

운영체제

Lecture 03: 프로세스



전남대학교 인공지능학부
박태준 (taejune.park@jnu.ac.kr)

지난 시간 복습

▶ 컴퓨터 아키텍쳐 ~ 폰노이만 구조

- 응용프로그램 ↔ 운영체제 ↔ 하드웨어!
- 폰노이만 구조 ‘메모리에 프로그램 적재 후 실행’ → 버스 → 병목현상 → 메모리 계층 구조 → 캐시!
 - CPU: Fetch → Decode → Execute!

▶ 다중 프로그래밍~대화형 구조 - 중간에 끼어든 작업에 대한 처리 방법

- 폴링: CPU가 입력을 가지러 감
- 인터럽트: CPU에 입력을 넣어줌
- DMA: 입력전담 직원

▶ 병렬처리: 파이프라ining

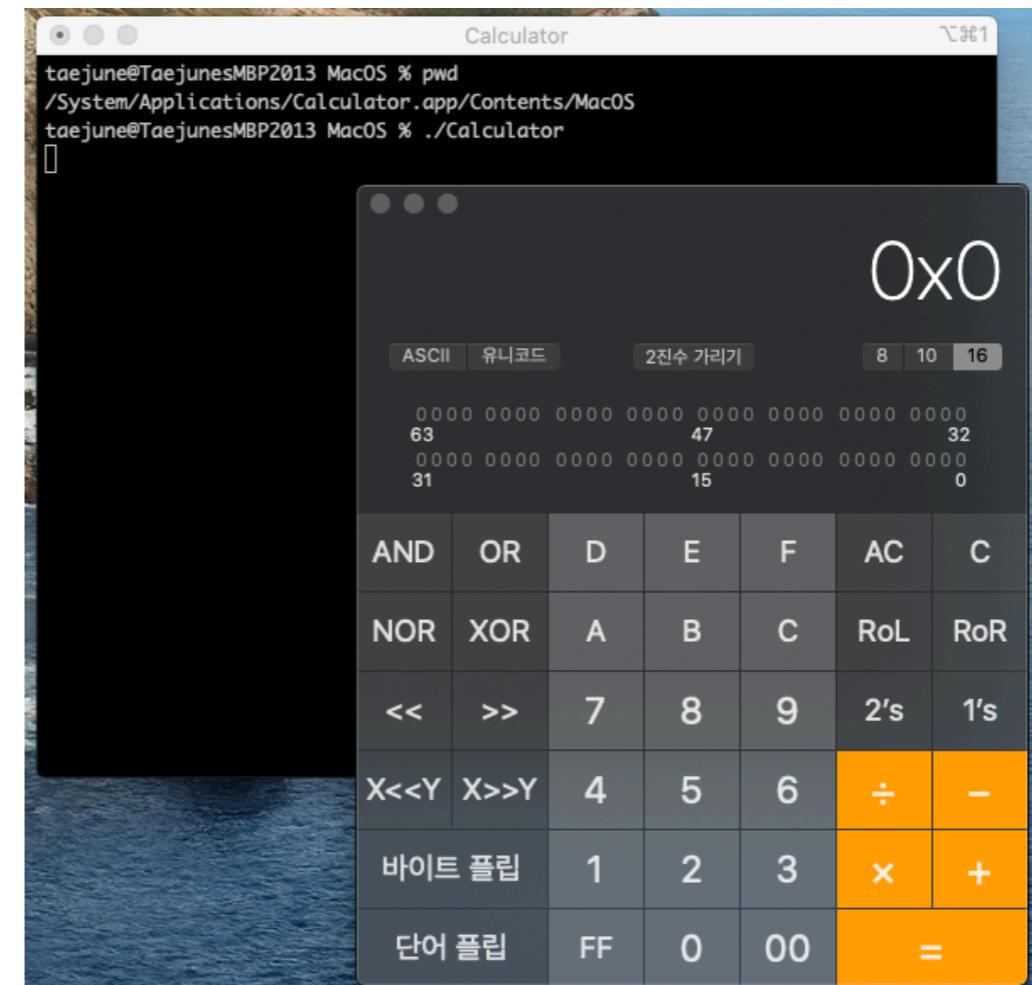
Goal

- ▶ 프로세스의 개념
 - 프로그램이 메모리에 올라가면 프로세스! → 어떻게 메모리에 올라가는 걸까?
 - 실행파일에서 프로세스까지...생명주기와 PCB: OS가 프로세스를 관리하는 방법
 - Context switching
- ▶ 프로세스의 구조
 - 메모리에 그냥 올라가는게 아니에요...!
 - 가상 메모리와 프로세스 메모리 구조
- ▶ 프로세스의 생성과 계층 구조
 - fork 및 exec, 부모 자식 관계

프로세스 Process

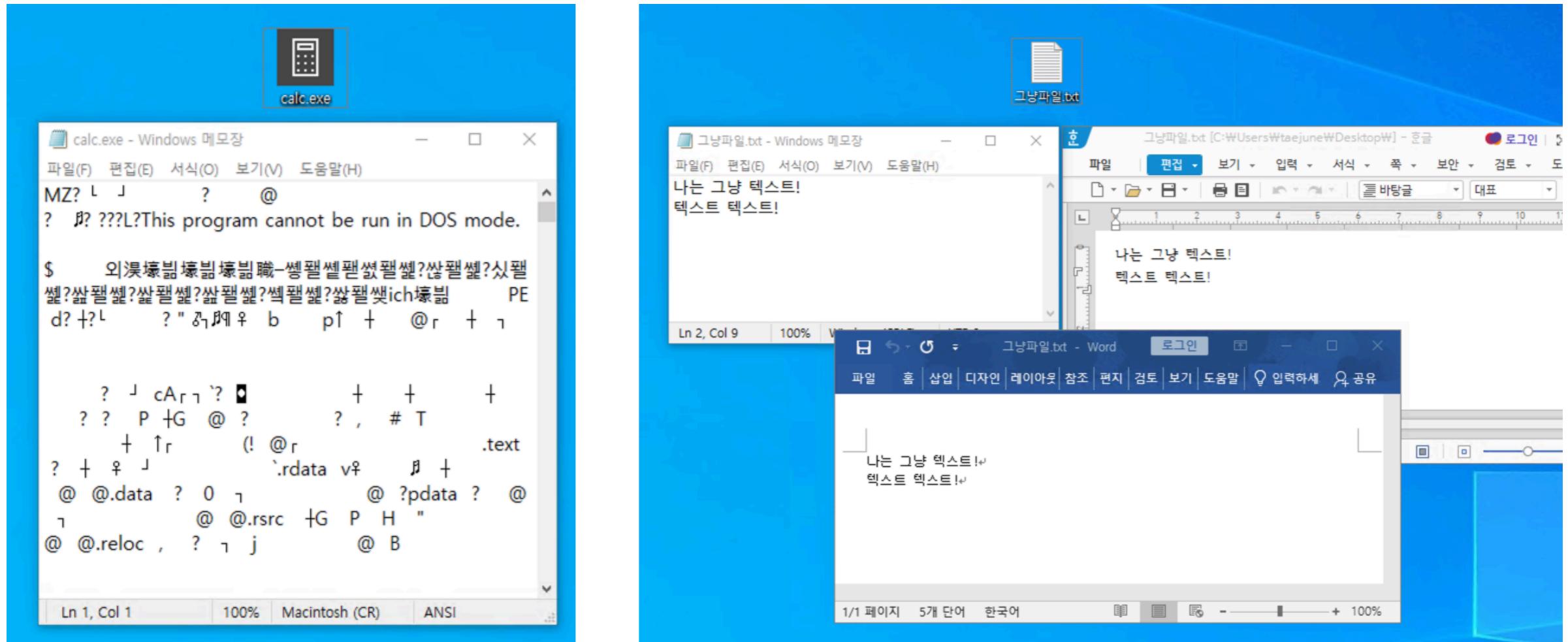
여러분은 프로그램을 어떻게 실행시키나요?

- ▶ 더블클릭을 하거나, 콘솔에서 명령어를 통해 실행하거나...
 - 어쨌든 어떤 파일을 ‘실행’시키는 과정을 거침



그냥 파일들과 무슨 차이가 있는걸까?

- ▶ 텍스트 파일들은 왜 ‘실행’이 안될까?
→ Executable vs Non-executable



실행파일과 프로세스

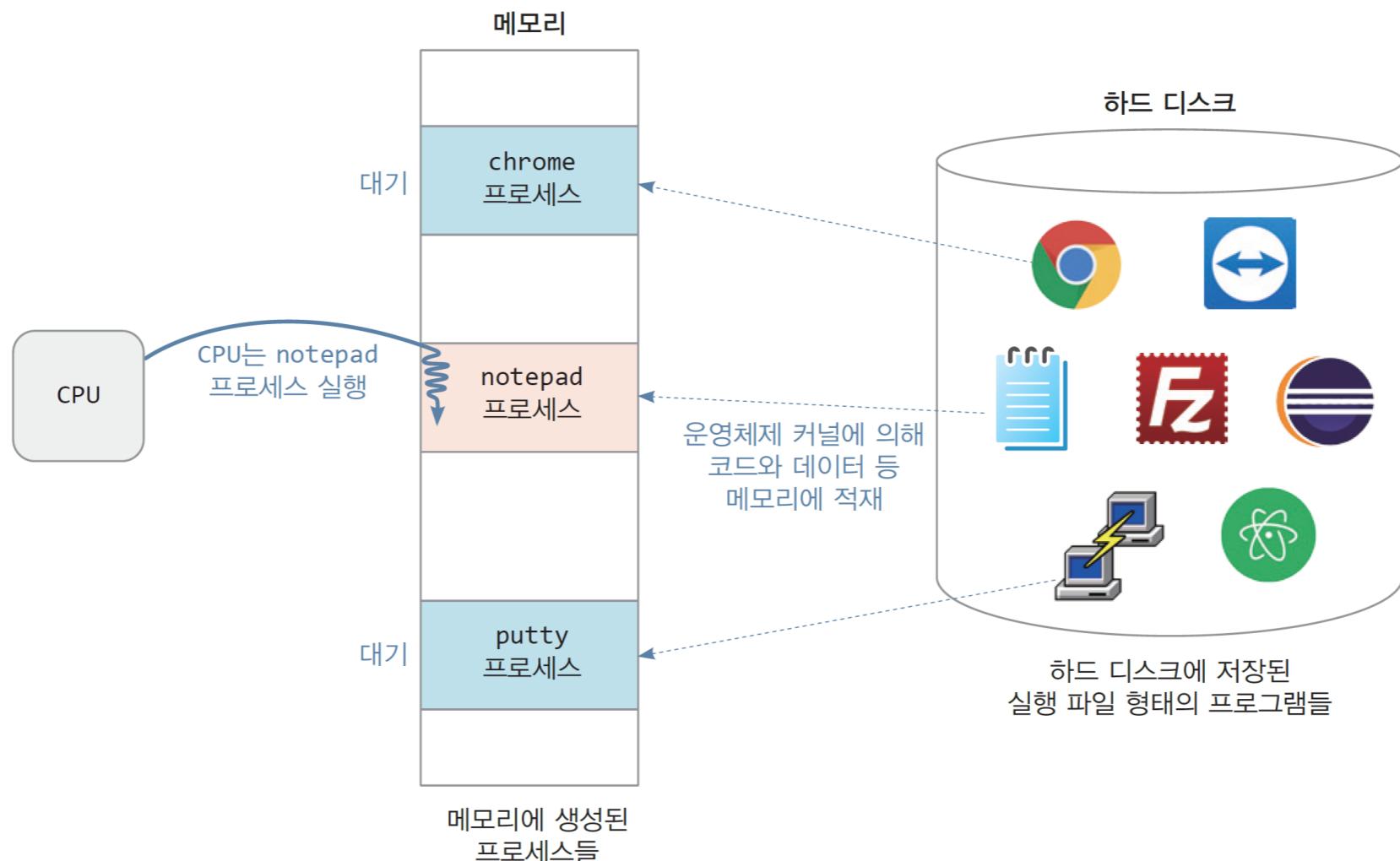
- ▶ 운영체제가 프로세스를 초기구동하는데 사용되는 파일 (**Program**)
 - 저장장치에 저장되어 있는 정적인 상태
 - Windows: .exe 파일 // Linux: ??? // macOS: ???
 - c.f., 파일 확장자의 개념
 - 운영체제는 Executable 한 파일을 읽어 들여서 프로그램을 **실행**
 - 파일 헤더 ‘매직 넘버’: MZ (ascii), ELF (ascii), 0xfeedface, 0xfeedfacf, 0xcafebabe
- ▶ 프로그램이 실행이되면? → **프로세스 Process**
 - 실행파일이 메모리에 로딩되어 실행되는 상태
 - 프로그램들은 메모리에 올라가야 실행될 수 있음! (프로그램 → 프로세스)

프로세스의 개념

- ▶ 주기억 장치에 상주된 프로그램이 CPU에 의해서 처리되는 상태
- ▶ CPU에 의해서 현재 실행되고 있는 프로그램
- ▶ 실행을 위해 메모리에 올라온 동적인 상태
- ▶ **PCB(Process Control Block, 프로세스 제어 블록)의 존재로서 명시되는 것**
- ▶ 프로세서가 할당되는 개체로서 디스패치(Dispatch)가 가능한 단위
 - 비동기적 행위를 일으키는 주체
 - CPU가 할당되는 실체
 - 운영체제가 관리하는 최소 단위의 작업(프로그램)
 - task란 용어와 함께 사용되며, 다양한 정의를 가짐
 - 프로그램과 달리 메모리에 주소 공간을 갖는 능동적인 객체

여러 프로그램을 동시에 실행하자! → 다중 프로그래밍

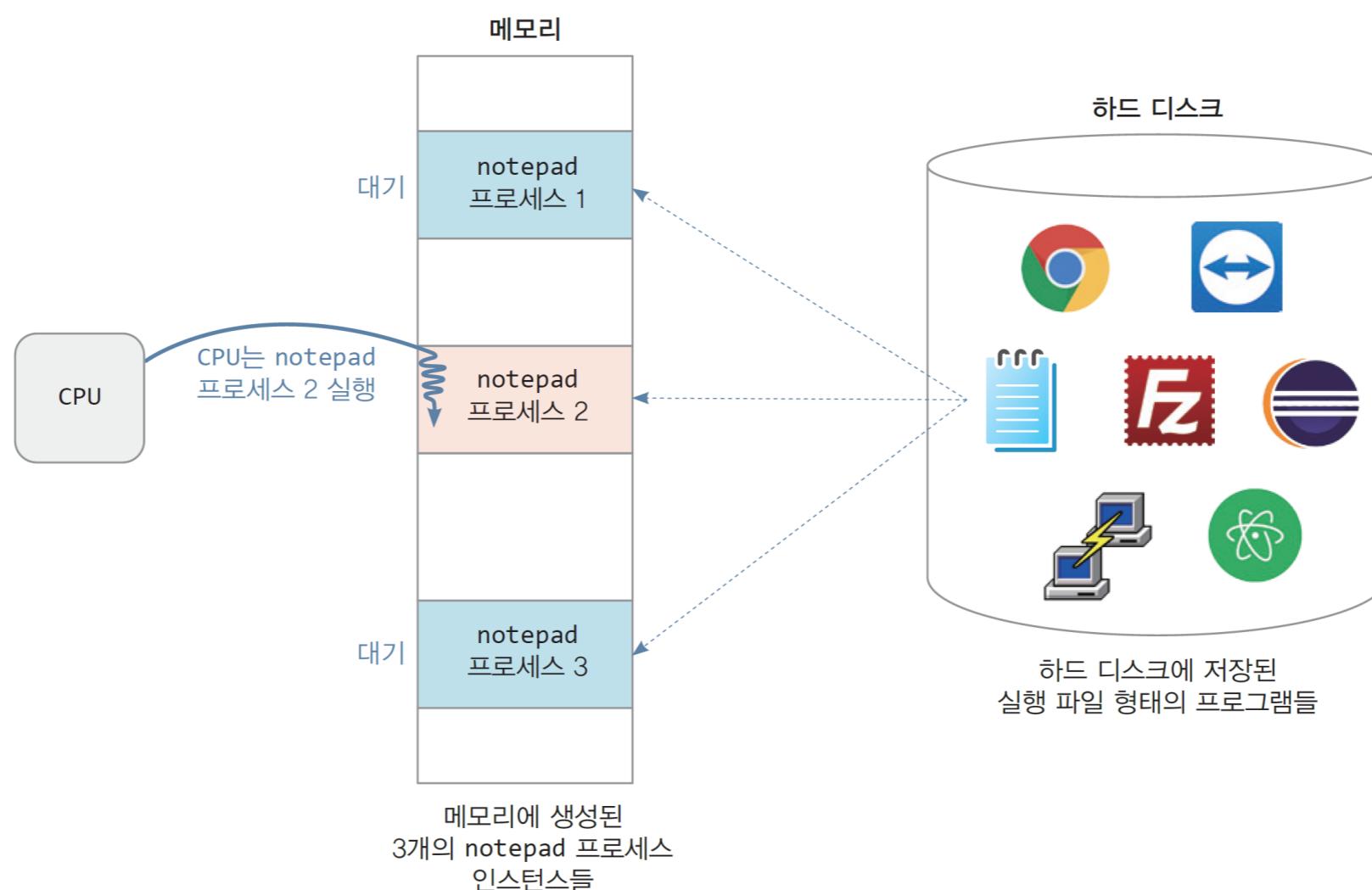
- ▶ 여러 프로세스들이 메모리에 동시에 있을 수 있음
 - 프로세스들은 상호 독립적인 메모리 공간에서 실행



그리고 하나의 프로그램은 여러 프로세스가 될 수 있음!

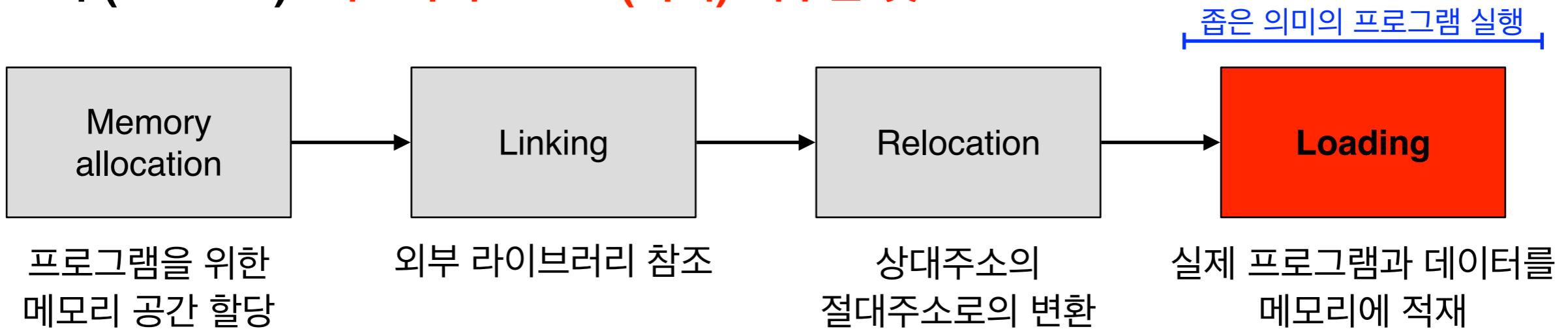
▶ 다중 인스턴스 (Multi-instance)

- 같은 프로그램이여도, 실행될 때마다 독립된 프로세스 생성
 - 각 프로세스는 독립된 메모리 공간을 가지고, 별개의 프로세스들로 취급됨.



실행파일이 메모리에 올라가는 과정; Loading

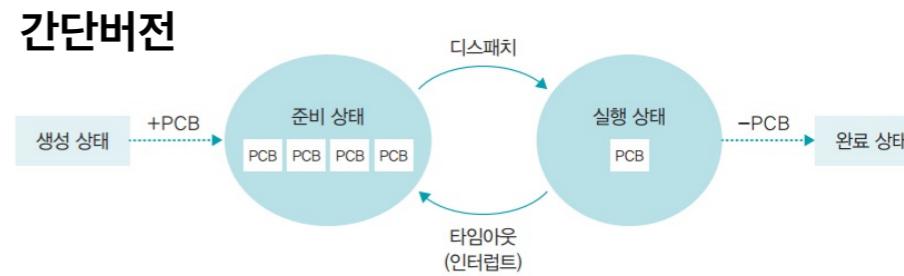
- ▶ 로더 (Loader): 메모리에 Load (적재) 해주는 것



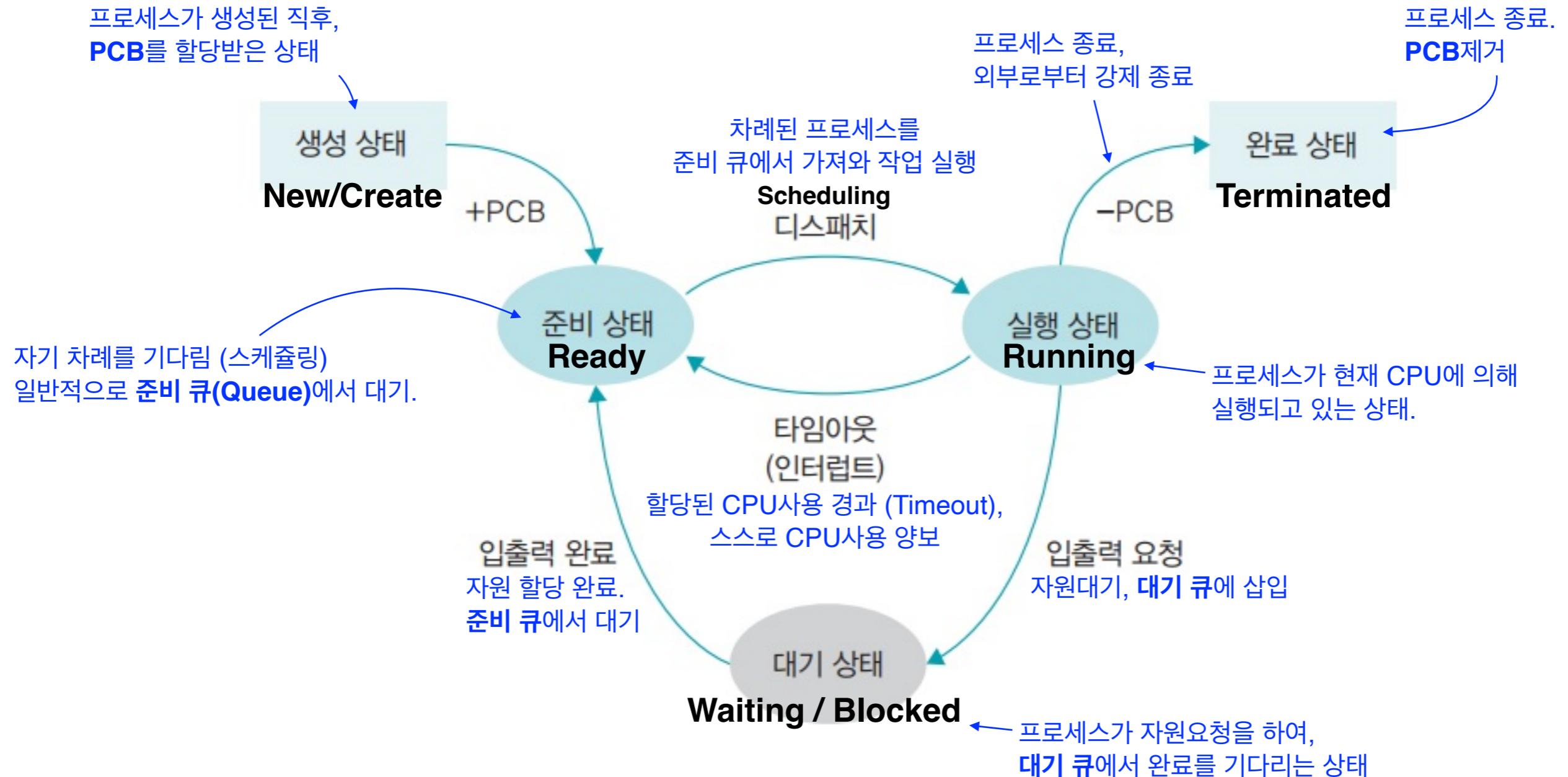
- ▶ 절대 로더 (Absolute loader)
 - 항상 고정된 위치에만 로딩됨; 재배치 및 링킹이 없음. 다중 프로그래밍 X
- ▶ 재배치 로더 (Relocation loader)
 - 프로그램이 여러개 실행되다보면, 메모리 위치상에 충돌이 있을 수도 있어요!
 - 주기억 장치의 상태에 따라 목적 프로그램을 주기억 장치 임의공간에 적재
- ▶ 동적 적재 (Dynamic loading)
 - 필요한 부분만 주기억장치에 적재하고 나머지는 보조기억장치에 저장

올라갔으면? 내려와야지!

프로세스의 생명주기; 프로세스의 生과 死



- ▶ 프로세스는 생성에서 종료까지 여러 상태로 바뀌면서 실행됨 (State change)



정리

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

(+@) 보류상태

- ▶ 프로세스가 ‘어떠한 이유로 인해’ 실행이 미뤄지고, 메모리에서 쫓겨난 상태
 - 메모리에서 쫓겨남 == 저장장치에 놓여짐
- ▶ 언제?
 - 메모리가 꽉차서...
 - 프로그램에 오류가 있어서...
 - Malware라서 격리
 - 매우 긴 주기로 실행되는 녀석이라서...
 - 입출력이 지연이 될 때
- e.g., 업데이트 소프트웨어

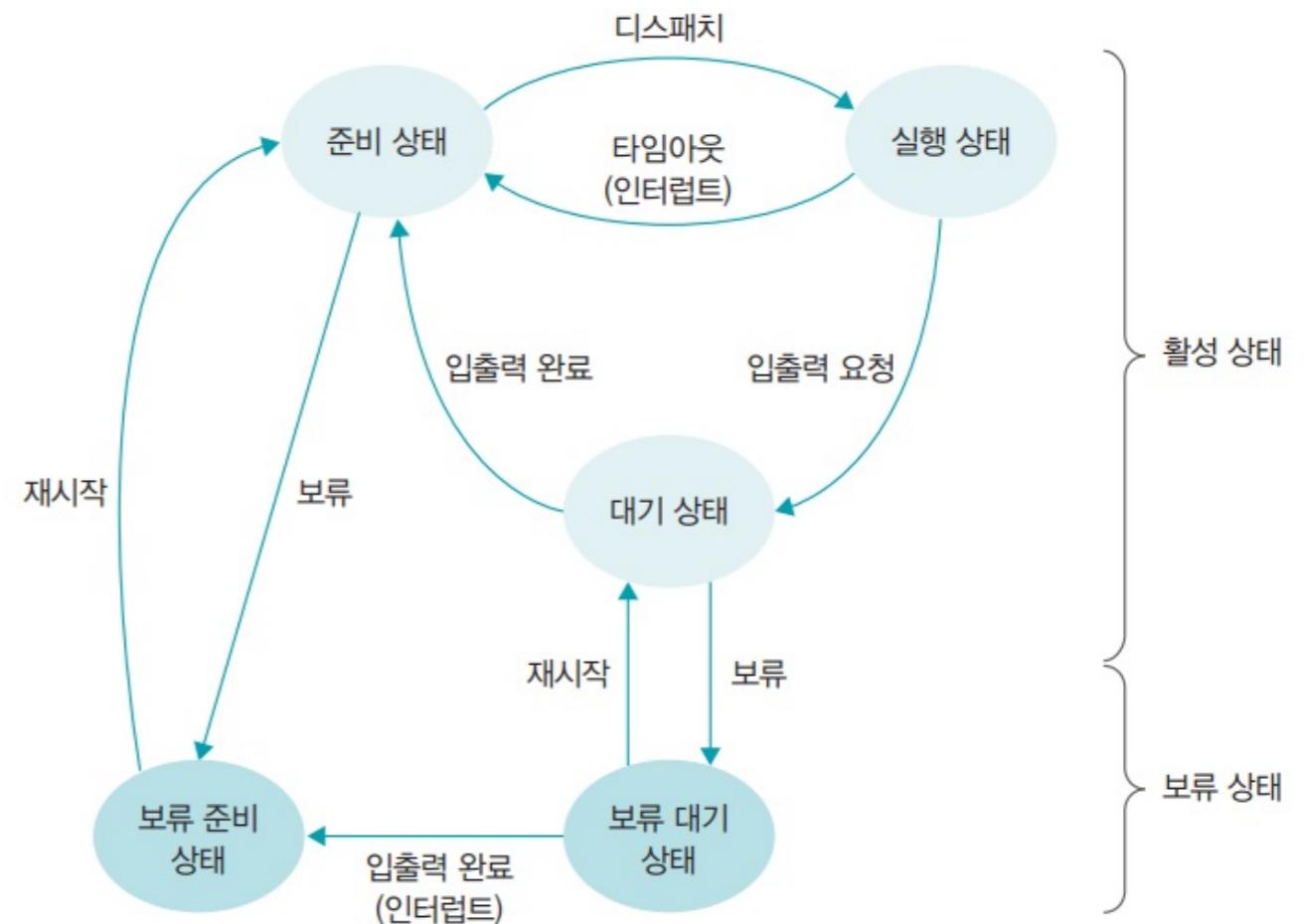
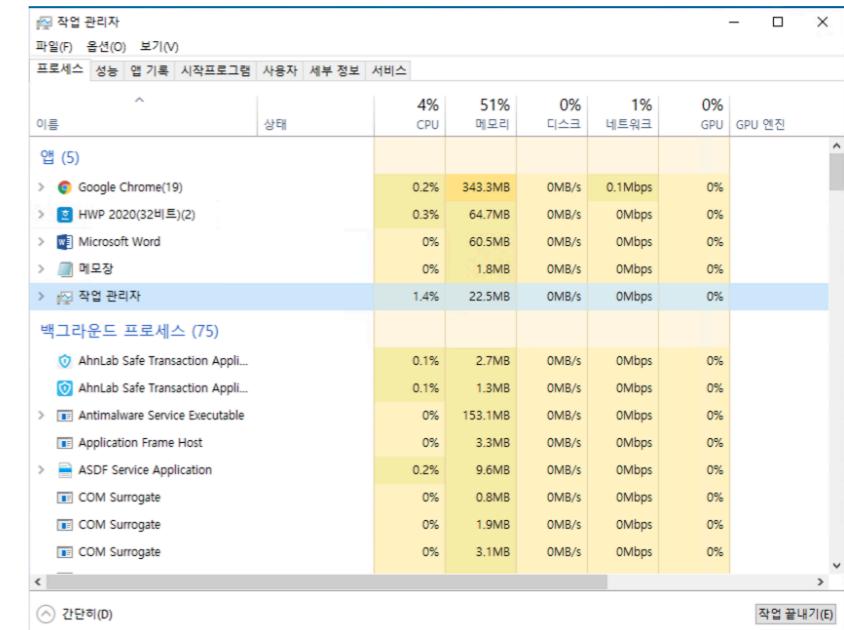


그림 3-11 보류 상태를 포함한 프로세스의 상태

어떻게 저런 일을 할까? 프로세스 관리

▶ 프로세스는 생성에서부터 종료까지, 모두 ‘커널(운영체제)’에 의해 관리됨

- 프로세스 생명주기: 생성, 실행, 중단, 일시 중단 및 재개
- 프로세스 정보 (Metadata) 관리
- 프로세스 통신, 동기화
- 컨테스트 스위칭 (Context switching, 문맥교환)



▶ Process Control Block, PCB

- 운영체제가 프로세스를 제어하기 위해 프로세스의 상태 정보를 저장하는 자료구조
- 프로세스마다 고유의 PCB가 생성됨
 - 프로세스가 종료되면 폐기됨!

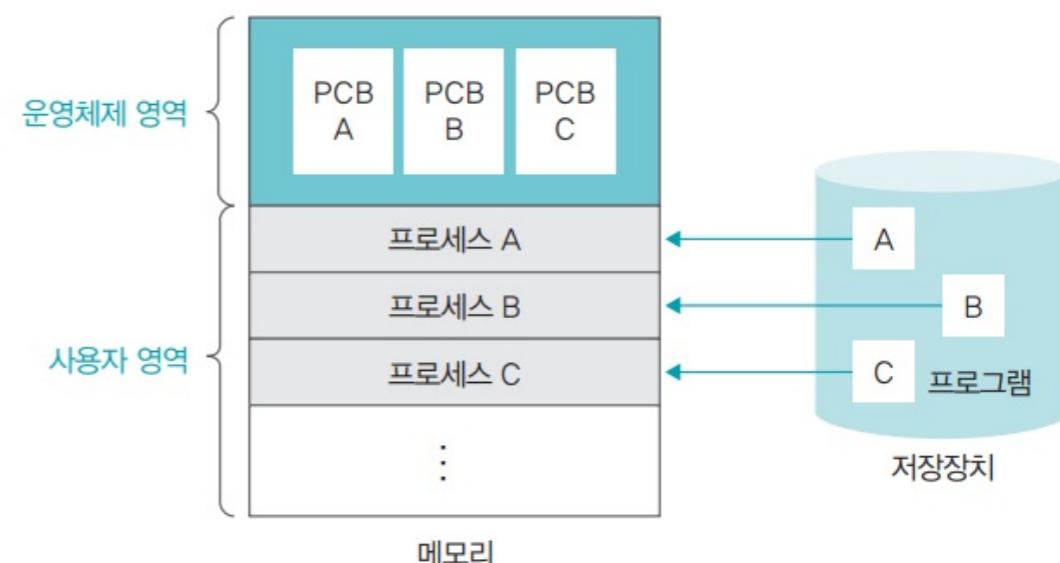
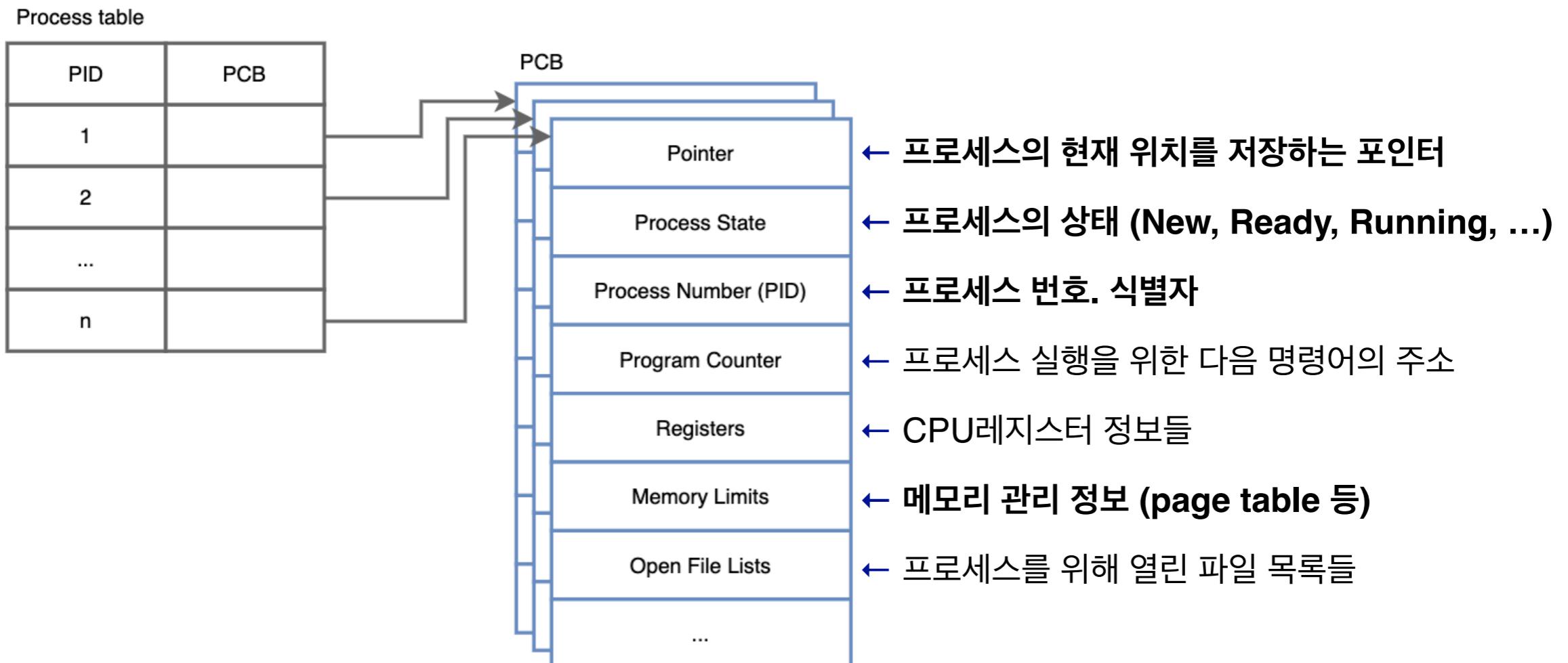


그림 3-7 프로그램이 메모리에 올라와 프로세스가 되는 과정

PCB의 구조

- ▶ 운영체제는 PCB들은 PCB table 또는 Process table이라는 곳을 통해 관리
- ▶ 주의! OS마다 다름!! 아래는 일반적인 예제!



이러한 정보들을 프로세스의 Context라고도 합니다!

프로세스 정보 보기

```
taejune@TaejunesMBP2013 ~ % ps -eal
UID  PID  PPID      F CPU PRI NI      SZ   RSS WCHAN    S          ADDR TTY      TIME CMD
 0    1    0    4004  0  37  0  4315096 22576 -  Ss          0 ??  2:17.52 /sbin/launchd
 0    87   1    4004  0   4  0  4306052 1088 -  Ss          0 ??  0:04.57 /usr/sbin/syslogd
 0    88   1    4004  0  31  0  4336300 11372 -  Ss          0 ??  0:11.45 /usr/libexec/UserEvent
 0    90   1    4004  0  20  0  4297260 1804 -  Ss          0 ??  0:02.44 /Library/PrivilegedHel
 0    92   1    4004  0  20  0  4296380 2124 -  Ss          0 ??  0:01.04 /System/Library/Privat
 0    93   1    4004  0  46  0  4856536 16820 -  Ss          0 ??  0:02.89 /usr/libexec/kextd
 0    94   1  1004004  0  50  0  4477656 7448 -  Ss          0 ??  1:23.23 /System/Library/Framew
 0    95   1    4004  0   4  0  4342648 15348 -  Ss          0 ??  0:00.61 /System/Library/Privat
 0    98   1    4004  0   4  0  4344592 15076 -  Ss          0 ??  0:27.34 /usr/sbin/systemstats
 0    99   1    400c  0  31  0  4341032 9528 -  Ss          0 ??  0:03.30 /usr/libexec/configd
 0   100   1    4004  0   4  0  4305892 1076 -  Ss          0 ??  0:00.03 endpointsecurityd
```

```
taejune@TaejunesMBP2013 ~ % ps -aj // linux: ps -aux
USER      PID  PPID  PGID  SESS JOBC STAT   TT      TIME COMMAND
root     39991 39990 39991      0   0 Ss  s000  0:00.15 login -fp taejune
taejune 39993 39991 39993      0   1 S  s000  0:00.08 -zsh
root     40079 39993 40079      0   1 R+  s000  0:00.00 ps -aj
taejune@TaejunesMBP2013 ~ %
```

Linux PCB

▶ task_struct

```
/ include / linux / sched.h

720  /*
721  *
722  * struct task_struct {
723  * #ifdef CONFIG_THREAD_INFO_IN_TASK
724  *     /*
725  *      * For reasons of header soup (see current_thread_info()), this
726  *      * must be the first element of task_struct.
727  *      */
728  *     struct thread_info          thread_info;
729  * #endif
730  *     unsigned int                __state;
731  *
732  * #ifdef CONFIG_PREEMPT_RT
733  *     /* saved state for "spinlock sleepers" */
734  *     unsigned int                saved_state;
735  * #endif
736  *
737  *     /*
738  *      * This begins the randomizable portion of task_struct. Only
739  *      * scheduling-critical items should be added above here.
740  *      */
741  *     randomized_struct_fields_start
742  *
743  *     void                      *stack;
744  *     refcount_t                 usage;
745  *     /* Per task flags (PF_*), defined further below: */
746  *     unsigned int                flags;
747  *     unsigned int                ptrace;
748  *
749  * #ifdef CONFIG_SMP
750  *     int                        on_cpu;
751  *     struct __call_single_node   wake_entry;
752  *     unsigned int                wakee_flips;
753  *     unsigned long               wakee_flip_decay_ts;
754  *     struct task_struct         *last_wakee;
755  *
756  *     /*
757  *      * recent_used_cpu is initially set as the last CPU used by a task
758  *      * that wakes affine another task. Waker/wakee relationships can
759  *      * push tasks around a CPU where each wakeup moves to the next one.
760  *      * Tracking a recently used CPU allows a quick search for a recently
761  *      * used CPU that may be idle.
762  *      */
763  *     int                        recent_used_cpu;
764  *     int                        wake_cpu;
765  * #endif
766  *     int                        on_rq;
767  *
768  *     int                        prio;
769  *     int                        static_prio;
770  *     int                        normal_prio;
771  *     unsigned int               rt_priority;
772  *
773  *     struct sched_entity        se;
774  *     struct sched_rt_entity     rt;
775  *     struct sched_dl_entity     dl;
776  *     const struct sched_class   *sched_class;
777  *
778  * #ifdef CONFIG_SCHED_CORE
779  *     struct rb_node             core_node;
780  *     unsigned long              core_cookie;
781  *     unsigned int               core_occupation;
782  * #endif
783  *
784  * #ifdef CONFIG_CGROUP_SCHED
785  *     struct task_group          *sched_task_group;
786  * #endif
787  *
788  * #ifdef CONFIG_UCLAMP_TASK
789  *     /*
790  * 
```

Windows PCB

▶ NtQueryInformationProcess

NtQueryInformationProcess function (winternl.h)

아티클 • 2021. 11. 24. • 읽는 데 4분 걸림



[NtQueryInformationProcess may be altered or unavailable in future versions of Windows. Applications should use the alternate functions listed in this topic.]

Retrieves information about the specified process.

▶ Syntax

C++

복사

```
__kernel_entry NTSTATUS NtQueryInformationProcess(
    [in]          HANDLE      ProcessHandle,
    [in]          PROCESSINFOCLASS ProcessInformationClass,
    [out]         PVOID       ProcessInformation,
    [in]          ULONG       ProcessInformationLength,
    [out, optional] PULONG     ReturnLength
);
```

대기 큐 → 리스트!

- 책) 각 자원을 대기할때 리스트의 형태로 큐를 구성
- 당연히 **Array, heap, tree**등 뭐든 가능 가능....!

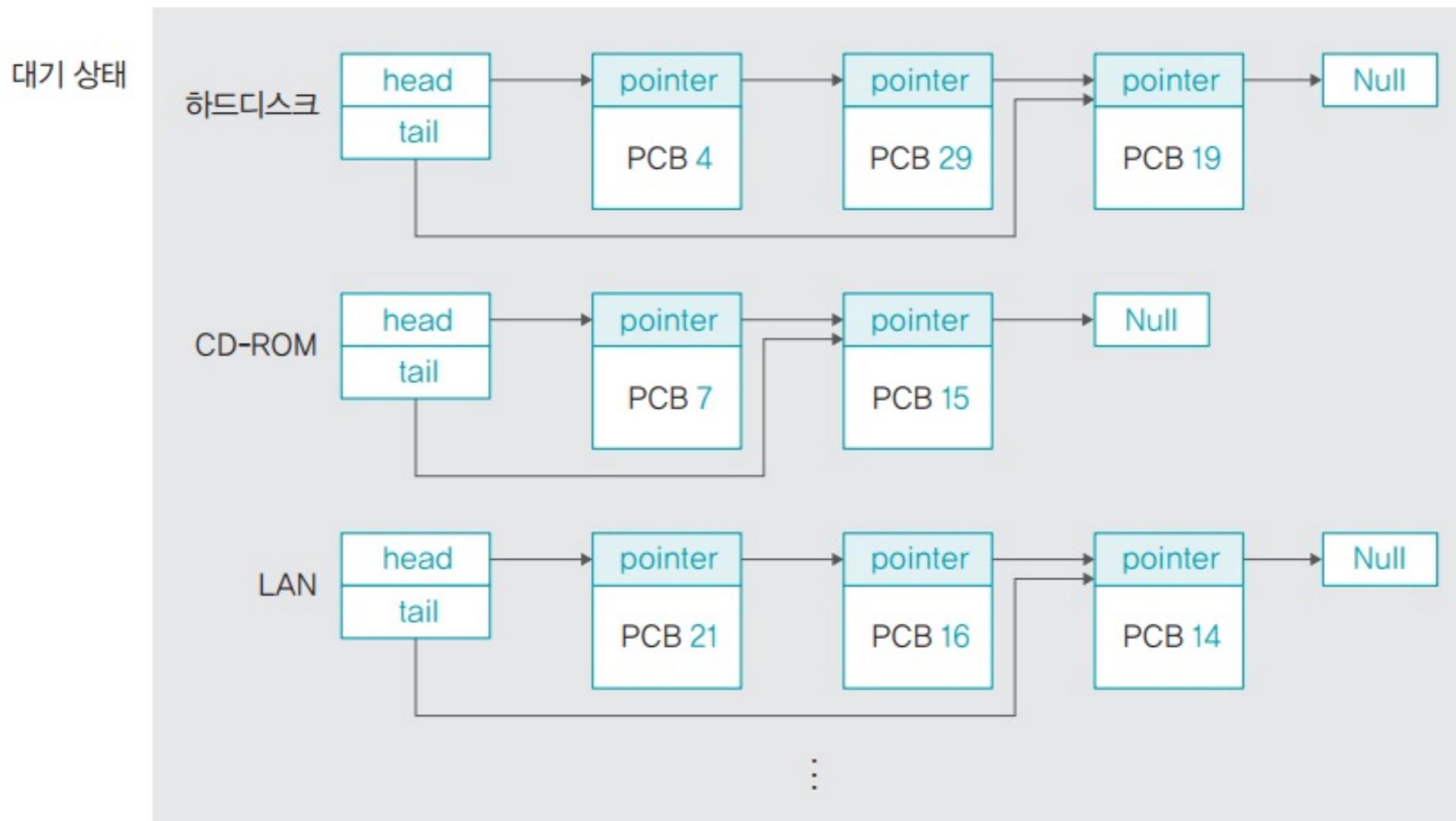
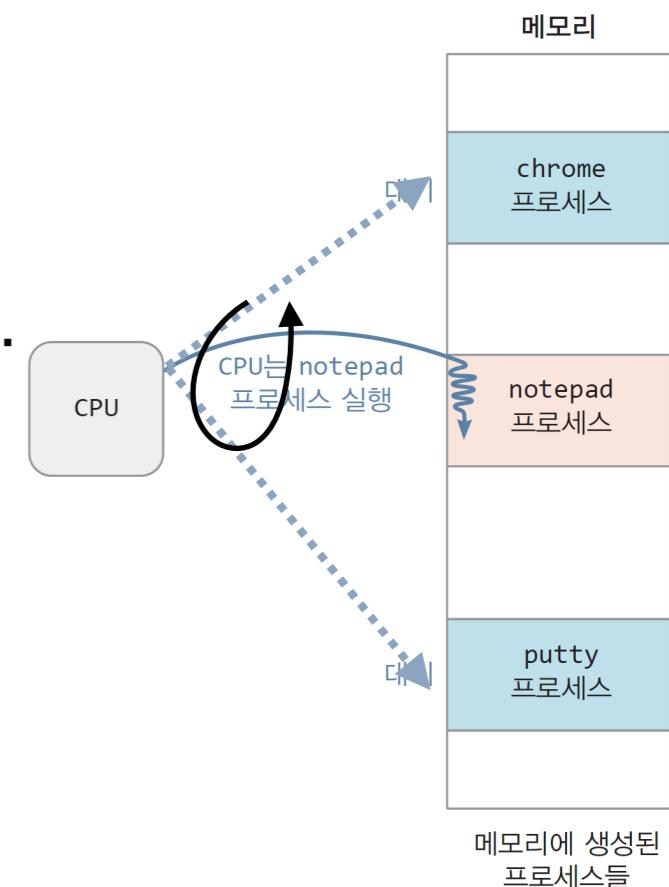


그림 3-13 대기 상태의 대기 큐

그리고, 다중 프로그래밍의 동작 원리를 생각해보면...

- ▶ 메모리에 프로세스가 여러개 있다고 해도, **CPU는 어쨌든 한번에 하나만 처리!**
 - 어떻게 여러개의 프로세스를 동시에 처리할 수 있을까?
- ▶ 시분할 (Time-slicing)!
 - **프로세스들에게 번갈아가며 CPU를 사용하게 하자**
 - 주의! 진짜로 동시에 실행되는 건 아님.
 - e.g., chrome → notepad → putty → chrome → notepad ...
 - 어떤 순서로 ‘번갈아가며’ 작업하게 할 것인가? → **스케줄링 문제**
 - 어떻게 번갈아 가게 하지? → **Context switching**



문맥 교환 Context switching

- ▶ 한 프로세스에서 다른 프로세스로 CPU를 넘겨주는 과정
- ▶ 실행 상태에서 나가는 프로세스의 PCB에는 지금까지의 작업 내용을 저장,
반대로 실행 상태로 들어오는 프로세스의 PCB의 내용으로 CPU가 다시 세팅

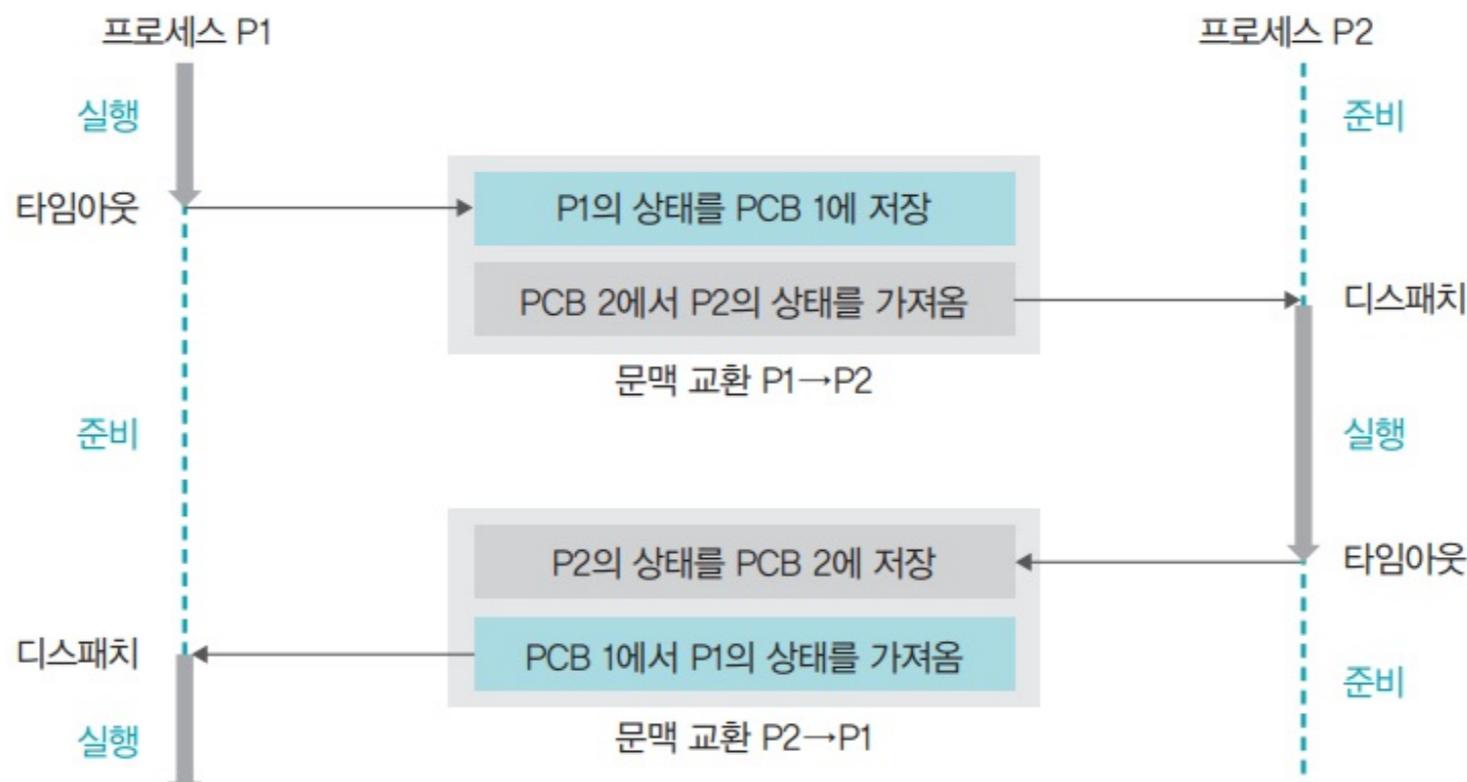


그림 3-15 문맥 교환 과정

여기까지 내용에서, 다시 고민해봐야할 문제들

- ▶ 프로그램은 로더를 통해 **메모리**에 적재되어야 한다! → 프로세스
 - 그럼 로더는 누가 ‘실행’ 하는 것인가?
 - 메모리 어디에? 그리고 어떻게? → Allocation 문제
 - 프로그램의 크기가 매우 크다면? → 가상 메모리 및 Paging
 - Context switching이 Suspend에 걸쳐 이루어 진다면??
→ Swapping 및 단편화 문제
 - 어떤 프로세스가 다른 프로세스 메모리 영역을 침범한다면? → 보안 문제



프로세스 메모리 구조

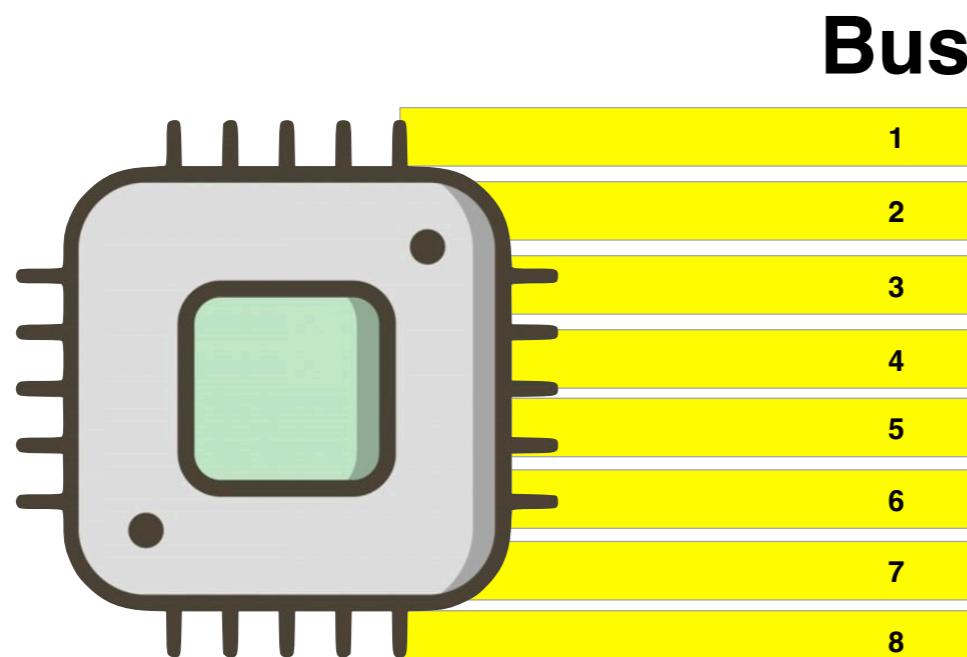
그런데 로더가 프로세스를 메모리에 무작정 올리는게 아니에요!

메모리를 ‘잘’쓰기 위한 몇가지 방법(규칙)들이 있어요!

메모리 주소



- ▶ 메모리는 CPU의 노트!
- ▶ 메모리 주소는 메모리내 위치의 식별자



Address	Data
0000 0000	0000 0000
0000 0001	0000 0000
0000 0010	0000 0000
0000 0011	0000 0000
0000 0100	0000 0000
0000 0101	0000 0000
0000 0110	0000 0000
0000 0111	0000 0000
0000 1000	0000 0000
0000 1001	0000 0000
0000 1010	0000 0000
0000 1011	0000 0000
0000 1100	0000 0000

CPU 주소 공간 (CPU address space)

- ▶ 물리 메모리 (주소) 라고도 함!
- ▶ CPU Bus의 크기에 의해 결정 == Word == CPU 아키텍쳐 크기
 - 32비트 CPU → 32개의 주소선 → 2^{32} 의 범위 → 4GB
 - 64비트 CPU → 64개의 주소선 → 2^{64} 의 범위 → 2,305,843.01 TB....!! → 16EB
- 주소 공간은 0번지부터 시작
- 1번지의 저장 공간 크기는 1바이트
- ▶ 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 (up to 4GB)에 2GB의 메모리가 설치되어 있을 때
→ 2GB를 넘어서 액세스하면 오류 발생

저 메모리 공간을 여러 프로세스가 나눠 써야합니다!

프로세스 주소 공간

- ▶ 프로세스가 실행중에 접근할 수 있도록 허용된 주소의 최대 범위 → **Segmentation**
 - 할당된 공간에 대한 경계 레지스터와 한계 레지스터를 벗어나는지를 감시
 - 이것은 CPU관점
- ▶ 하지만, 프로세스 관점에서는 ‘가상메모리’라는 개념이 적용됩니다! → 프로세스는 자신이 CPU 주소 공간 전체를 독점하는 것 처럼 보여요! (**Logical memory vs Physical memory**)
 - 모든 프로세스는 자신만의 가상 주소 공간을 가짐
 - 32bit CPU에서 작동되는 프로세스? 논리적으로 4GB의 메모리를 할당 받은 것 처럼 보임 (물리 메모리가 2GB라더라도)
- ▶ 프로세스 주소 공간은 2부분으로 나뉘어짐
 - 사용자 공간 User space: 코드, 데이터, 힙, 스택 영역
 - 커널 공간 Kernel space
 - 프로세스가 시스템 호출을 통해 이용하는 커널 공간
 - 커널 코드, 커널 데이터, 커널 스택(커널 코드가 실행될 때)
 - 커널 공간은 모든 사용자 프로세스에 의해 공유

왜 가상 메모리를 사용할까?

▶ 메모리에 대한 확장성!

- 물리적 메모리는 한정적이지만, 가상 메모리는 더 큰 공간으로 구성 가능!
- 초과 분은 보조기억장치등을 활용, 가상 주소 공간은 저장장치의 구분 없이 하나의 가상 공간으로 활용 가능 → Page swap등이 필요 (나중에 배움)

▶ 모든 프로그램에 대한 동일한 메모리 공간 제공

- 각 프로세스는 다른 프로세스를 신경 쓸 필요가 없음
- 각 프로세스간 메모리 격리 (Memory protection) → 보호

▶ RTOS등은 가상 메모리를 사용하지 않고 직접 접근 하기도 함

프로세스가 메모리에 올라 갈 때

▶ 1. 코드(code) 영역 (크기가 Compiled time 결정됨)

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

프로그램이 프로세스가 될 때,

OS가 프로그램을 이렇게 적재 시킵니다!

▶ 2. 데이터(data) 영역 (크기가 Compiled time 결정됨)

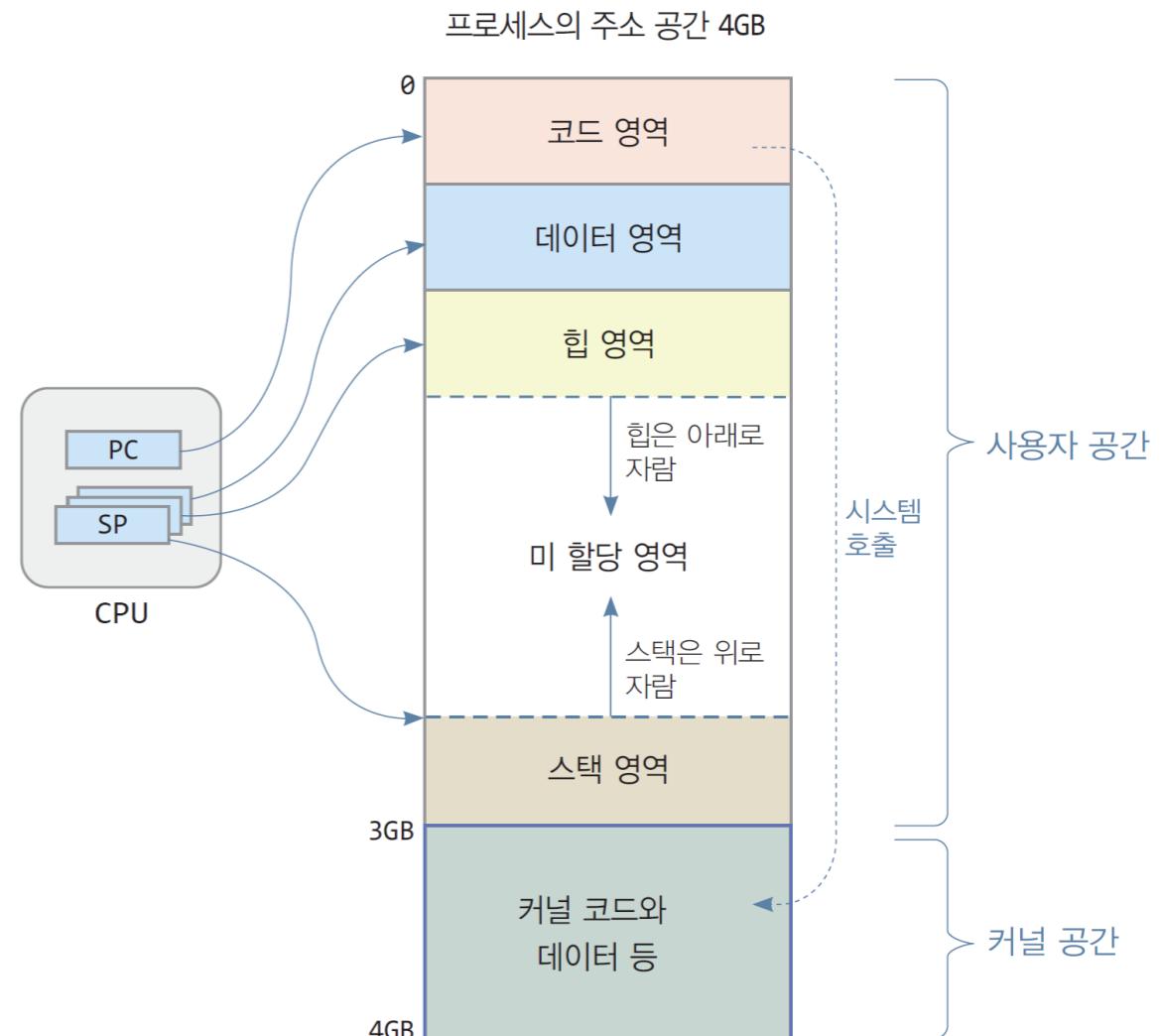
- 전역 변수 (Global) 공간, 정적 데이터 (Static) 공간
 - rdata, data, bss 등으로 구분됨
 - 사용자 프로그램과 라이브러리 포함
- 프로세스 적재 시 할당, 종료 시 소멸

▶ 3. 힙(heap) 영역 (크기가 runtime 결정됨)

- 프로세스가 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 (크기가 runtime 결정됨)

- 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 return하면 할당된 공간 반환

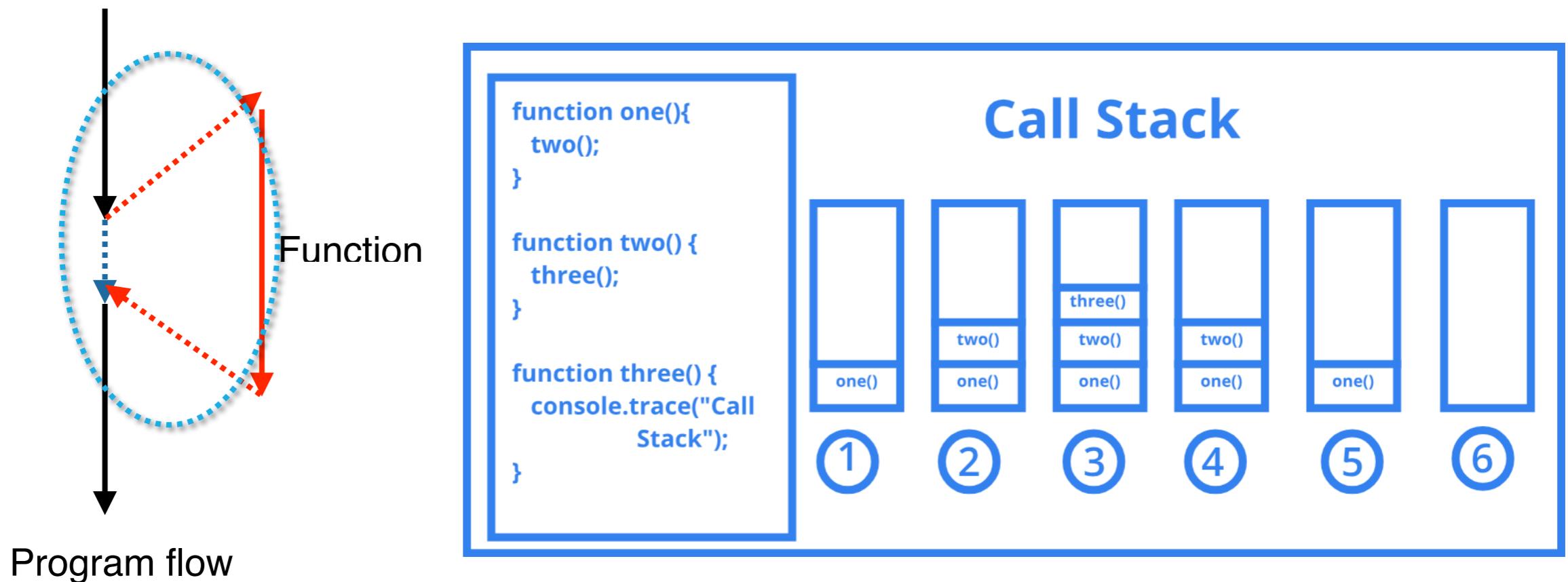


왜 이렇게 생겼을까?

- ▶ 일단 나눈 가장 큰 이유: 데이터를 공유하여 메모리 사용량을 줄이기 위해, 스택 구조와 전역변수의 활용성을 위해
- ▶ **Code** 영역이 구분 된 이유
 - 프로그램 Code는 프로그램이 만들어지고 (i.e., 컴파일) 나서는 바뀔 일이 전혀 없음 → 즉, Read-Only
 - 같은 프로세스가 여러개의 프로세스로서 실행된다면? → 코드영역을 공유(Share) 함으로서 메모리 사용량을 줄일 수 있음.
- ▶ **Data** 영역이 구분 된 이유
 - 전역 변수와 정적(static) 변수가 저장되는 영역 → 변수니깐, Read-Write! (c.f., 리터럴(literal) 들은 read-only)
 - 전역 및 정적변수 → 프로그램이 구동되는 동안은 항상 접근 가능해야함 → 프로그램 실행과 관계없이 독립적인 영역이 필요
- ▶ **Heap** 영역이 구분된 이유?; **Heap**과 **Stack**이 자라나는 방향이 다른 이유?
 - 프로그램이 실행 도중에 필요할 때마다 할당 받는 공간 → 얼마나 필요할지 예측 불가!
 - Stack도 실행 상태에 따라 얼마나 사용될지 모름 → 동적 공간에서 서로 공간을 유연하게 활용

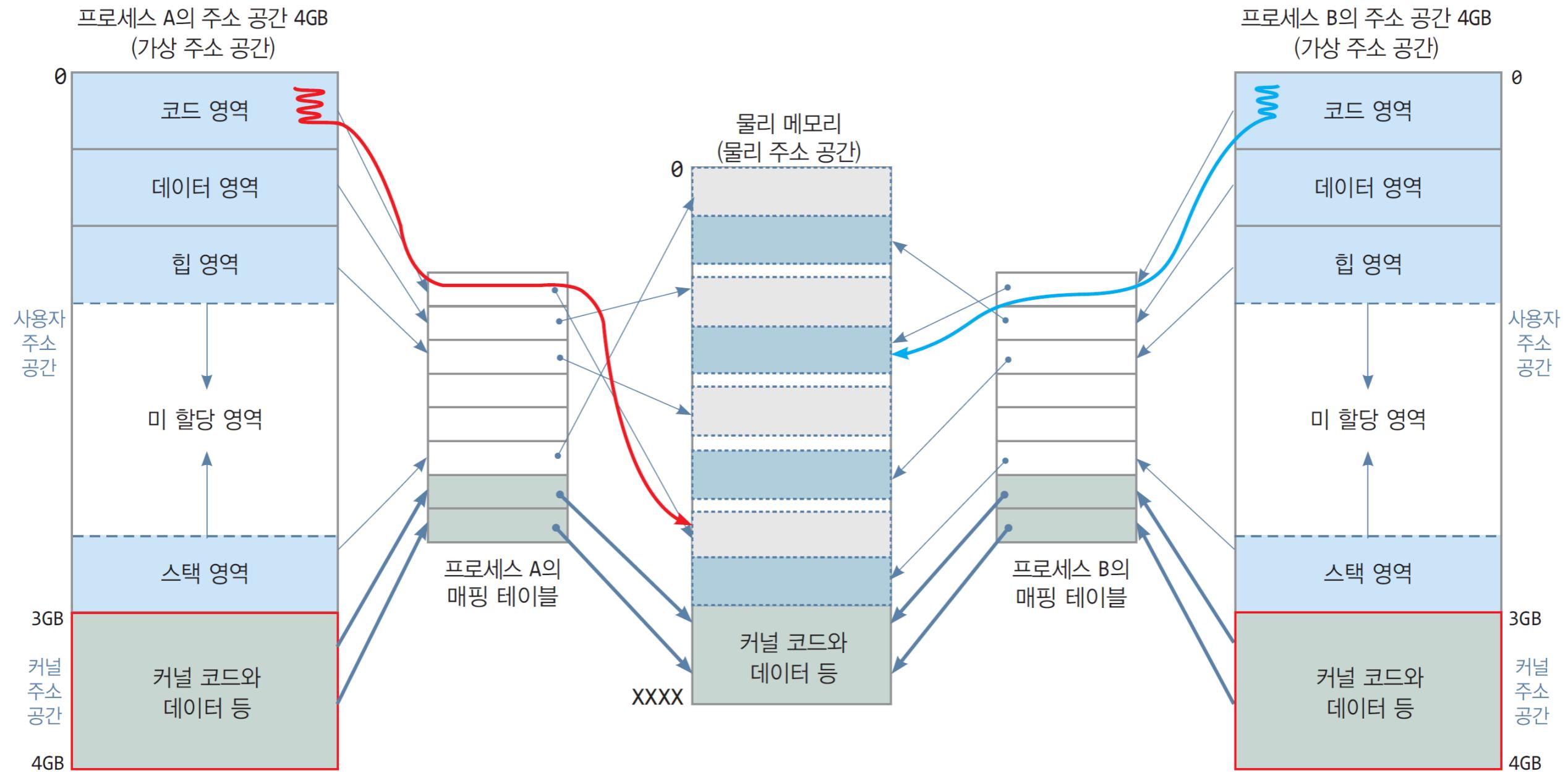
왜 이렇게 생겼을까? (2)

- ▶ Stack 영역이 분리된 이유
 - 함수는 프로그램의 실행 단위! 프로그램은 함수의 호출로 이루어져 있음
 - 지역변수 및 매개변수, 반환 값등의 존재
 - 스택 구조를 이용하면 함수의 호출 순서와 반환 대상등의 관리가 편함
 - Callstack: 스택 프레임을 통한 함수의 실행 순서 구분.



프로세스 주소 to 물리 주소

- ▶ (가상메모리에서 다시 나와요!)



프로세스 메모리 구조

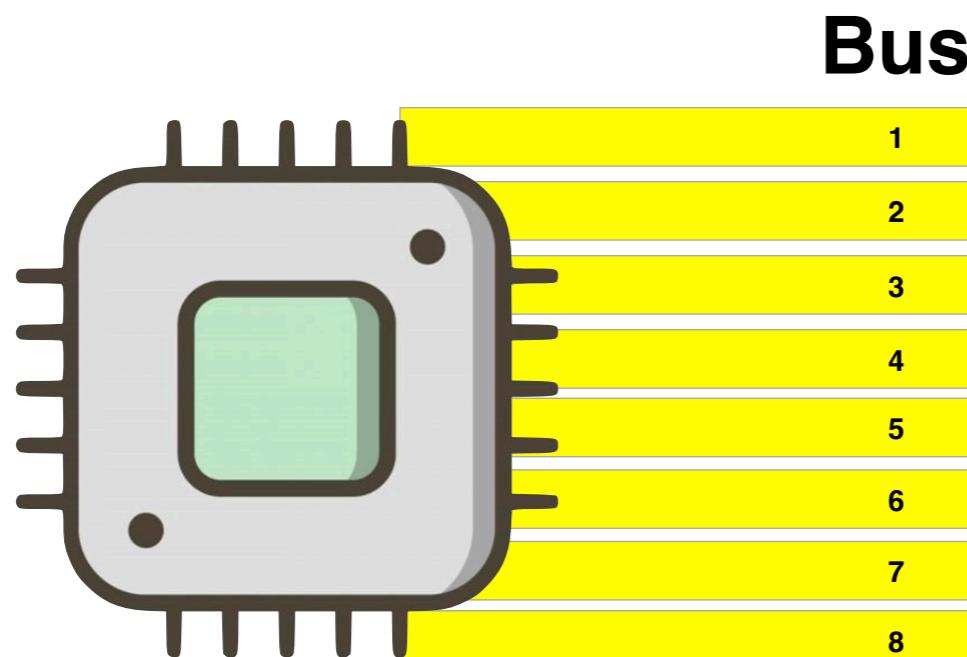
그런데 로더가 프로세스를 메모리에 무작정 올리는게 아니에요!

메모리를 ‘잘’쓰기 위한 몇가지 방법(규칙)들이 있어요!

메모리 주소



- ▶ 메모리는 CPU의 노트!
- ▶ 메모리 주소는 메모리내 위치의 식별자



Address	Data
0000 0000	0000 0000
0000 0001	0000 0000
0000 0010	0000 0000
0000 0011	0000 0000
0000 0100	0000 0000
0000 0101	0000 0000
0000 0110	0000 0000
0000 0111	0000 0000
0000 1000	0000 0000
0000 1001	0000 0000
0000 1010	0000 0000
0000 1011	0000 0000
0000 1100	0000 0000

CPU 주소 공간 (CPU address space)

- ▶ 물리 메모리 (주소) 라고도 함!
- ▶ CPU Bus의 크기에 의해 결정 == Word == CPU 아키텍쳐 크기
 - 32비트 CPU → 32개의 주소선 → 2^{32} 의 범위 → 4GB
 - 64비트 CPU → 64개의 주소선 → 2^{64} 의 범위 → 2,305,843.01 TB....!! → 16EB
- 주소 공간은 0번지부터 시작
- 1번지의 저장 공간 크기는 1바이트
- ▶ 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 (up to 4GB)에 2GB의 메모리가 설치되어 있을 때
→ 2GB를 넘어서 액세스하면 오류 발생

저 메모리 공간을 여러 프로세스가 나눠 써야합니다!

프로세스 주소 공간

- ▶ 프로세스가 실행중에 접근할 수 있도록 허용된 주소의 최대 범위 → **Segmentation**
 - 할당된 공간에 대한 경계 레지스터와 한계 레지스터를 벗어나는지를 감시
 - 이것은 CPU관점
- ▶ 하지만, 프로세스 관점에서는 ‘가상메모리’라는 개념이 적용됩니다! → 프로세스는 자신이 CPU 주소 공간 전체를 독점하는 것처럼 보여요! (**Logical memory vs Physical memory**)
 - 모든 프로세스는 자신만의 가상 주소 공간을 가짐
 - 32bit CPU에서 작동되는 프로세스? 논리적으로 4GB의 메모리를 할당 받은 것처럼 보임 (물리 메모리가 2GB라더라도)
- ▶ 프로세스 주소 공간은 2부분으로 나뉘어짐
 - 사용자 공간 User space: 코드, 데이터, 힙, 스택 영역
 - 커널 공간 Kernel space
 - 프로세스가 시스템 호출을 통해 이용하는 커널 공간
 - 커널 코드, 커널 데이터, 커널 스택(커널 코드가 실행될 때)
 - 커널 공간은 모든 사용자 프로세스에 의해 공유

왜 가상 메모리를 사용할까?

▶ 메모리에 대한 확장성!

- 물리적 메모리는 한정적이지만, 가상 메모리는 더 큰 공간으로 구성 가능!
- 초과 분은 보조기억장치등을 활용, 가상 주소 공간은 저장장치의 구분 없이 하나의 가상 공간으로 활용 가능 → Page swap등이 필요 (나중에 배움)

▶ 모든 프로그램에 대한 동일한 메모리 공간 제공

- 각 프로세스는 다른 프로세스를 신경 쓸 필요가 없음
- 각 프로세스간 메모리 격리 (Memory protection) → 보호

▶ RTOS등은 가상 메모리를 사용하지 않고 직접 접근 하기도 함

프로세스가 메모리에 올라 갈 때

▶ 1. 코드(code) 영역 (크기가 Compiled time 결정됨)

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

프로그램이 프로세스가 될 때,

OS가 프로그램을 이렇게 적재 시킵니다!

▶ 2. 데이터(data) 영역 (크기가 Compiled time 결정됨)

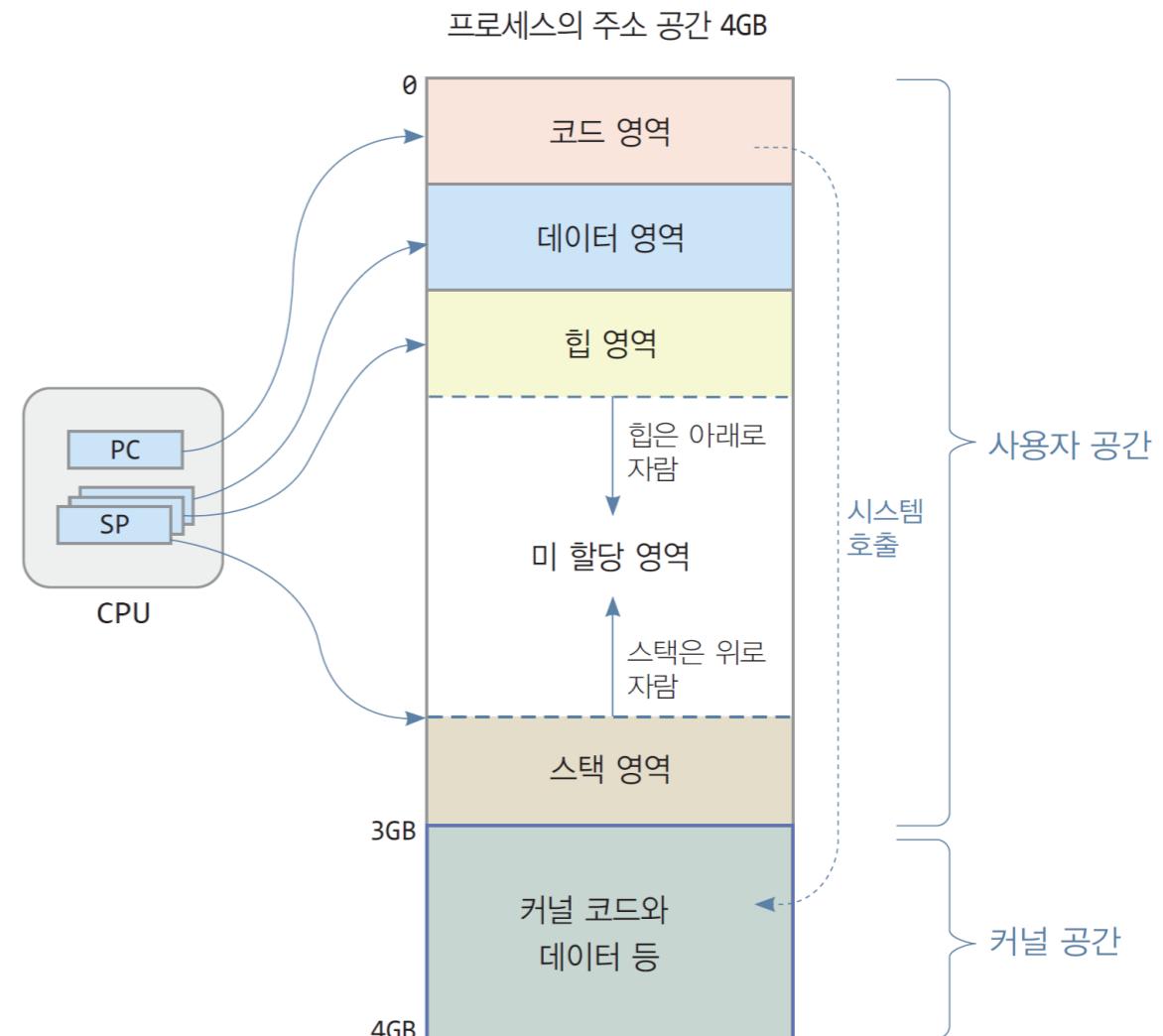
- 전역 변수 (Global) 공간, 정적 데이터 (Static) 공간
 - rdata, data, bss 등으로 구분됨
 - 사용자 프로그램과 라이브러리 포함
- 프로세스 적재 시 할당, 종료 시 소멸

▶ 3. 힙(heap) 영역 (크기가 runtime 결정됨)

- 프로세스가 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 (크기가 runtime 결정됨)

- 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 return하면 할당된 공간 반환

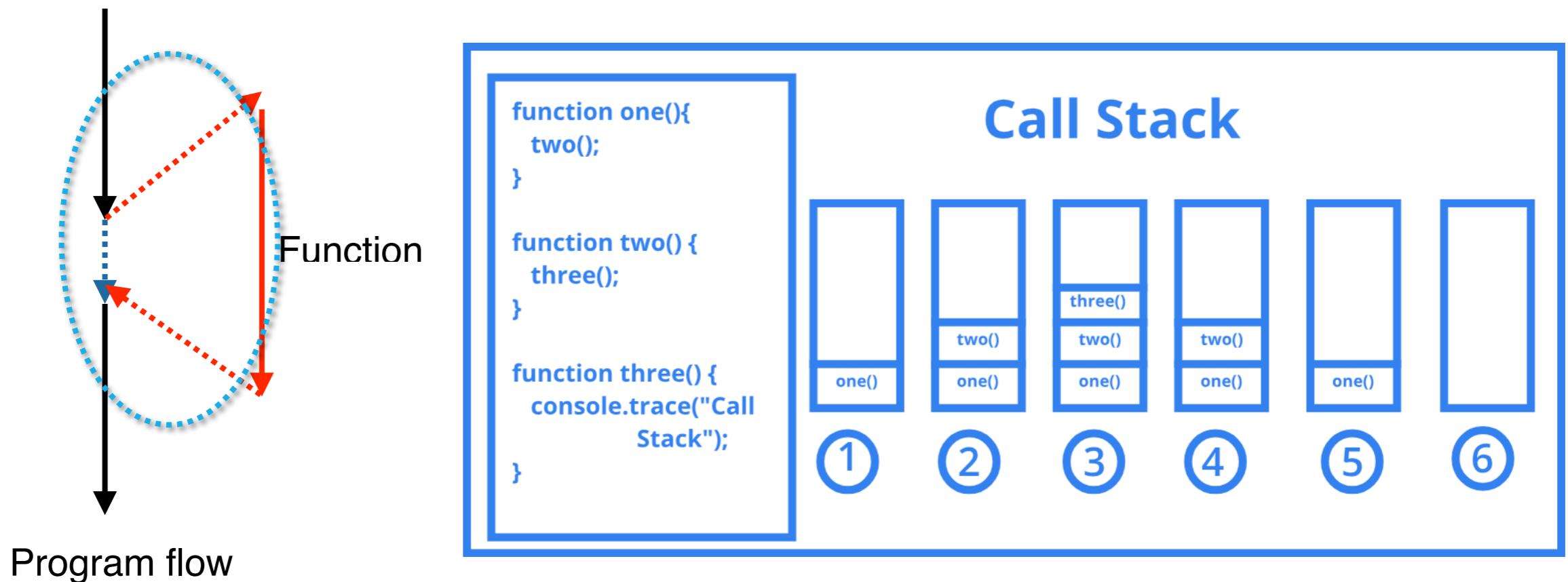


왜 이렇게 생겼을까?

- ▶ 일단 나눈 가장 큰 이유: 데이터를 공유하여 메모리 사용량을 줄이기 위해, 스택 구조와 전역변수의 활용성을 위해
- ▶ **Code** 영역이 구분 된 이유
 - 프로그램 Code는 프로그램이 만들어지고 (i.e., 컴파일) 나서는 바뀔 일이 전혀 없음 → 즉, Read-Only
 - 같은 프로세스가 여러개의 프로세스로서 실행된다면? → 코드영역을 공유(Share) 함으로서 메모리 사용량을 줄일 수 있음.
- ▶ **Data** 영역이 구분 된 이유
 - 전역 변수와 정적(static) 변수가 저장되는 영역 → 변수니깐, Read-Write! (c.f., 리터럴(literal) 들은 read-only)
 - 전역 및 정적변수 → 프로그램이 구동되는 동안은 항상 접근 가능해야함 → 프로그램 실행과 관계없이 독립적인 영역이 필요
- ▶ **Heap** 영역이 구분된 이유?; **Heap**과 **Stack**이 자라나는 방향이 다른 이유?
 - 프로그램이 실행 도중에 필요할 때마다 할당 받는 공간 → 얼마나 필요할지 예측 불가!
 - Stack도 실행 상태에 따라 얼마나 사용될지 모름 → 동적 공간에서 서로 공간을 유연하게 활용

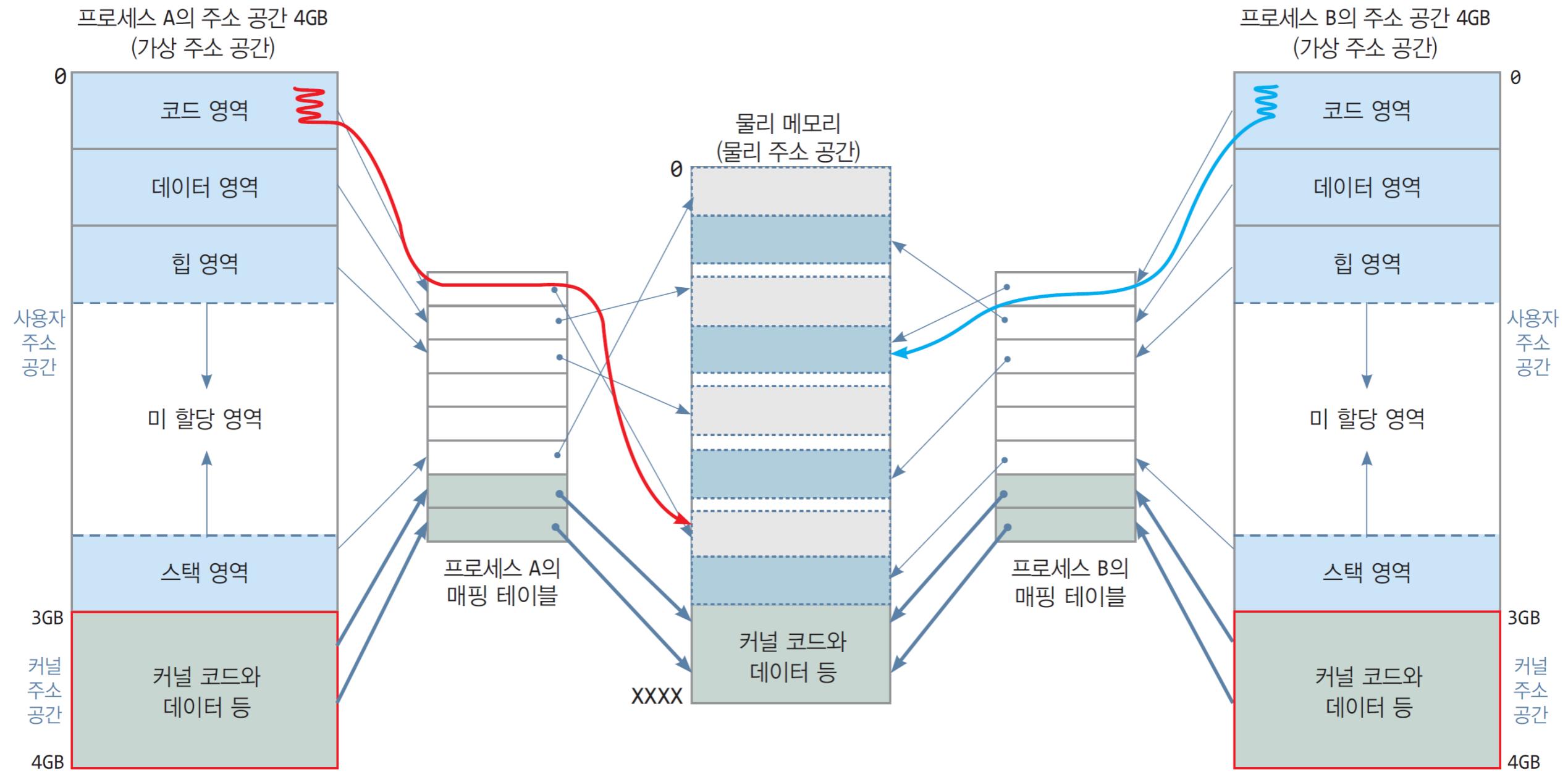
왜 이렇게 생겼을까? (2)

- ▶ Stack 영역이 분리된 이유
 - 함수는 프로그램의 실행 단위! 프로그램은 함수의 호출로 이루어져 있음
 - 지역변수 및 매개변수, 반환 값등의 존재
 - 스택 구조를 이용하면 함수의 호출 순서와 반환 대상등의 관리가 편함
 - Callstack: 스택 프레임을 통한 함수의 실행 순서 구분.



프로세스 주소 to 물리 주소

- (가상메모리에서 다시 나와요!)



Human OS!

프로세스 구성 영역을 직접 그려보자 (1)

- ▶ 다음 프로그램이 실행을 시작하여 **실행 한 직후** 상태의 프로세스 메모리 공간을 그려봅시다
(즉, main()이 호출되자 말자)

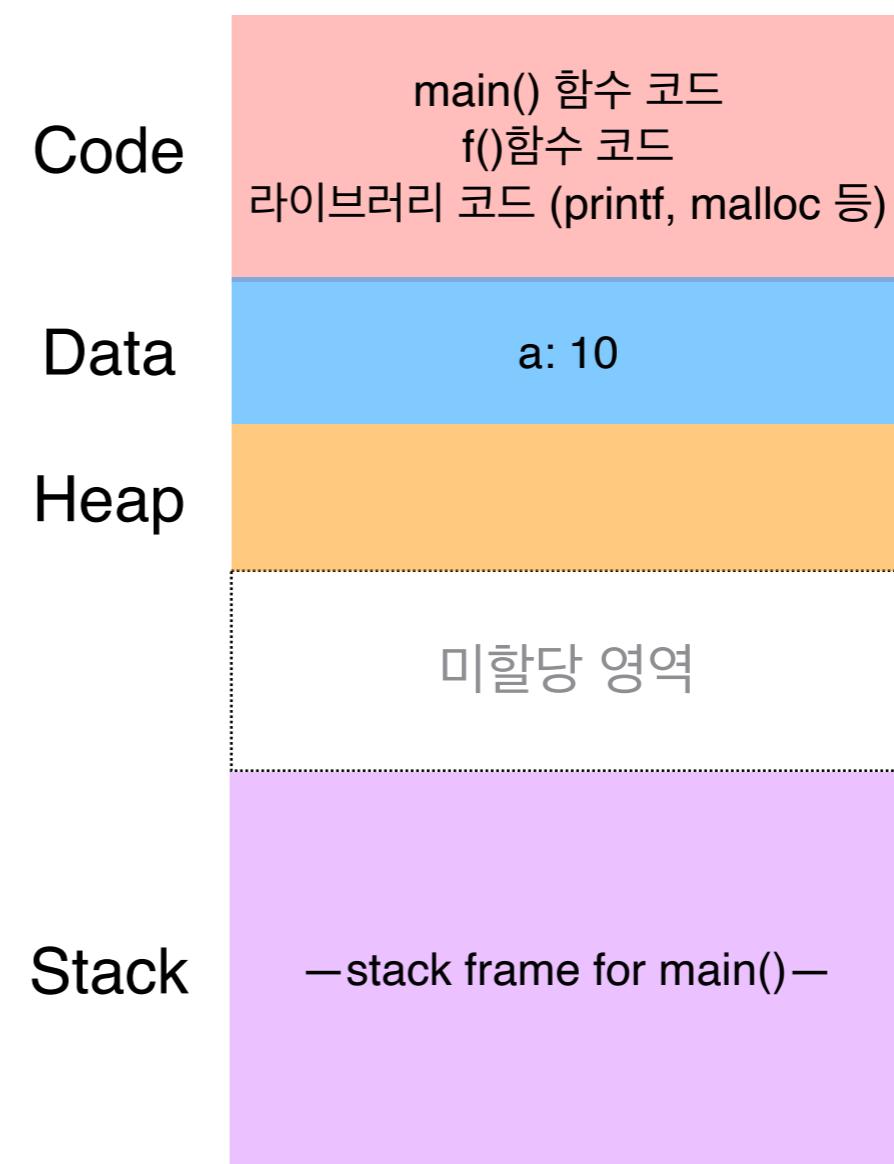
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

Human OS!

프로세스 구성 영역을 직접 그려보자 (2)

- ▶ Code, data 영역에 각각 코드와 전역변수가 배치됨
- ▶ Stack영역에 main 함수에 대한 스택프레임 생성됨
- ▶ 다음, 13번째 까지 실행을 한다면?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

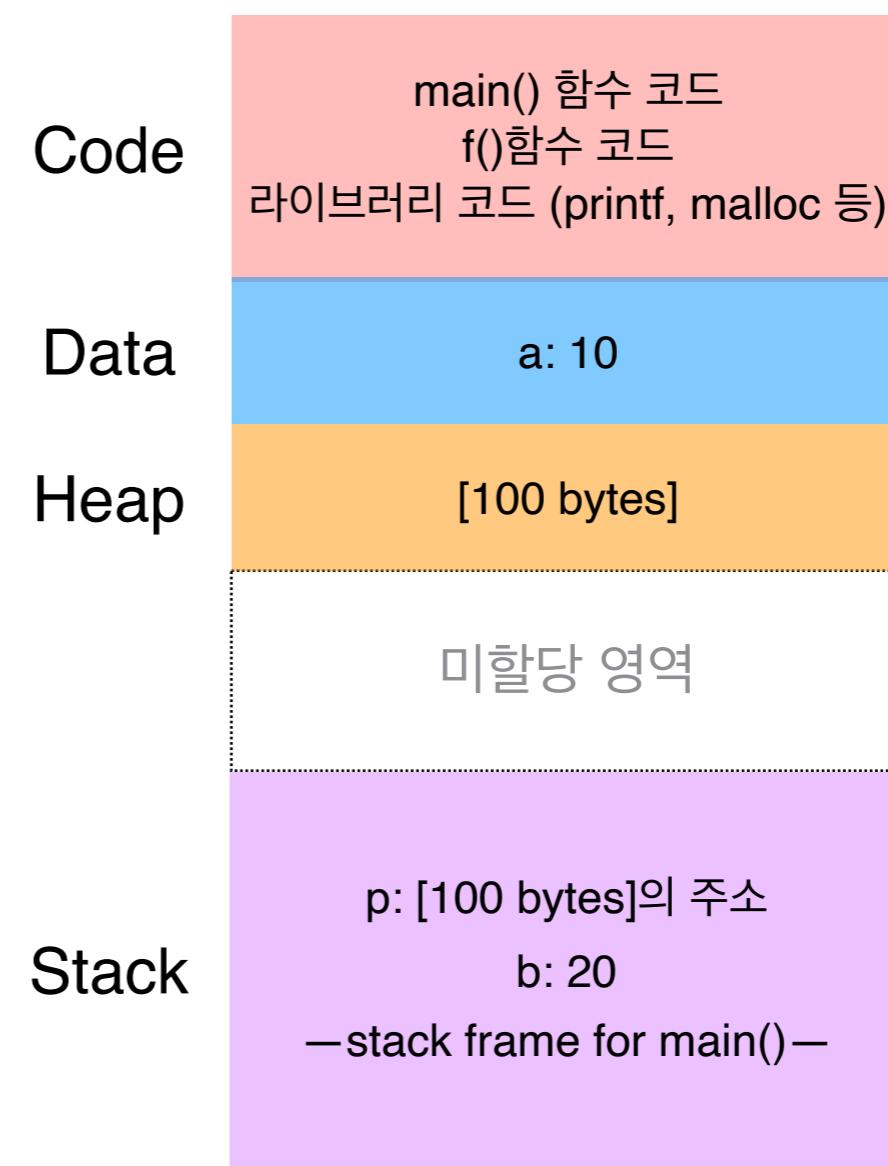


Human OS!

프로세스 구성 영역을 직접 그려보자 (3)

- ▶ 스택에 b, p에 대한 변수 공간 할당
- ▶ 힙영역에 100바이트 크기만큼의 공간 할당
- ▶ 다음, 14→f()호출→8번째 까지 실행을 한다면?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

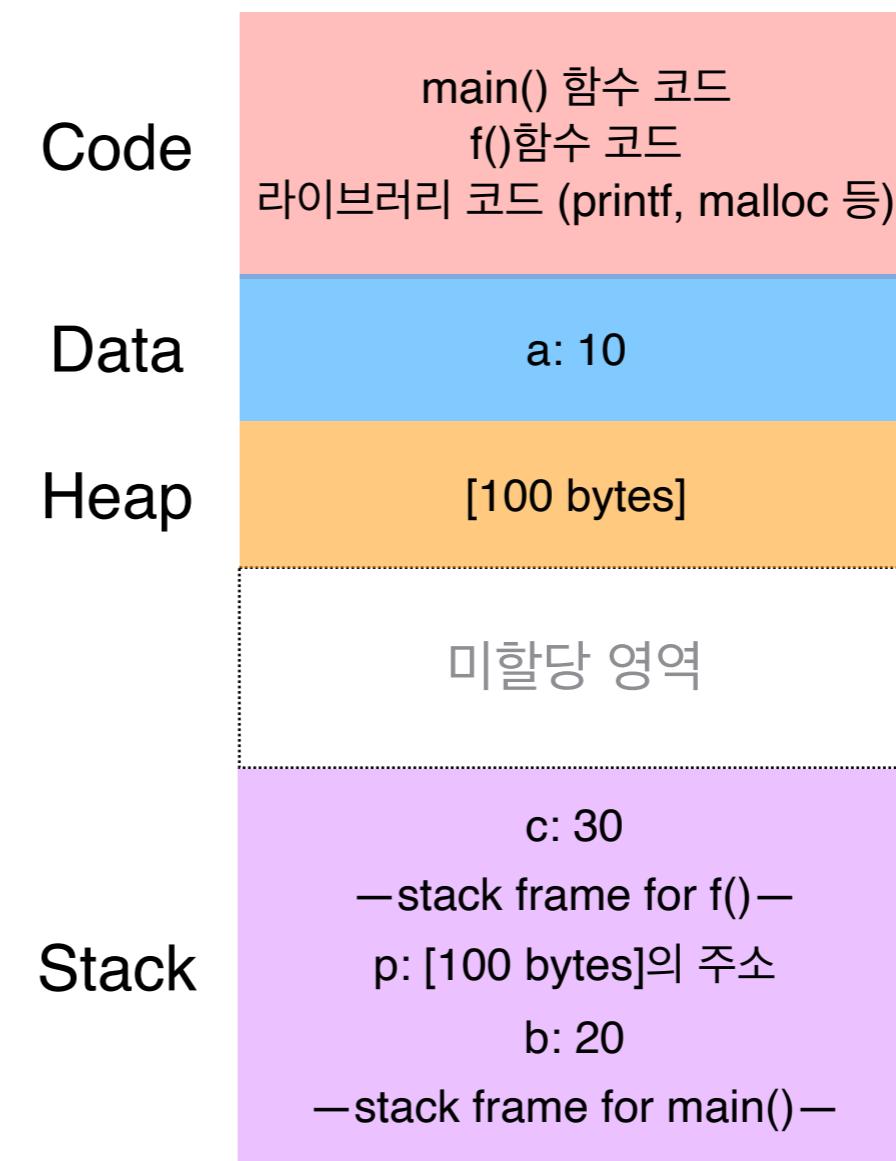


Human OS!

프로세스 구성 영역을 직접 그려보자 (4)

- ▶ 함수 f()에 대한 스택 프레임 생성
- ▶ 변수 c에 대한 공간할당
- ▶ 다음, 15번째 까지 실행을 한다면?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

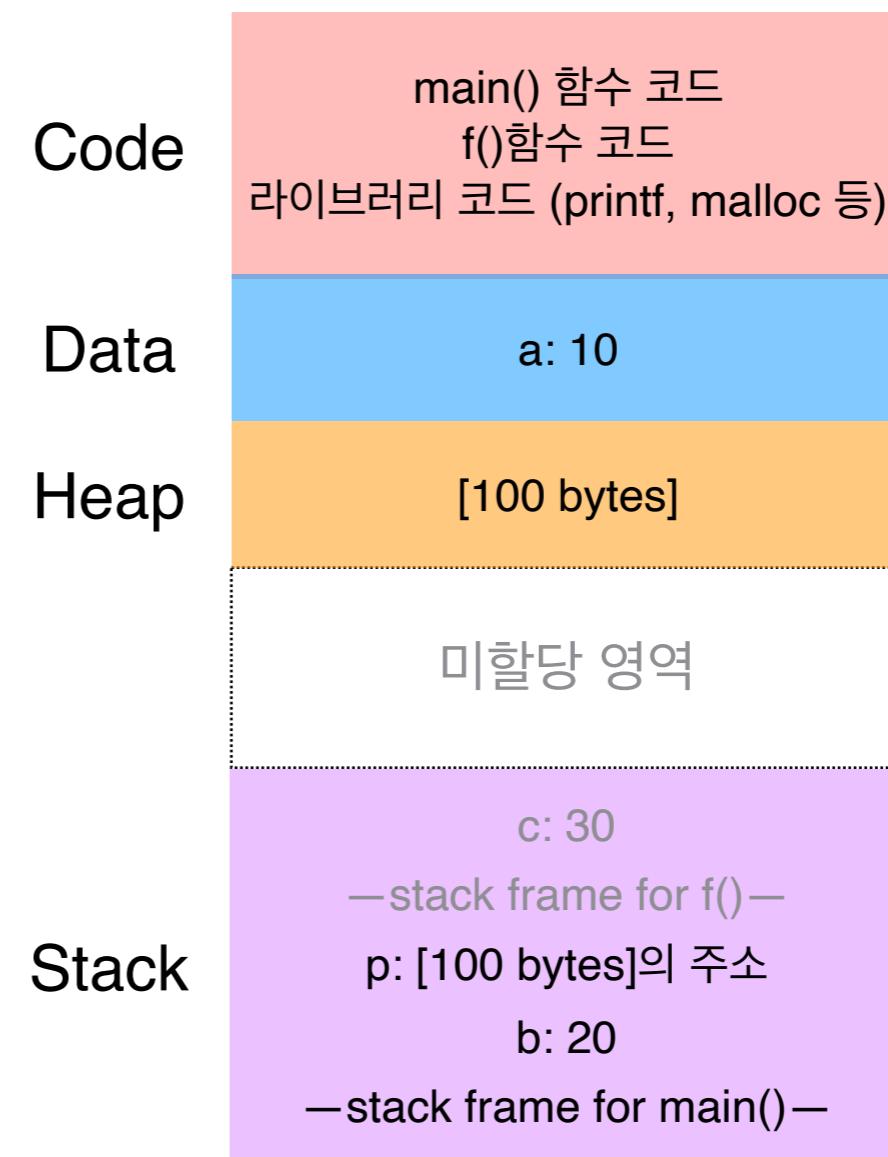


Human OS!

프로세스 구성 영역을 직접 그려보자 (5)

- ▶ f에 대한 스택프레임 제거 (변수 c는 자동으로 정리 (i.e., 쓰레기값))
- ▶ 다음, 16번째 까지 실행을 한다면?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

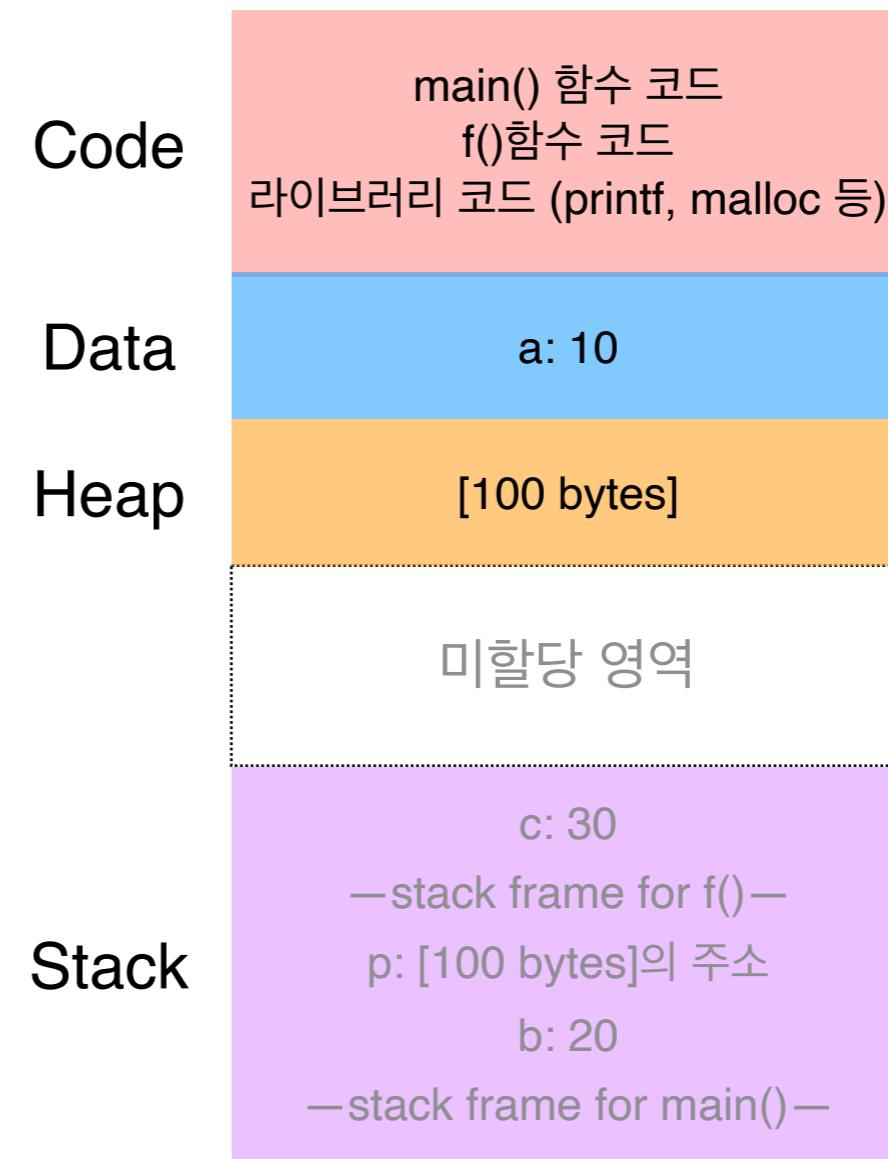


Human OS!

프로세스 구성 영역을 직접 그려보자 (6)

- ▶ main에 대한 스택프레임 제거 (변수 b, p는 자동으로 정리)
- ▶ Heap에 대한 메모리 할당은 여전히 남아있음 (Memory leak)
- ▶ 최종 실행 종료가 되면?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```

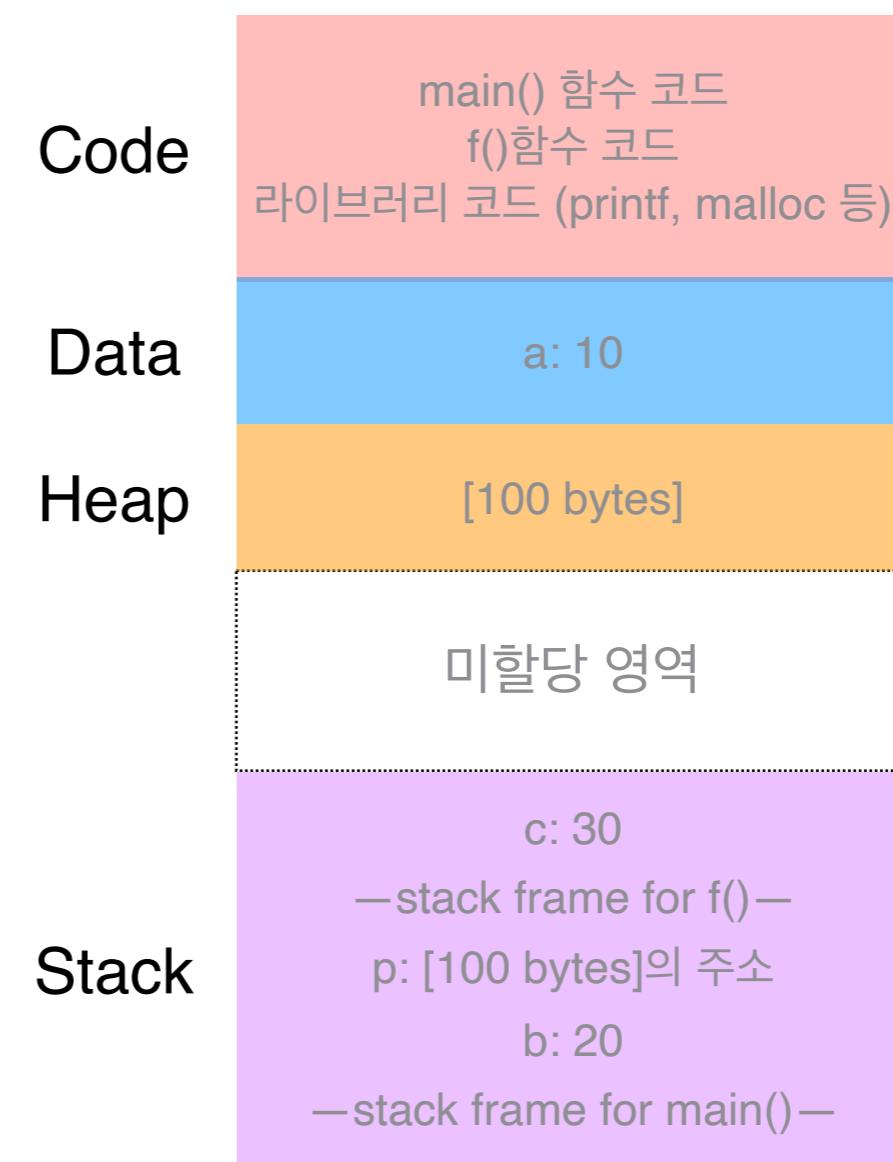


Human OS!

프로세스 구성 영역을 직접 그려보자 (7)

- ▶ 운영체제가 모든 메모리를 회수
- ▶ 만약 이걸 OS가 제대로 못해주면? Memory leak!

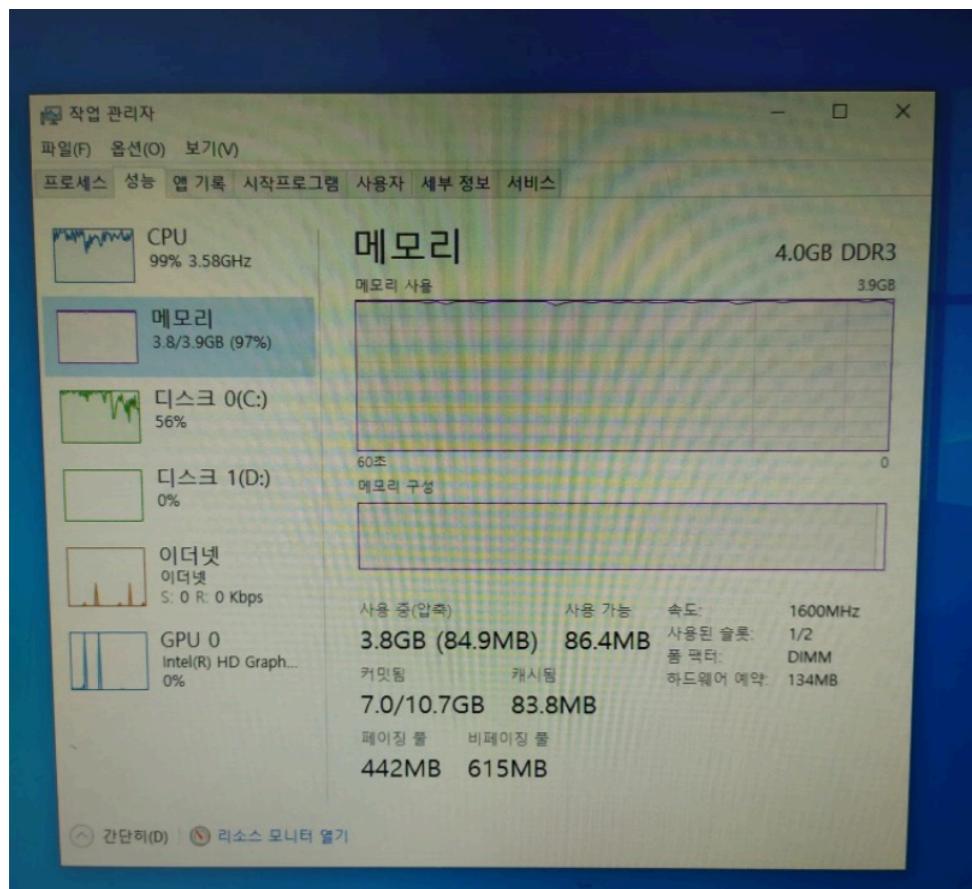
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 10;
5
6 void f() {
7     int c = 30;
8     printf("%d", c);
9 }
10
11 int main() {
12     int b = 20;
13     int *p = (int*)malloc(100);
14     f();
15     printf("%d", b);
16     return -1;
17 }
```



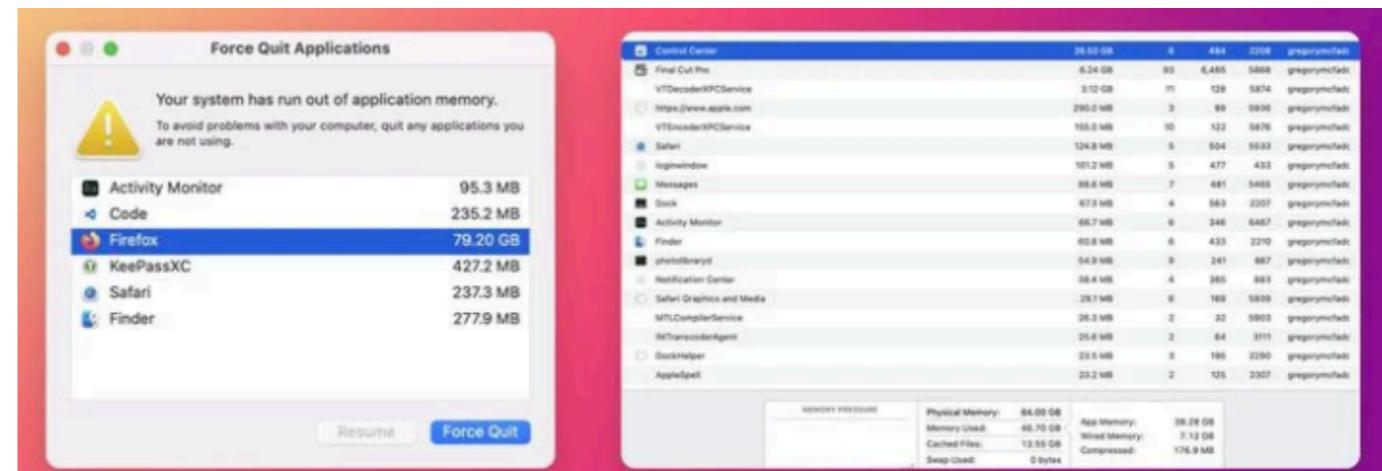
메모리 누수는 꽤나 중요한 문제

- ▶ 하지만 최신 OS에서도 꽤 흔함...
- ▶ 요런거도 잘 해주는게 OS의 중요한 요소

Window 10



macOS Monterey



프로세스의 생성과 복사

메모리에 올리는 것만으로는 프로세스가 만들어지지 않아요!

최종적으로 Context를 만들어줘야합니다! (i.e., PCB)

프로세스 생성

- ▶ 프로세스는 언제 생성되는가? → 프로세스가 생성되는 5가지 경우
 - 시스템 부팅과정에서 필요한 프로세스 생성
 - 사용자의 로그인 후 사용자와 대화(제어)를 위한 프로세스 생성 (bash, explorer.exe, finder.app)
 - 새로운 프로세스를 생성하도록 하는 사용자의 명령 (vi hello.c)
 - 배치 작업 실행 시(at, batch 명령)
 - 사용자 응용프로그램이 시스템 호출로 새 프로세스 생성
- ▶ 중요한건 메모리에 올라갔다고 단순히 프로세스가 아님!
 - 그냥 메모리에 올라'만' 와있으면, 아무런 의미가 없음!
 - PCB가 존재하여 OS가 제어 가능한 형태가 되어야함 == CPU를 할당받아 실행이 가능

프로세스 생성 과정

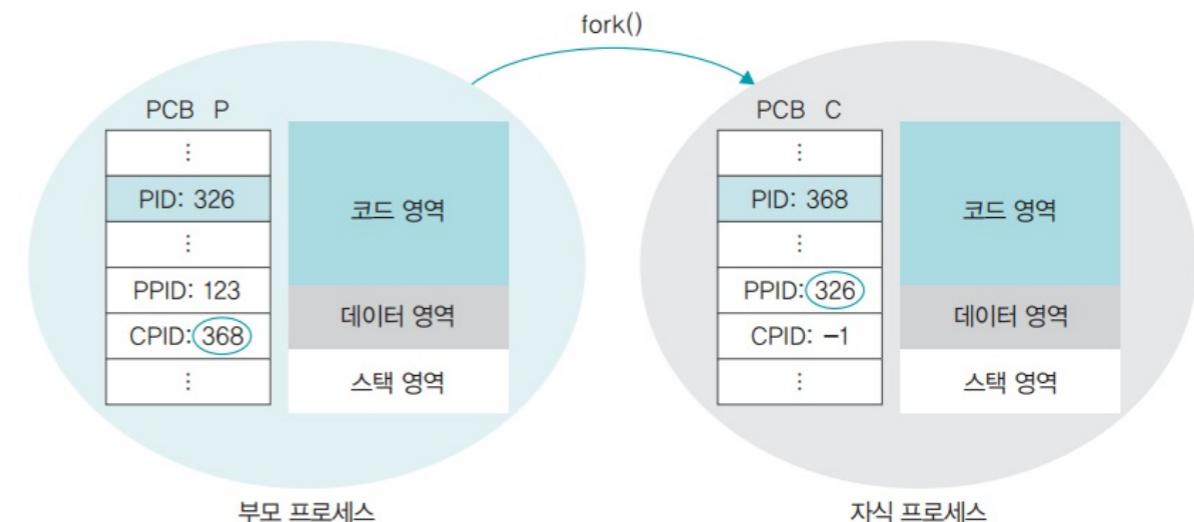
- ▶ 1. 생성하려는 실행파일의 경로를 OS에 전달
- ▶ 2. OS는 메모리에 프로그램을 적재
 - Code 영역에 프로그램의 코드를 적재시키고, Data영역에 전역/정적 변수들을 할당
 - 스택과 힙은 아직 아무것도 없으므로 초기화만 시킴
- ▶ 3. PCB 공간을 할당받고 (malloc) 필요한 정보를 채움
 - 프로세스 식별자를 결정 - 새로운 PID 번호 할당
 - 프로세스 정보 기록
 - 프로세스 테이블에서 새 항목 할당
 - 새로 할당된 프로세스 테이블에 PCB 연결
- ▶ 4. PCB에 프로세스 상태를 ready 상태로 표시하고, 준비 큐에 장착

나름 할일이 많다!: 그래서 복사에 의한 생성 fork()

- ▶ **기존에 있는 프로세스가 다른 프로세스를 생성**
 - 프로세스를 복사하는 시스템 콜을 통해서 프로세스 생성
 - 리눅스: fork() 시스템 콜
 - Windows: CreateProcess() 등 시스템 콜
 - 엄밀히 윈도우에는 fork()는 없음. 얘 자체는 fork()+exec()에 가깝습니다. 큰 맥락은 비슷!
- ▶ UNIX 계열의 OS는 시스템이 부팅 할 때 0번 프로세스 (init)만 자체적으로 생성
→ **나머지 프로세스는 '복제'를 통해 생성함**
 - 자주 사용되는 프로세스에 대해 매번 반복할 필요가 없음 (e.g., bash shell)
 - 관리상 편리해짐 (프로세스 계층 구조)
 - Process간 통신

fork() 시스템 콜

- ▶ 실행 중인 프로세스로부터 새로운 프로세스를 복사하는 함수
- ▶ 실행 중인 프로세스와 똑같은 프로세스가 하나 더 만들어짐 "int pid = fork();"
 - fork를 호출한 프로세스: 부모 프로세스, fork된 프로세스: 자식 프로세스
 - 부모 프로세스의 모든 환경, 메모리, PCB 등을 복사
 - 부모와 동일한 모양이지만, 독립된 주소 공간에 위치
 - 단, PCB에서 아래 내용은 달라짐!
 - PID는 다름! 당연히 새로운 프로세스니깐!
 - PPID (Parent PID): 부모의 PID로
 - CPID (Child PID): 자식이 없으면 -1
 - 메모리 관련 정보: 독립된 주소 공간을 소유하므로



fork() 실행 과정

- ▶ 자식은 부모의 Program Counter 도 복제하여, fork() 다음의 if(pid == 0) 라인부터 실행됨
 - 자식 프로세스는 pid = fork(); 이전 라인을 실행하지 못함
 - ▶ fork() 함수의 리턴값
 - 부모 프로세스에게는 자식 프로세스의 PID 리턴
 - 자식 프로세스에게는 0 리턴

```
#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); }

}
```

```
#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); }

}
```

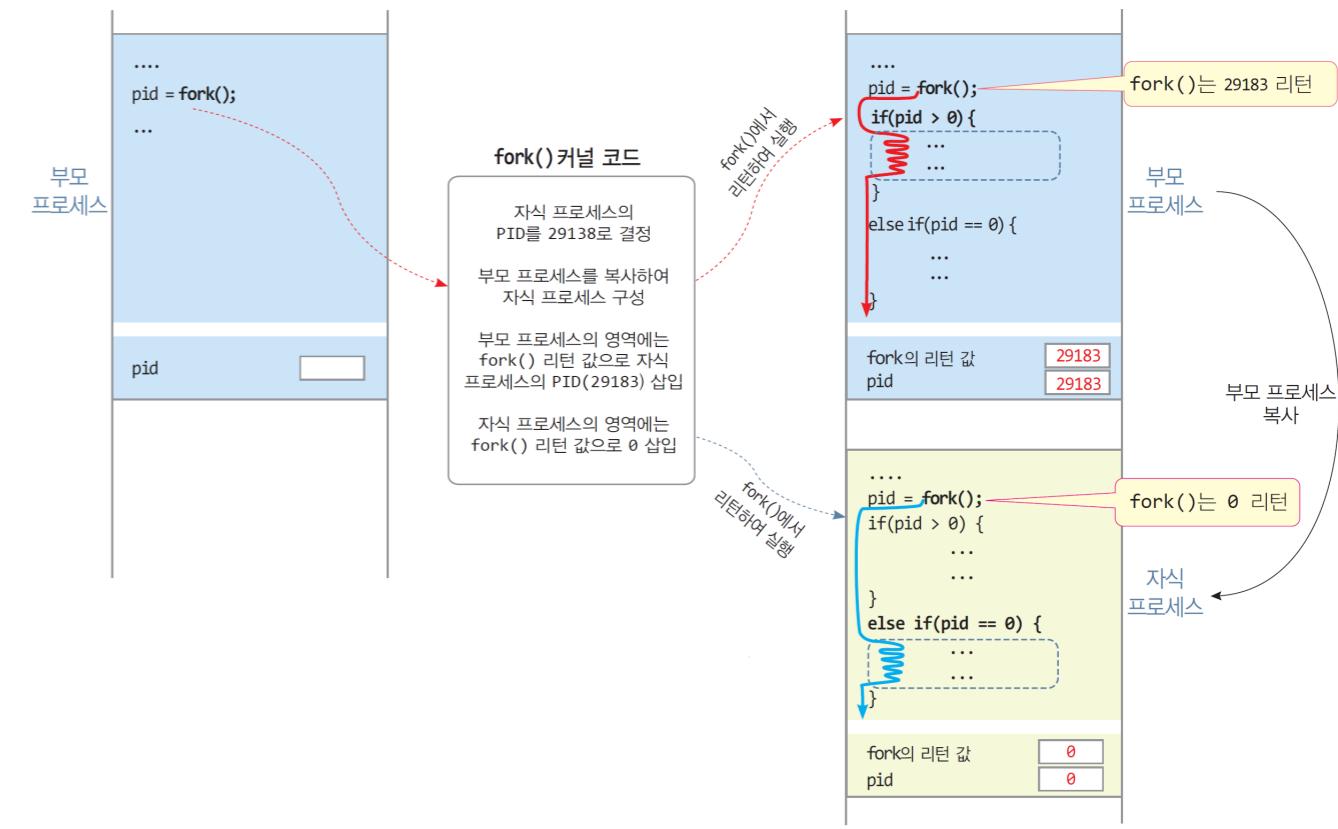


그림 3-19 fork() 시스템 호출 코드의 예

KIONNAM
KIONNAM
UNIVERSITY

 CHONNAM
NATIONAL UNIVERSITY

실행 예제

복불가능! (xNIX계열)

```
----- Run "forkex.c" -----
Parent: fork()'s return == Child pid = 5270
Parent: pid = 5269
-Child: fork()'s return pid = 0
-Child: pid = 5270, parent's pid = 5269
-Child: sum = 5050
Parent has been finished
-----
```

▶ (윈도우는 나중에!)

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 int main() {
7     pid_t pid;
8     int i, sum=0;
9
10    pid = fork(); // fork! -> create a child process
11    if(pid > 0) { // Run by the parent
12        printf("Parent: fork()'s return == Child pid = %d\n", pid);
13        printf("Parent: pid = %d\n", getpid());
14
15        wait(NULL); // Wait for the child
16        printf("Parent has been finished\n");
17        return 0;
18    }
19    else if(pid == 0) { // Run by the child
20        printf("\t-Child: fork()'s return pid = %d\n", pid);
21        printf("\t-Child: pid = %d, parent's pid = %d\n", getpid(), getppid());
22
23        for (i=1; i<=100; i++) sum += i;
24
25        printf("\t-Child: sum = %d\n", sum);
26        return 0;
27    }
28    else { // Error
29        printf("fork error");
30        return 0;
31    }
32 }
```

fork()의 장점과 단점

▶ fork() 시스템 호출의 장점

- 프로세스의 생성 속도가 빠름
- 추가 작업 없이 자원을 상속할 수 있음
- 시스템 관리를 효율적으로 할 수 있음 (프로세스 계층 구조)

▶ 단점

- 매번 모든 Context의 복사본을 만드는 것은 매우 비효율적
- 특히, 맨 처음 만든 프로그램 프로세스(0) 이외에는 다른 프로그램을 동작 할 수 없음
→ 말이 안 된다! 계속 복사되기 때문에...
- 그래서 UNIX OS 는 fork()를 한 다음 exec() 라는 시스템 콜을 호출함

프로세스 오버레이(process overlay): exec() 시스템 콜

- 기존의 프로세스를 새로운 프로세스로 전환(재사용)하는 함수
 - 현재 실행중인 프로세스의 주소 공간에 새로운 응용프로그램을 적재하고 실행
 - fork()**: 새로운 프로세스를 복사하는 시스템 호출
 - exec()**: 프로세스는 그대로 둔 채 내용만 바꾸는 시스템 호출
 - execvp(), execv(), exec(), execle()
 - 실행 파일을 로딩하여 현재 프로세스의 이미지 위에 단순히 덮어씀

주의! 프로세스를 새로 생성하는 것이 아님

- 프로세스의 PID 변경 없음
- 프로세스의 메모리 공간(코드, 데이터, 힙, 스택)에 새로운 프로그램이 적재
- 보통 fork()를 통해 생성된 자식 프로세스가 exec() 실행
- loader가 exec를 통해 호출됨

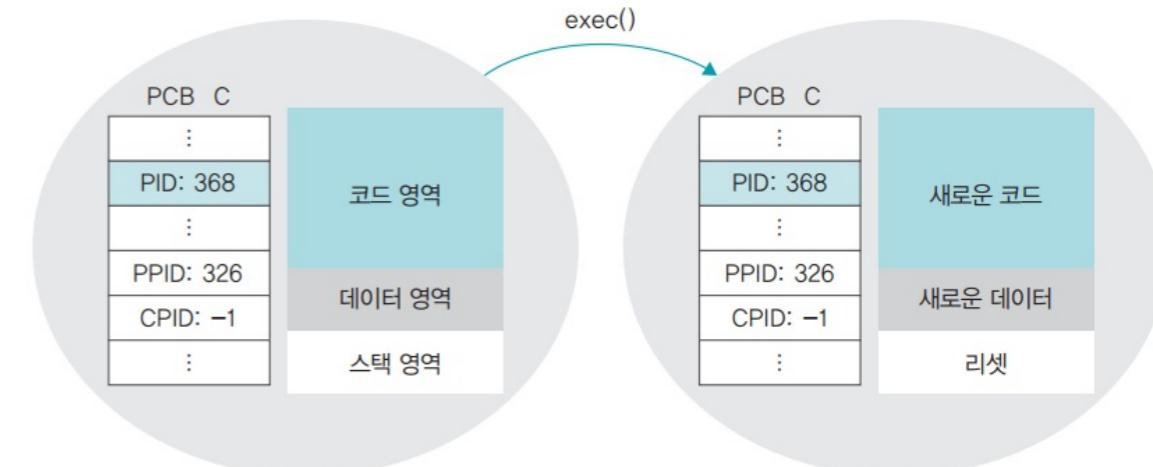


그림 3-21 exec() 시스템 호출 후 프로세스의 변화

exec() 실행 과정

▶ 메모리

- Code 영역에 있는 기존의 내용을 지우고 새로운 코드로 바꿔버림
- Data 영역이 새로운 변수로 채워지고 힙/스택 영역이 리셋

▶ PCB

- PID, PPID, CPID, 메모리 관련은 유지
 - 새로운 프로세스가 전환 되더라도 종료 후 부모 프로세스로 돌아올 수 있음
- Program counter 및 기타 register, 파일 정보 등이 모두 리셋

```

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

void main()
{   int pid;

    pid=fork(); ←

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

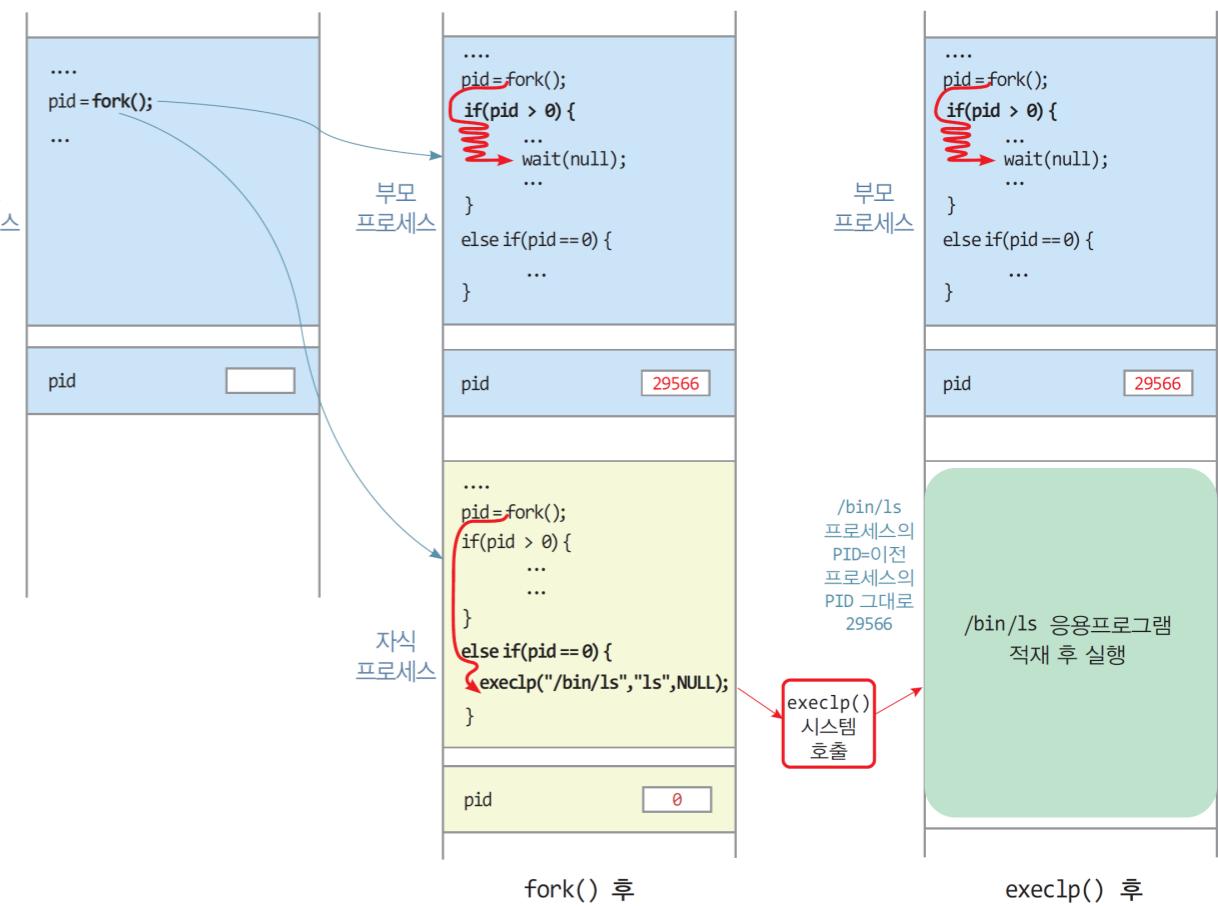
    else if(pid==0) { /* child process */
                      execlp("mplayer", "mplayer", NULL);
                      exit(0); }

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

```

부모 프로세스

그림 3-22 exec() 시스템 호출 코드의 예



자식이 일하는 동안 부모는 뭘 하는가? 기다려야지...: wait() 시스템 콜

- ▶ 자식 프로세스가 끝나기를 기다렸다가, 자식 프로세스가 종료되면 이어서 실행을 계속하는 시스템 콜
- ▶ e.g., Linux shell에서,
 - fore ground process가 있으면 shell이 wait 상태
 - 프로세스가 실행중인 상태에서 "ctrl+z → 프로세스 일시중단"
→ 쉘이 wait에서 빠져나옴

복불가능! (xNIX계열)

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 int main() {
7     pid_t pid;
8     int status;
9
10    pid = fork();
11
12    if (pid > 0) {
13        printf("Parent: Waiting for the child\n");
14        wait(&status); // Wait.
15        printf("Parent: child's exit code=%d\n", WEXITSTATUS(status));
16        return 0;
17    }
18    else if (pid == 0) {
19        execlp("/bin/ls", "ls", NULL); // run ls on the child
20    }
21    else {
22        printf("fork error!");
23        return 0;
24    }
25 }
```

그럼 자식이 끝난 것은 어떻게 아는가?

exit() 시스템 콜

- ▶ 작업의 종료를 알리는 시스템 콜

→ 종료를 명시적으로 알림으로서 부모는 자식이 사용하던 자원을 빨리 회수!

- ▶ 종료 코드

- 부모 프로세스에게 상태나 종료의 이유를 전달하는 값, e.g., `exit(1)`

- 보통 정상종료는 0, 나머지는 1-255 범위 내에서 임의로 사용!

- 여태 코드를 짜면서 `exit(nnn)`을 써본적이 한번도 없다? No!

- `main()` 함수의 리턴 값 (e.g., `return 0;`) == `exit(0);`

- 내부적으로 `exit()` 시스템 콜이 실행되도록 컴파일 됨

- OS가 자동적으로 `exit()`를 호출해주도록 하여 프로그램을 종료 시키도록 하게 함

- 이걸 부모가 확인해야지 최종적으로 자식 프로세스가 종료 됨.

프로세스가 종료 되면....:

exit() 시스템 콜을 통한 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

- 아직 PCB가 프로세스 테이블에서 제거된 것은 아님. 후술할 부모가 종료를 확인해야함

▶ (3) 자식 프로세스들을 init 프로세스에게 입양

▶ (4) 부모 프로세스에게 SIGCHLD 신호 전송 (일종의 종료 알림 신호)

- **부모의 의무:** SIGCHLD를 수신하고 wait() 시스템 호출로 자식의 종료 코드 읽기 실행
죽은 자식이 남긴 정보를 확인 후, 자식 프로세스의 PCB가 완전히 제거됨.
- **만약에 부모가 자식의 종료 신호를 제때 확인하지 못하면? 자식은 좀비 프로세스가 됨!**
 - PCB가 남아있으므로 ps 명령어 등으로 존재를 확인할 수 있다.

좀비 프로세스 확인

- ▶ 간단한 예제
 - xNIX에서 돌려보세요!

- ▶ 이미 메모리 정리는 다 됐으므로
점유하는 메모리 size는 0!
 - 좀비는 단순히 PCB정리가
안된것 뿐!

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int main(){
8     pid_t pid, zompid;
9     int status;
10
11    pid = fork();
12    if( pid > 0 ) { // parent's code
13        sleep(10); // sleep for 10 sec
14        zompid = wait(&status); // wait for the child
15        printf("Parent: child %d has been finished with %d\n",
16               zompid, WEXITSTATUS(status));
17        return 0;
18    }
19    else if(pid == 0){ // child's code
20        printf("Child: I am done %d \n", getpid());
21        exit(100); // exit code 100
22    }
23    else{
24        printf("fork error\n");
25        return 0;
26    }
27 }
```

```

taejune@TaejunesMBP2013 Workspace % gcc zombie.c; ./a.out &
[1] 58137
taejune@TaejunesMBP2013 Workspace % 자식 프로세스 : 58138 종료합니다.
```

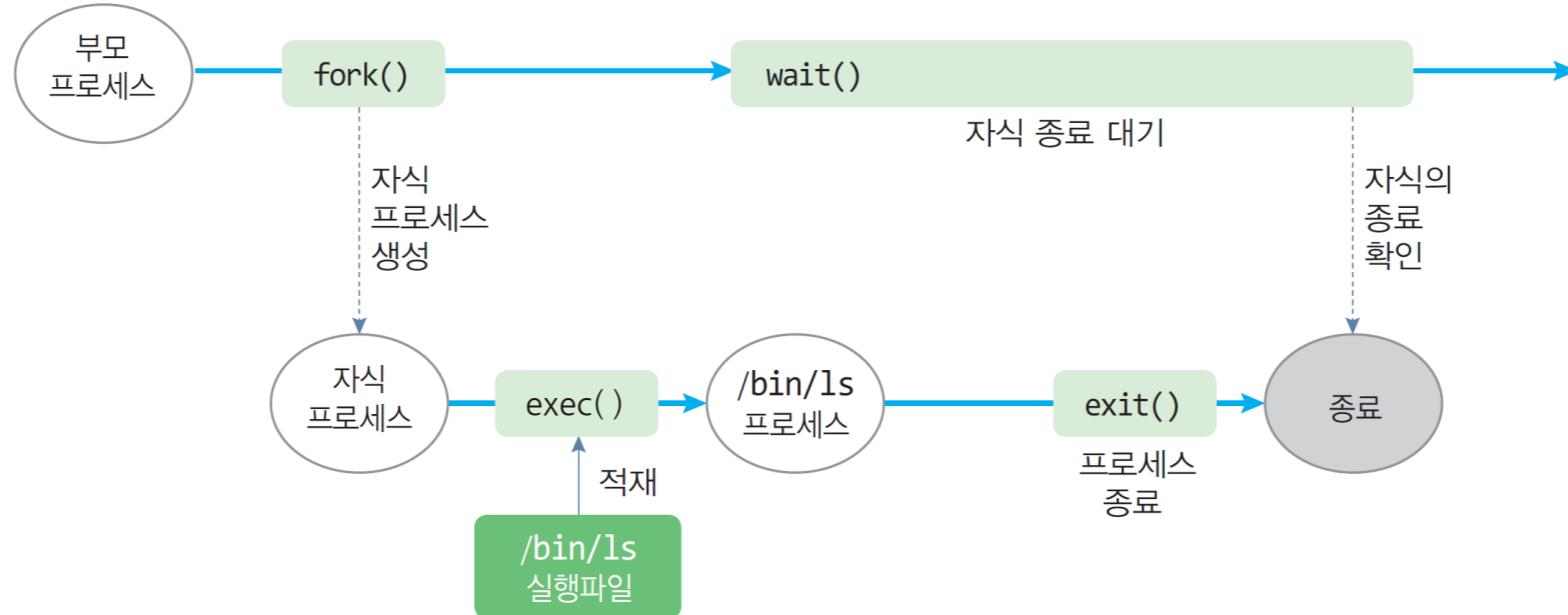
```

taejune@TaejunesMBP2013 Workspace % ps -l
  UID  PID  PPID      F CPU PRI NI      SZ    RSS WCHAN   S          ADDR TTY      TIME CMD
  501 39993 39991    4006  0 31  0 4297420  1780 -      S          0 ttys000  0:00.16 -zsh
  501 58137 39993    4006  0 31  5 4270244   704 -      SN         0 ttys000  0:00.00 ./a.out
  501 58138 58137   2006  0  0  5      0     0 -      ZN         0 ttys000  0:00.00 (a.out)
  501 54548 54547    4006  0 31  0 4297200   980 -      S+         0 ttys001  0:00.02 -zsh
taejune@TaejunesMBP2013 Workspace % 부모프로세스 : 자식 PID=58138, 종료 코드 =100
```

```

[1] + done      ./a.out
taejune@TaejunesMBP2013 Workspace %
```

정리해서, UNIX OS에서의 프로세스 생성(및 종료) 과정



▶ fork() → exec()의 구조

- fork()를 통해 프로세스를 만들고, exec()를 통해 필요한 프로세스를 실행함
 - 이때 생성을 한 프로세스는 부모 프로세스, 생성 된 프로세스는 자식 프로세스
 - 부모는 wait()를 통해 기다리고, 자식은 exit()를 통해 자신의 종료를 알림
- ▶ 모든 프로세스는 최초의 조상이 있다! 그것이 1번 프로세스, **init!**

그냥 프로세스 만들면 안되는가? 왜 이렇게 하는가?

- ▶ 프로세스 생성 과정이 간소화 됨
- ▶ 프로세스 관리가 쉬워짐 → 부모를 통해 자식을 관리 할 수 있음 (계층 구조)
 - abort() 호출 : 부모가 신호를 보내서 자식을 죽이는 시스템 콜
- ▶ 그리고 프로세스 간 통신 (IPC)
 - 프로세스는 독자적인 메모리 공간(가상 주소 공간)을 가진 것 처럼 운영됨.
 - 서로 간섭이 불가! 서로 통신도 불가!: 서로 통신이 필요하다면? → e.g., SIGCHLD

- 파일을 이용해 서로 의사소통

여기서 fork() 가 사용 됨!

- 같은 파일 핸들러를 두 프로세스가 공유할 수 있음
- fork가 아니면?

같은 파일을 두 프로세스에서 동시에 못열죠!

```
int fd; // 파일 시스템 변수

foo() {
    fd = open("pile"); // 파일을 연다.
    if(fork() == 0) { // 자식일 경우
        read(fd,...); // 하나의 파일로 서로 의사소통함
    }
    else { // 부모일 경우
        write(fd,...); // 하나의 파일로 서로 의사소통함
    }
}
```

윈도우의 경우...

- ▶ CreateProcess function ≈ fork+exec

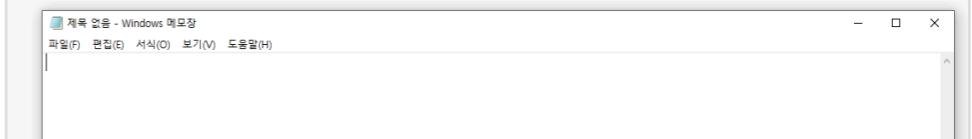
- ▶ fork와의 차이

- fork: 자식 프로세스가 부모 프로세스의 주소 공간을 상속받음
- CreateProcess: 자식 프로세스에게 구체적으로 어떤 프로그램을 실행할 것인지 요구.
(주소값이 명확해야함)

복불가능! (윈도우)

```
1 #include <stdio.h>
2 #include <windows.h>
3
4 int main(){
5     STARTUPINFO si = {0};
6     PROCESS_INFORMATION pi = {0};
7
8     si.cb = sizeof(si);
9
10    if(!CreateProcessA(NULL,
11                      "C:\\\\windows\\\\system32\\\\notepad.exe",
12                      NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)){
13        printf("Error!\n");
14        return -1;
15    }
16
17    WaitForSingleObject(pi.hProcess, INFINITE);
18    printf("Child has been finished\n");
19
20    CloseHandle(pi.hProcess);
21    CloseHandle(pi.hThread);
22
23    return 0;
24 }
```

C:\Users\taejune\Desktop\testc\testc>createprocess.exe



C:\Users\taejune\Desktop\testc\testc>createprocess.exe
Child has been finished

C:\Users\taejune\Desktop\testc\testc>

프로세스 계층 구조

부모가 자식을 만들고, 자식은 또 자식을 만들고...

모든 프로세스의 조상 init

- 유닉스의 모든 프로세스는 init 프로세스의 자식이 되어 트리 구조를 이룸
 - 윈도우는 이정도 까진 아니고...! 어쨌든 비슷함

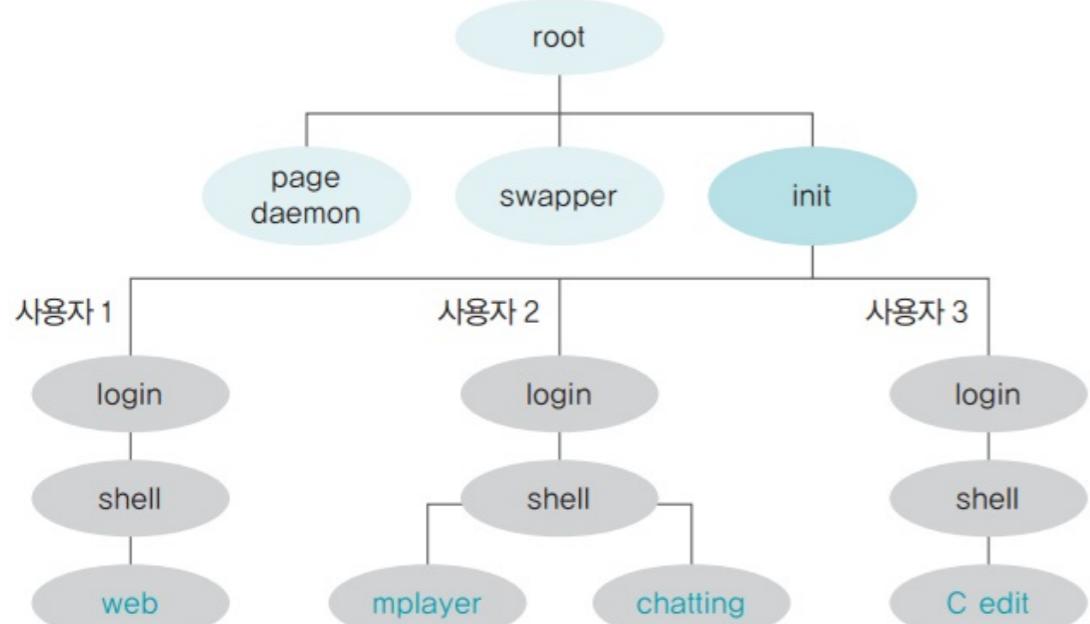
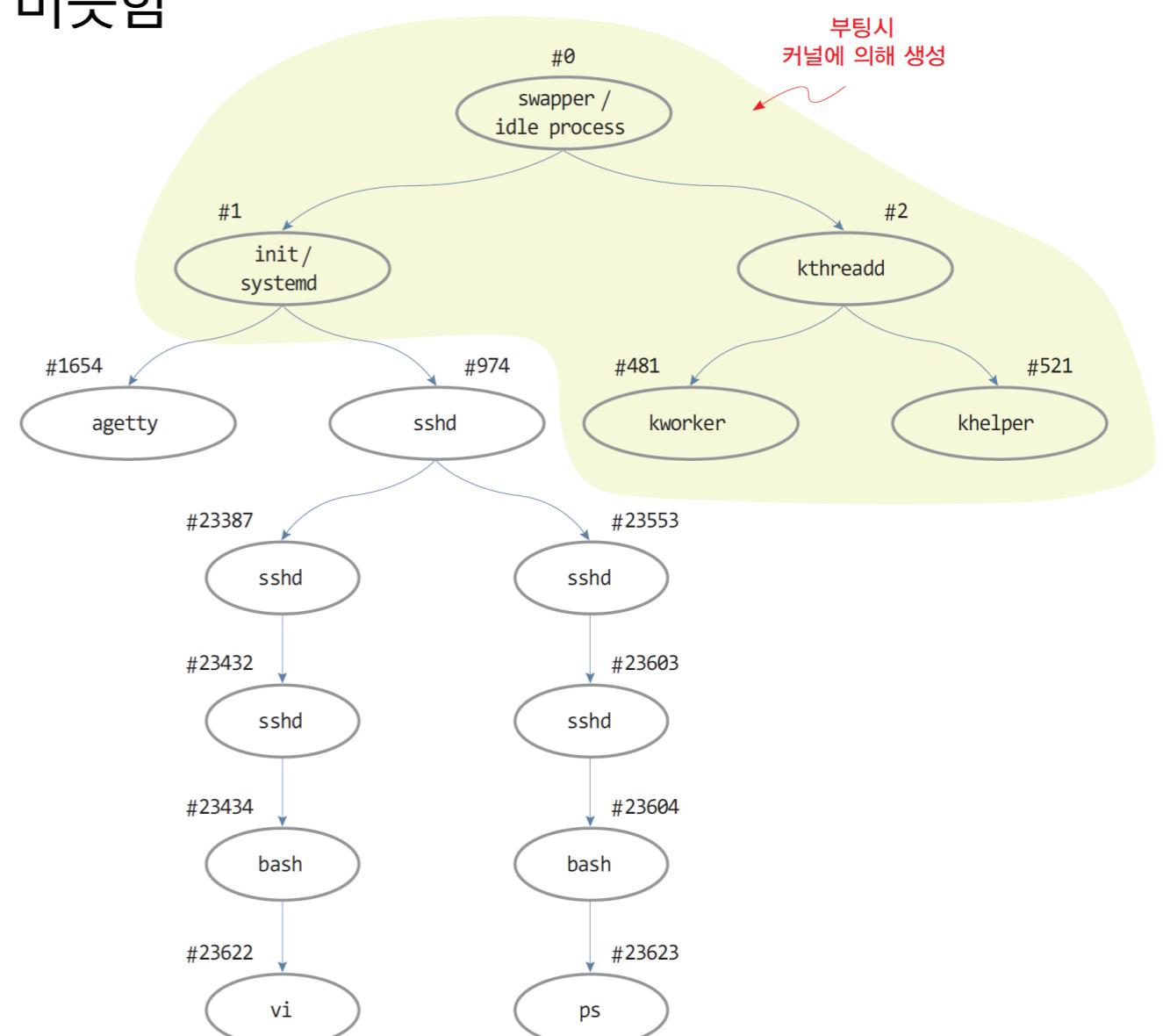


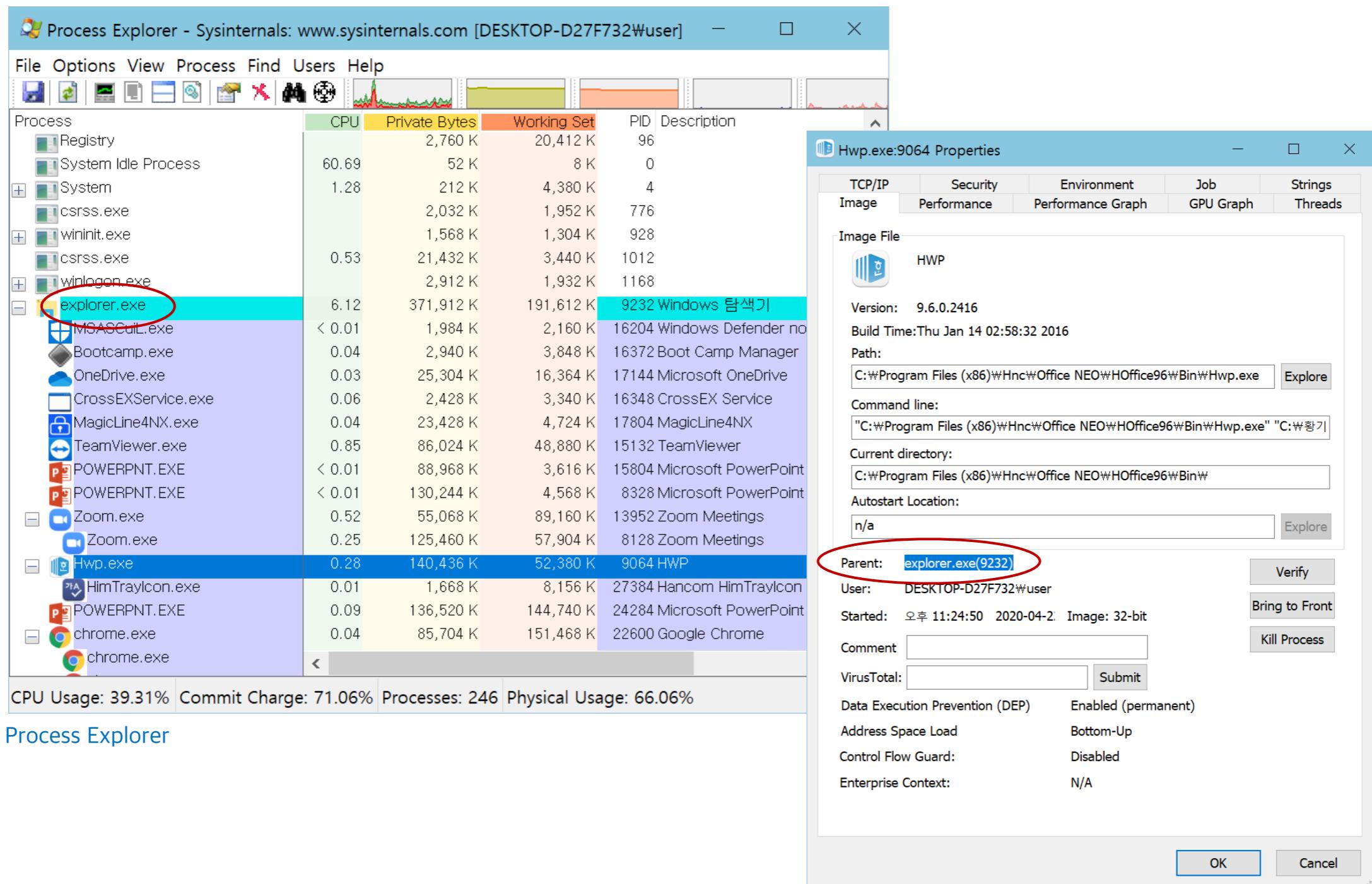
그림 3-23 유닉스의 프로세스 계층 구조



e.g., 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 - ttym1 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
$
```

e.g., Windows에서의 계층 구조



계층 구조

- ▶ **프로세스는 일반적으로 부모-자식 관계**
 - #0 프로세스가 시스템 부팅시 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()
- ▶ **PID 0, 1, 2 등의 몇몇 조상 프로세스는 시스템 콜이 아닌
부팅 시 OS차원에서 수작업(hand-craft)으로 생성됨!**

왜 계층 'fork-exec' 구조인가?

- ▶ 여러작업을 처리하기에 용이
- ▶ 프로세스의 재사용이 용이

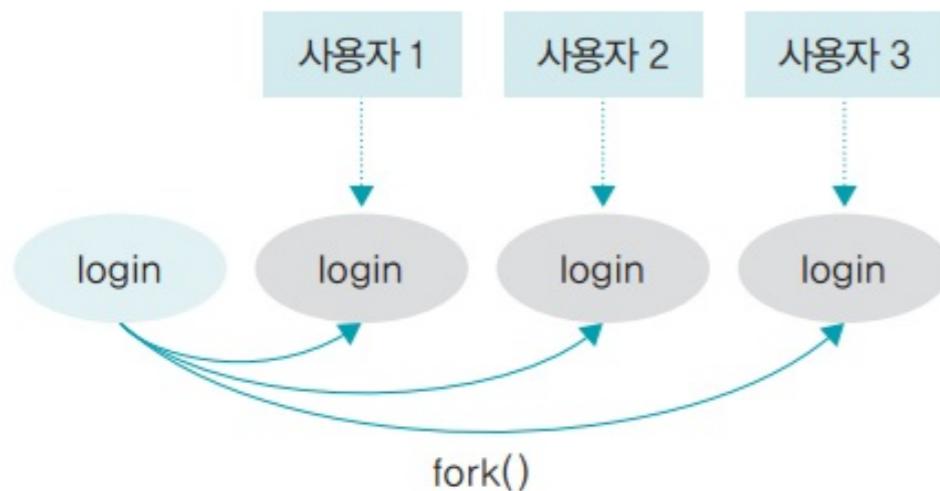


그림 3-24 여러 사용자를 동시 처리

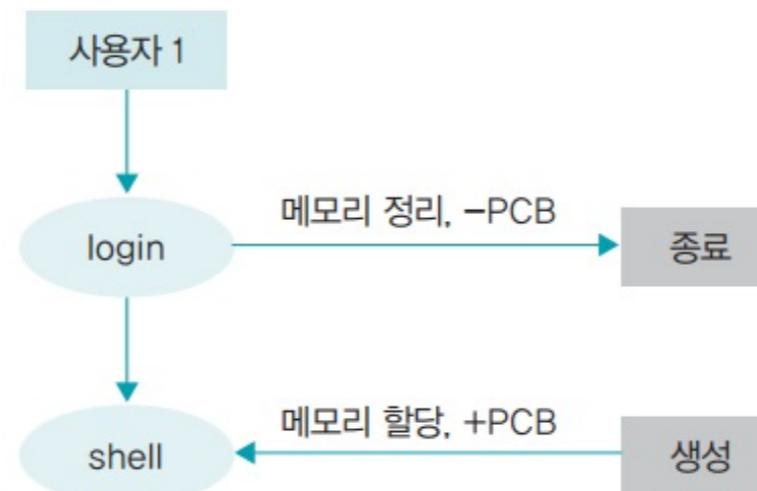
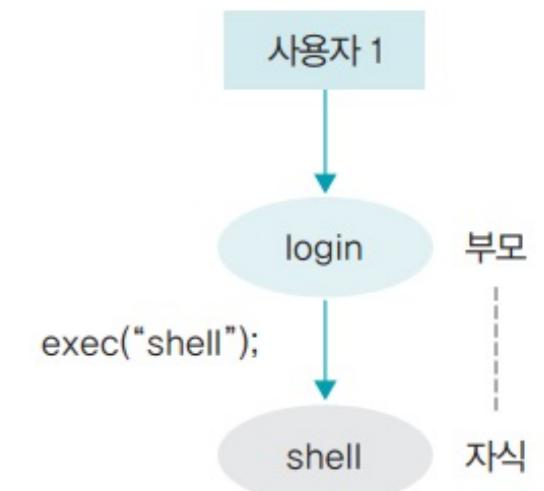


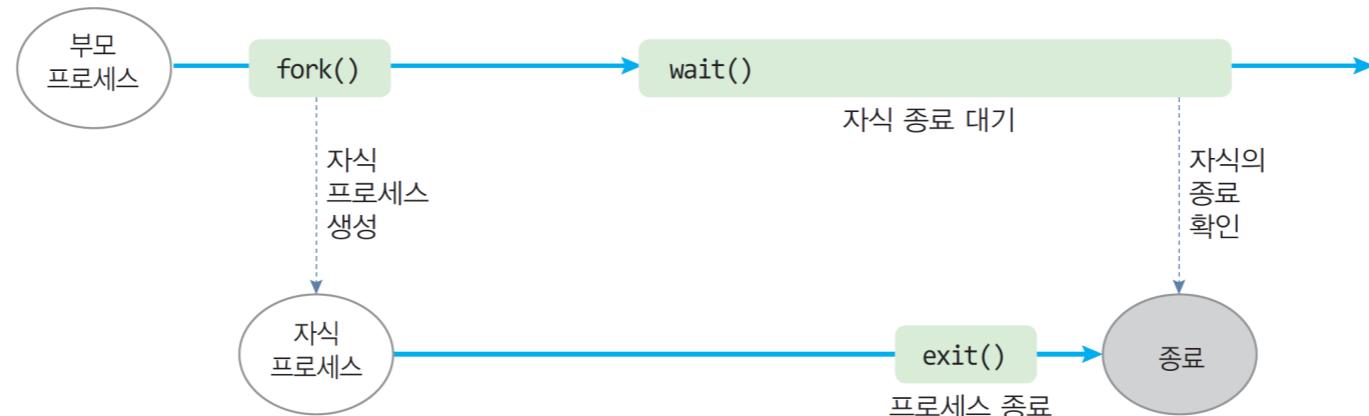
그림 3-25 프로세스의 재사용



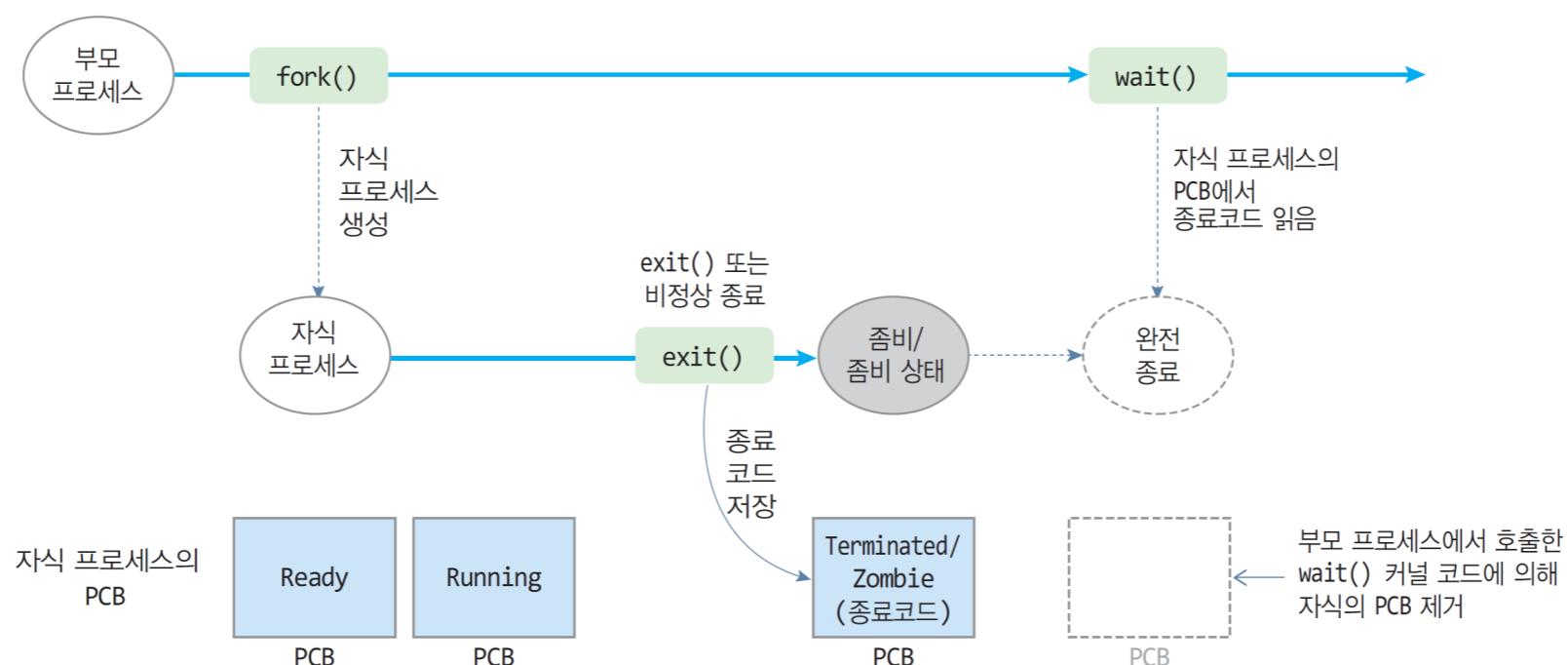
- ▶ 관리가 용이
 - 프로세스 간 책임 관계가 분명해짐 → 자원 회수등이 쉬움

부모-자식간 실행 관계 (1)

- 부모가 자식을 생성한 후 자식의 종료를 기다리는 경우 (평범...)



- 부모가 자식의 종료를 (제때) 못받아주면 → **자식 프로세스는 좀비 상태가 됨**

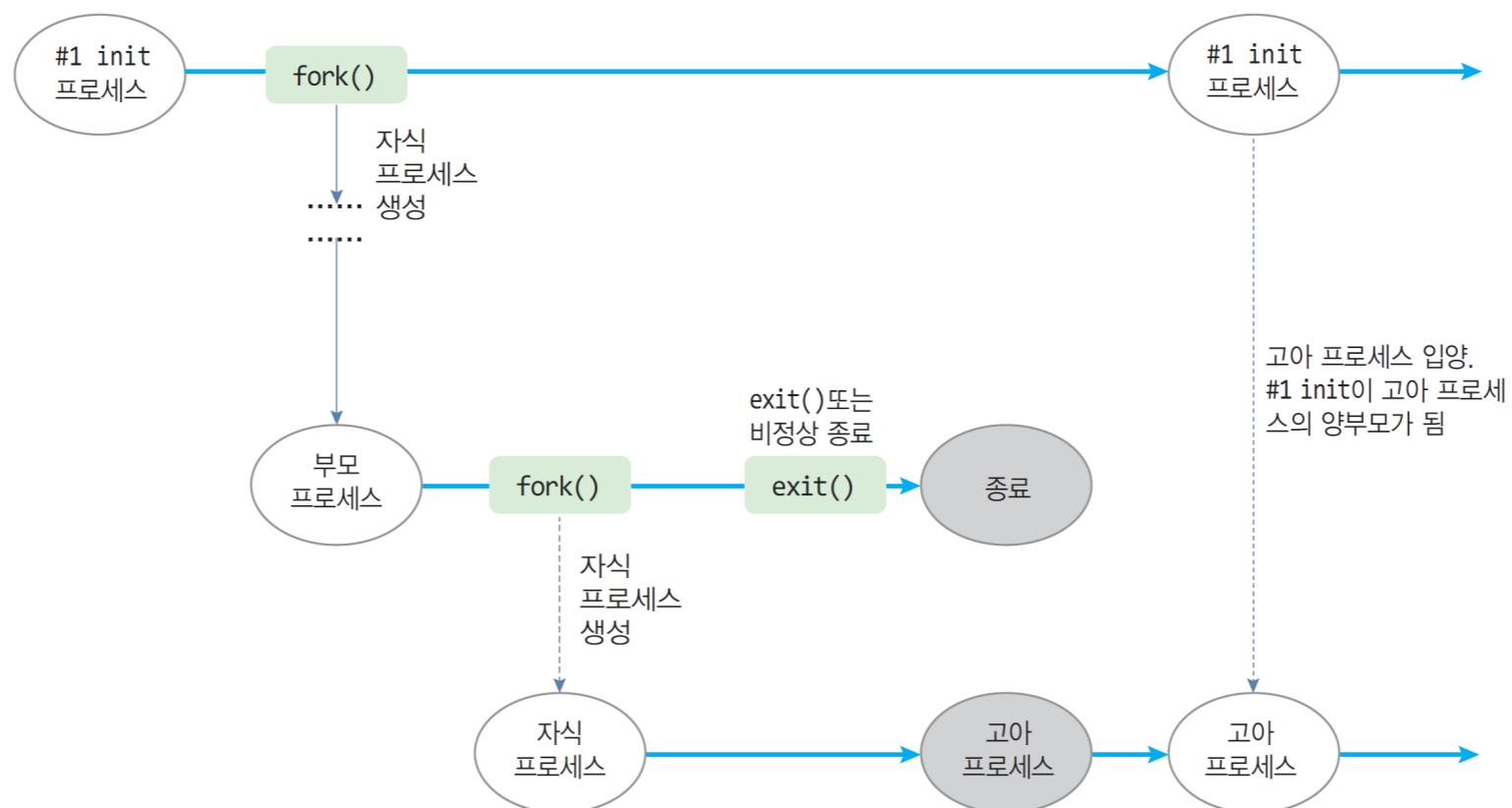


좀비 프로세스의 제거

- ▶ 좀비 프로세스가 메모리를 차지 하지 않는다 하지만... PCB의 낭비는 발생!
 - 커널 입장에서 봤을때는 PCB를 유지하기 위한 얼마간의 자원을 소모
 - 무엇보다 커널이 유지할수 있는 PCB 테이블의 크기에 제한이 있음
 - 많은 좀비프로세스가 발생할경우 시스템 성능에 영향! (e.g., 프로세스 스케줄링을 위한 확인 등)
 - 시스템을 모니터링 할때 기분이 나빠지는 심리적효과 (정신적인 데미지!)
- ▶ 좀비 프로세스 제거
 - 부모 프로세스에게 SIGCHLD 신호를 보내기 → 부모 프로세스에서 wait() 호출하여 처리
 - 부모에게 SIGCHLD 핸들러가 없다면 좀비는 제거되지 못함...!
 - 부모 프로세스를 강제 종료 → 좀비를 고아(Orphan)화
 - 좀비는 init 프로세스의 자식이 되고 → init이 wait() 호출하여 좀비 프로세스 제거

부모-자식간 실행 관계 (2)

- ▶ 고아 프로세스(Orphan Process): 부모가 먼저 종료한 자식 프로세스
- ▶ 부모 프로세스가 종료할 때 일반적으로
 - 커널(exit() 시스템 호출 코드)은 자식 프로세스가 있는지 확인
 - 자식이 있으면, 자식 프로세스(고아)를 init 프로세스에게 입양 → PPID가 1로 변경
 - (운영체제에 따라 모든 자식 프로세스 강제 종료시키기도 함)



부모/자식 프로세스 ID확인

▶ 간단한 예제

복불가능! (xNIX계열)

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main() {
6     pid_t pid, ppid;
7
8     pid = getpid(); // get a PID
9     ppid = getppid(); // get a Parent's PID
10
11    printf("Process ID = %d, Parent's ID = %d\n", pid, ppid);
}
```

```
----- Run "process_info.c" -----
Process ID = 3807, Parent's ID = 3801
-----
Press ENTER or type command to continue
```

복불가능! (Windows)

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <tlhelp32.h>
4
5 int main()
6 {
7     int pid = GetCurrentProcessId();
8
9     HANDLE h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
10    PROCESSENTRY32 pe = { 0 };
11    pe.dwSize = sizeof(PROCESSENTRY32);
12
13    if (Process32First(h, &pe)) {
14        do{
15            if (pe.th32ProcessID == pid) {
16                printf("Process ID = %d, Parent's ID = %d\n",
17                      pid, pe.th32ParentProcessID);
18            }
19        } while (Process32Next(h, &pe));
20    }
21    CloseHandle(h);
22 }
```

```
C:\Users\taejune\Desktop\testc\testc>test.exe
Process ID = 7576, Parent's ID = 6488

C:\Users\taejune\Desktop\testc\testc>test.exe
Process ID = 5708, Parent's ID = 6488

C:\Users\taejune\Desktop\testc\testc>test.exe
Process ID = 1576, Parent's ID = 6488
```

다른 프로세스의 종류 (1)

- ▶ 백그라운드 프로세스와 포그라운드 프로세스
 - **백그라운드 프로세스 (Background process)**
 - 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 (Foreground process)**
 - 실행되는 동안 터미널 사용자의 입력을 독점하는 프로세스

다른 프로세스의 종류 (2)

- ▶ 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
 - 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

끝으로...

- ▶ 모든 프로세스의 조상 init!

그 위/옆에 있는 프로세스는 누구?

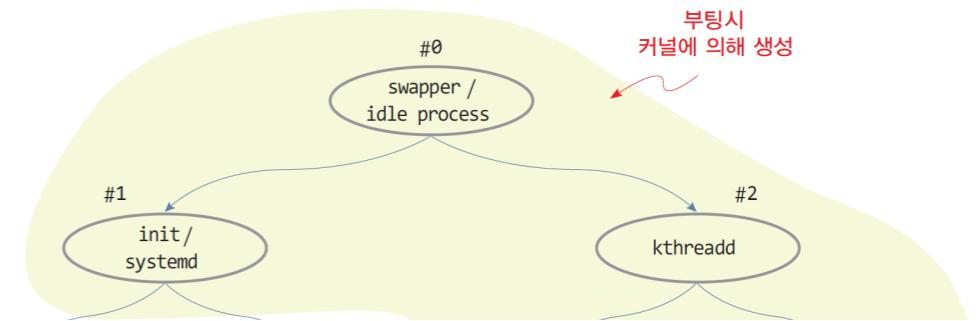
- ▶ 0번 프로세스

- swapper (UNIX): 부팅 담당 및 #1 init을 생성
- idle (LINUX), system idle process (Windows)

- 우선 순위가 가장 낮은 프로세스. 아무것도 안함.
 - 실행중인 프로세스가 1개도 없는 상태에 빠지지 않기 위해 만든 프로세스

- ▶ 2번 프로세스; kthreadd

- 커널 프로세스는 커널 공간에서만 실행하는 프로세스를 의미
- 대부분 커널 스레드 형태로 구동 → kthreadd 모든 커널 프로세스(thread)의 조상



Summary

▶ 프로세스 Process

- 실행파일(Program)이 메모리에 올라가 작동되는 상태!
- OS는 Process Control Block (PCB)를 통해 프로세스를 관리
 - 프로세스의 생애 주기: New → ready → running → (blocked) → terminated

▶ 프로세스 메모리 구조: 메모리에 그냥 올리는게 아니다!

- 프로세스 가상 메모리: 각 프로세스는 마치 메모리를 혼자 독점하는 것처럼 보인다!
- Code, data, heap, stack 영역

▶ 프로세스 생성과 복사: 프로세스가 매번 맨땅에 만들어지는건 아니에요!

- fork() → exec()
- wait(), exit()

▶ 프로세스 계층 구조: 부모와 자식 관계 → 조상님이 계십니다!

- 좀비 프로세스: 부모가 자식의 종료를 확인하지 않아 → 작업 종료 후 PCB가 남은 상태
- 고아 프로세스: 부모가 먼저 죽은 상태 → init으로 입양!