process의 문맥 (Context)
  - process: A program in execution

1. 하드웨어 문맥:
   Program Counter, 여러 register값

2. process의 memory공간:
   code, data, stack

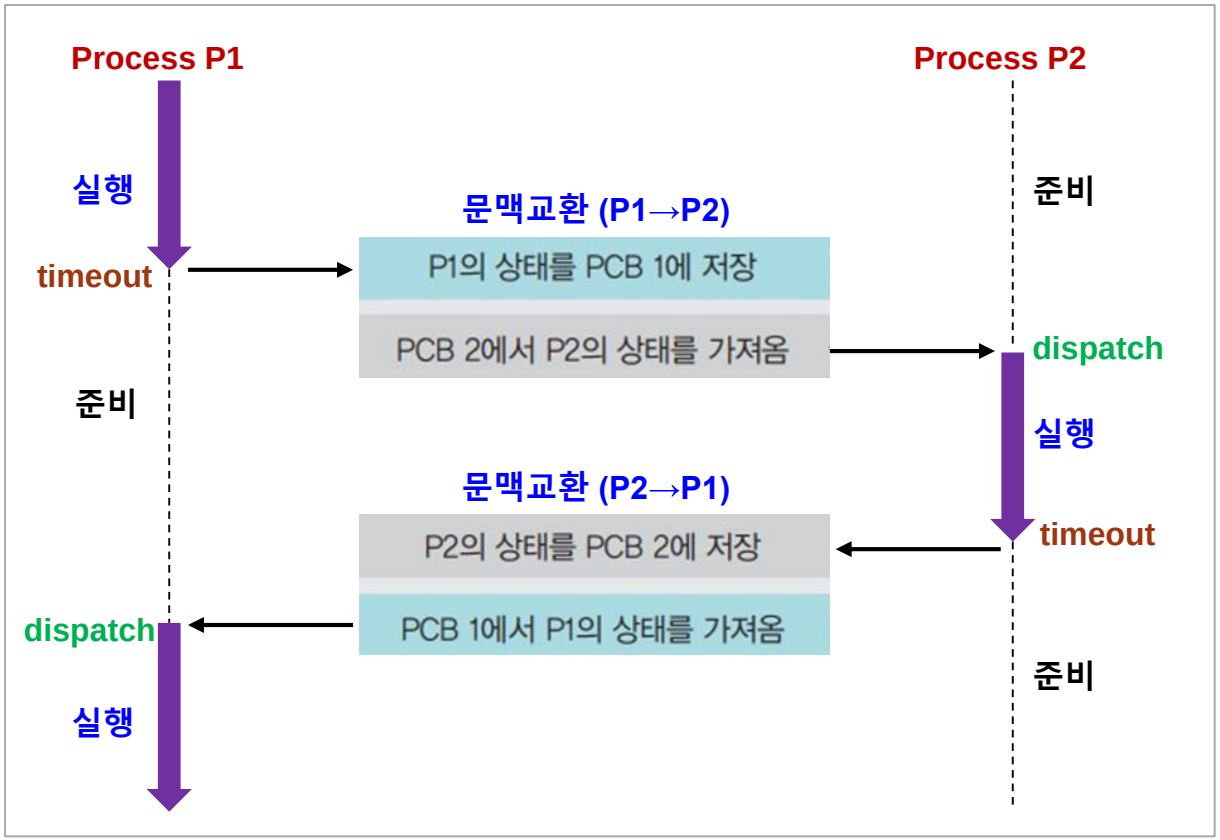
3. process의 kernel관련 자료:
   PCB, Kernel stack



Context switching

- **Context switching(문맥 전환)**
  - CPU를 차지하던 Process가 나가고 새로운 Process를 받아들이는 작업
  - 실행 상태에서 나가는 PCB에는 지금까지의 작업 내용을 저장하고, 반대로 실행 상태로 들어오는 PCB의 내용으로 CPU가 다시 세팅

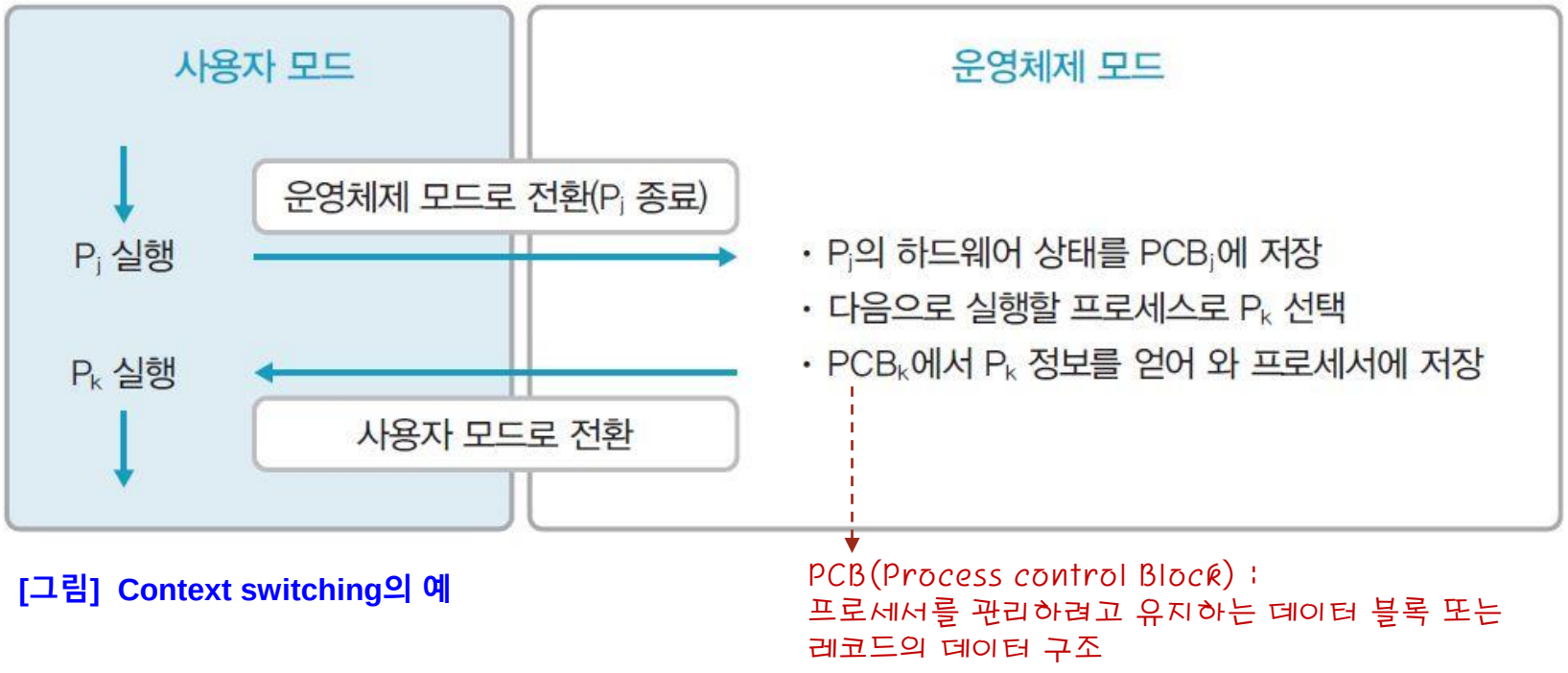
* context switching 절차



[그림] Context switching 과정

Context switching

- Context switching(Context switching)
  - 오버헤드 발생
  - 오버헤드는 시간 비용 소요되어 **운영체제 설계 시 불필요한 Context switching 감소가 주요 목표**



[그림] Context switching의 예

Context switching의 발생

- **Process의 Context switching 발생**
  - 실행 중인 Process에 **interrupt가 발생**하면 운영체제가 다른 Process를 실행 상태로 바꾸고 제어를 넘겨주어 Process Context switching 발생
  - 인터럽트 유형에 따른 루틴 분기
    - **I/O interrupt** : 입출력 동작이 발생 확인, 다른 Process를 실행 상태로 전환
    - **timer interrupt** : 실행 중인 Process 할당 시간 조사, 준비 상태로 바꾸고, 다른 Process를 실행 상태로 전환
  - 인터럽트가 항상 context switching으로 되지는 않음
    → ISR을 실행한 후 **현재 실행 중인 Process를 재실행** 하는 경우도 있음
  - Context switching 에서는 **오버헤드**가 발생

감사합니다.