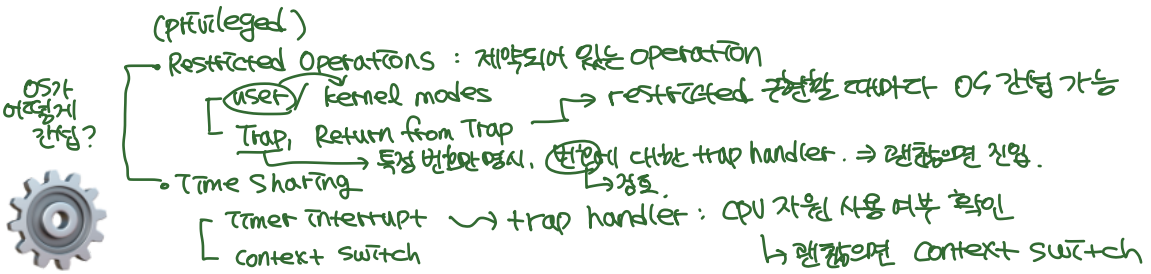
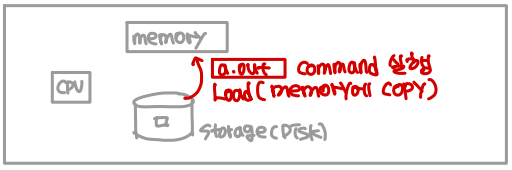


03. Limited Direct Execution

▼ Direct Execution



running program에 대한 제한 없이 OS가 아무것도 control 못 함 ⇒ OS 관여 없는 version!

*간섭 없는 환경 example ⇒ 실행 파일이 실행되었을 때 or terminal로 실행되었을 때 상황

OS	Program
① Create entry for process list → PCB 생성	
② Allocate memory for program → memory 할당	a.out in Linux → Disk에 저장됨
③ Load program into memory → program load	
④ Set up stack with argc/argv → 파라미터 : stack을 통해 전달	
⑤ Clear registers → 레지스터 초기화	
⑥ Execute call main() → main 함수 호출	
	⑦ Run main()
	⑧ Execute return from main()
⑨ Free memory of process → 자원 회수	
⑩ Remove from process list → PCB 제거	

(PCB (process 정보저장하는 자료구조))

함수를 위한 지역변수, 저장소

⇒ OS가 잔소리 많음...

1. OS가 program들이 효율적으로 실행되도록 어떻게? *(restricted operation)* → 자원을 효율적으로 분배하기 위해 마음대로 실행하면 안 되는 연산 관리 및 제한
2. OS가 running을 중지하거나 time sharing은 어떻게? *(time sharing)* → process switch

⇒ 관여하는 mechanism이 필요 (execute call ~ execute return 사이에!)

⇒ 실행 도중에 문제 없는지 계속 check(효율적으로!) ⇒ 간섭 필요 ★

▼ Limited Direct Execution

▼ problem #1 : Restricted operations(privileged operations)

꼭 실행되어야 하는 operation이지만 os의 control 내에서만 실행되어야 함

---

ex) issuing an I/O request to a disk ⇒ cpu나 memory 같은 더 많은 시스템 자원 접근

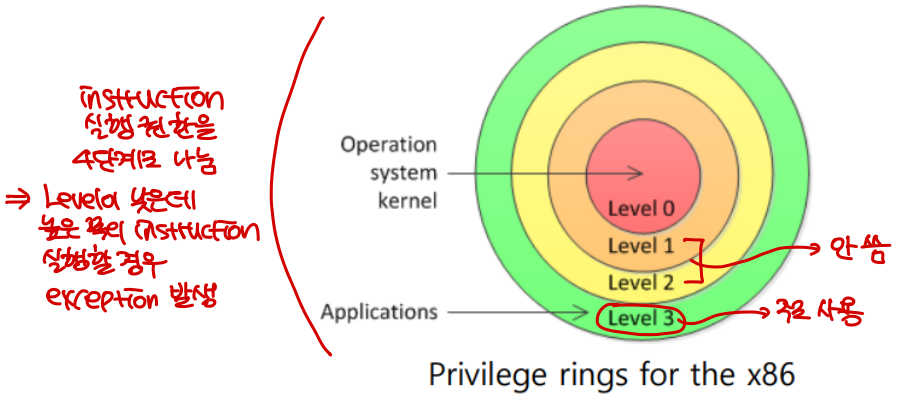
→ 항상 부족하기에 응용 sw가 최대한 공평하게 사용하도록 관리함

+) application : restricted operation 무조건 수행은 가능해야 함. 단, 안전하게 OS의 control 내에서!

▼ processor modes

restriced operation 효과적으로 사용하기 위해서 cpu가 processor mode를 구분

(HW가 지원!)



Privilege rings for the x86

- **user mode**
  - 우리가 일반적으로 사용하는 program level
  - restricted operation 사용 불가 → exception 발생
- **kernel mode**
  - OS runs in kernel mode
  - restricted operation 사용 가능

▼ System calls

user process(mode) : 어떻게 privileged operations 사용?

⇒ system call이 process의 mode를 변경

- **trap instructions** : processor mode up → privileged operation 가능
  - user mode → kernel mode
- **return-from-trap instructions** : system call 끝나면 사용 (trap 끝내고 원래대로 돌아올 때)
  - return into the calling user program
  - kernel mode → user mode

⇒ 어떻게 OS 내부로 진입함?

→ 임의의 kernel 영역 주소 공간으로는 점프 불가 → 꽤 제한적

→ 정해진 영역만 호출 가능

⇒ trap table : trap handler들의 위치를 가지고 있는 table

- **trap table**
  1. *booting* → OS : trap handler의 주소를 CPU에게 알려줌 → trap table의 번호에 해당하는 trap handler 호출
     - trap table은 하나만 존재 (OS가 관리)
  2. system call number가 각 system call에 할당됨.

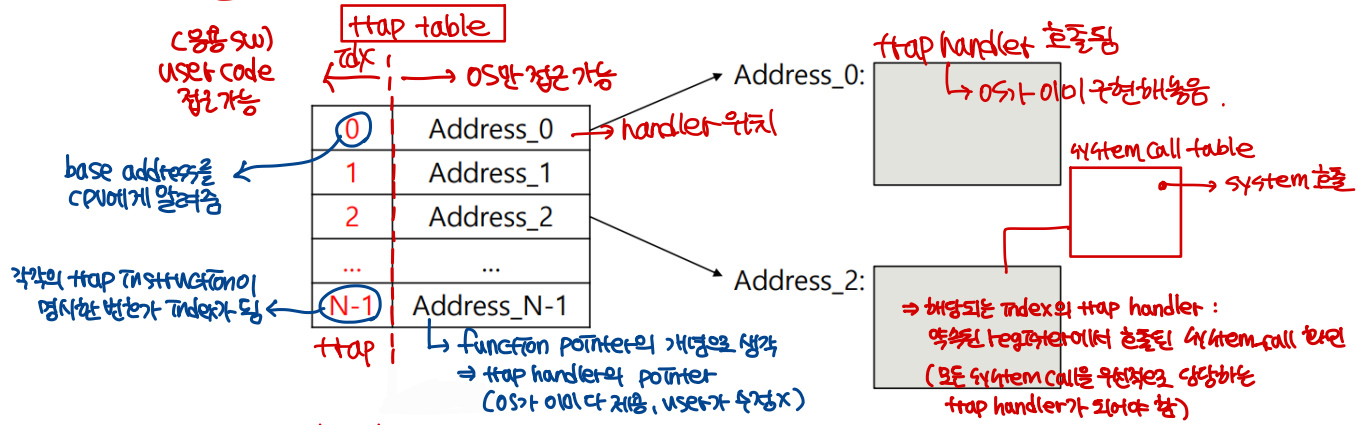
(trap handler들이 OS에 이미 포함되어 있음.)

- OS의 특정 address 제공 X ⇒ restricted operation을 위한 trap handler 호출
  - trap handler : register 위치 확인 및 저장.
- 해당 호출에 대해 합법적인지 OS가 검사

3. user code : 특정 주소에 jump는 불가 **but** index 번호로 특정 호출 가능

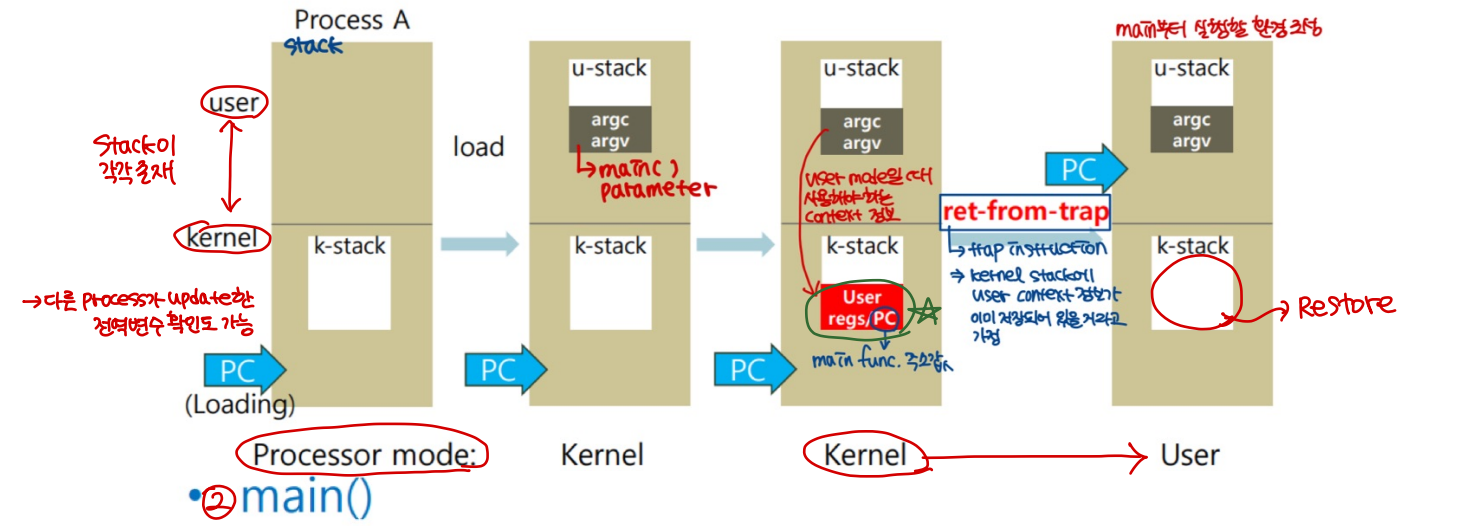
⇒ trap instruction만 restricted operation이 가능하게 됨!

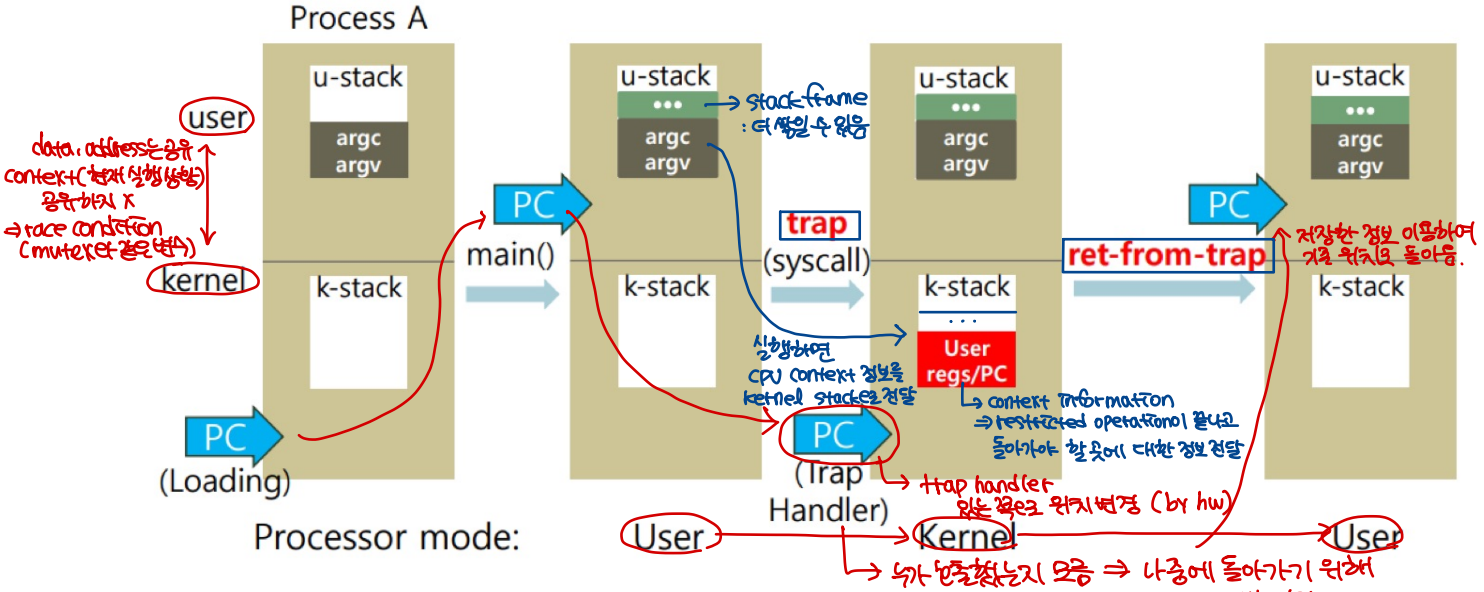
⇒ OS : trap table(table에 정의 해놓은 gate 관리 포함)만 관리하면 할 일 끝!



*privileged operation

OS	Hardware (CPU)	Program
Initialize trap table	Remember address of syscall handler	
① Create entry for process list ② Allocate memory for program ③ Load program into memory ④ Set up user stack with argc/argv ⑤ Fill kernel stack with regs/PC ⑥ Return-from-trap	Restore regs from kernel stack Move to user mode Jump to main	Run main() ... Call system call Trap into OS
Handle trap Return-from-trap	Save regs/PC to kernel stack Move to kernel mode Jump to trap handler	
Free memory of process Remove from process list	Restore regs from kernel stack Move to user mode Jump to PC after trap	Return from main() Trap (via exit())





▼ Problem #2: Switching Between Processes

어떤 process에게 cpu 자원을 할당? ⇒ time sharing으로 자원 할당 여부

1. cooperative apporoach
   - wait for system calls → OS가 time sharing에 관련된 정책을 펼침
     - system call이 호출되는 시점까지 기다려서 OS가 cpu를 가져옴
     - 너무 오래 실행되는 process는 cpu를 포기함
   - wait for errors
     - application : illegal operation을 실행할 때 OS에게 cpu 맡김
       - ex. dividing by zero, segmentation fault → 해결책 : process 종료

⇒ 좋은 solution이 될 수 없음.

2. non-cooperative approach → timer 사용
   - OS takes control

timer device : interrupt가 주기적으로 발생하도록 프로그래밍 가능
→ interrupt가 증가되면 현재 실행중인 process 중지, 미리 구성된 interrupt handler 실행됨

- timer interrupt : system이 시작될 때 꼭 필요
  - OS가 cpu 사용할 기회가 없어지는 걸 방지
    → cpu 내부 timer 존재 → interrupt 일어나도록 ⇒ 응용 process system call을 언제, 얼마나?
    (sw적으로) (OS가 설정해서 timer를 주기적으로 가지도록)
- Context Switch
  Linux : timer interrupt handler 내부에 CPU scheduler 존재 ⇒ context switch func. 존재
  - saving and restoring context (응용 sw보다 os(kernel)가) + context switch가 필요한지 판단
    - 실행하고 있던 process의 register value를 저장 (onto its kernel stack)
    - kernel stack에 있던 실행될 process를 다시 저장
    - return-from-trap instruction 실행 ⇒ 저장되어 있던 정보 사용
      → system : 다른 process의 실행 재개 보장.

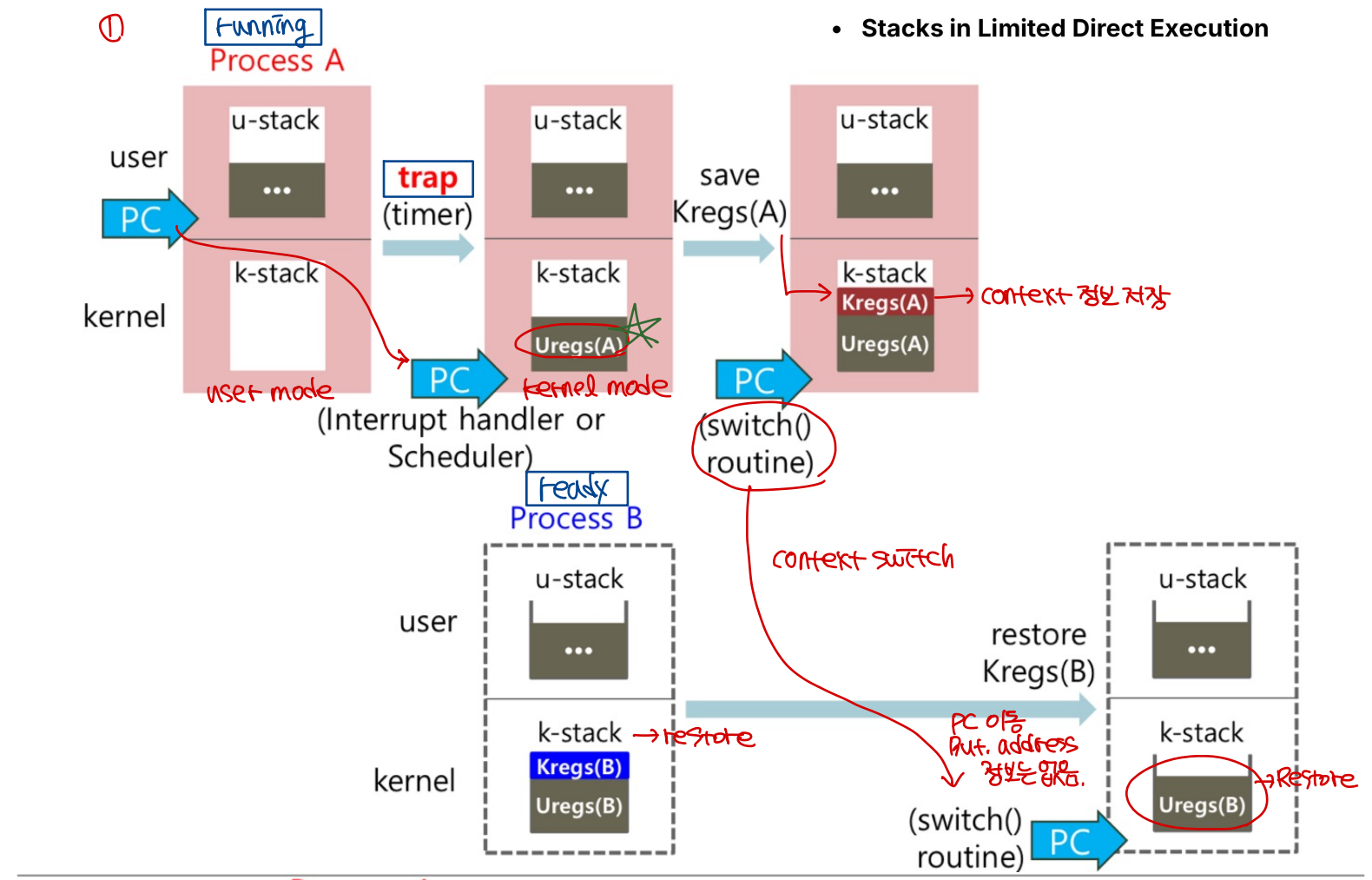
* context switch

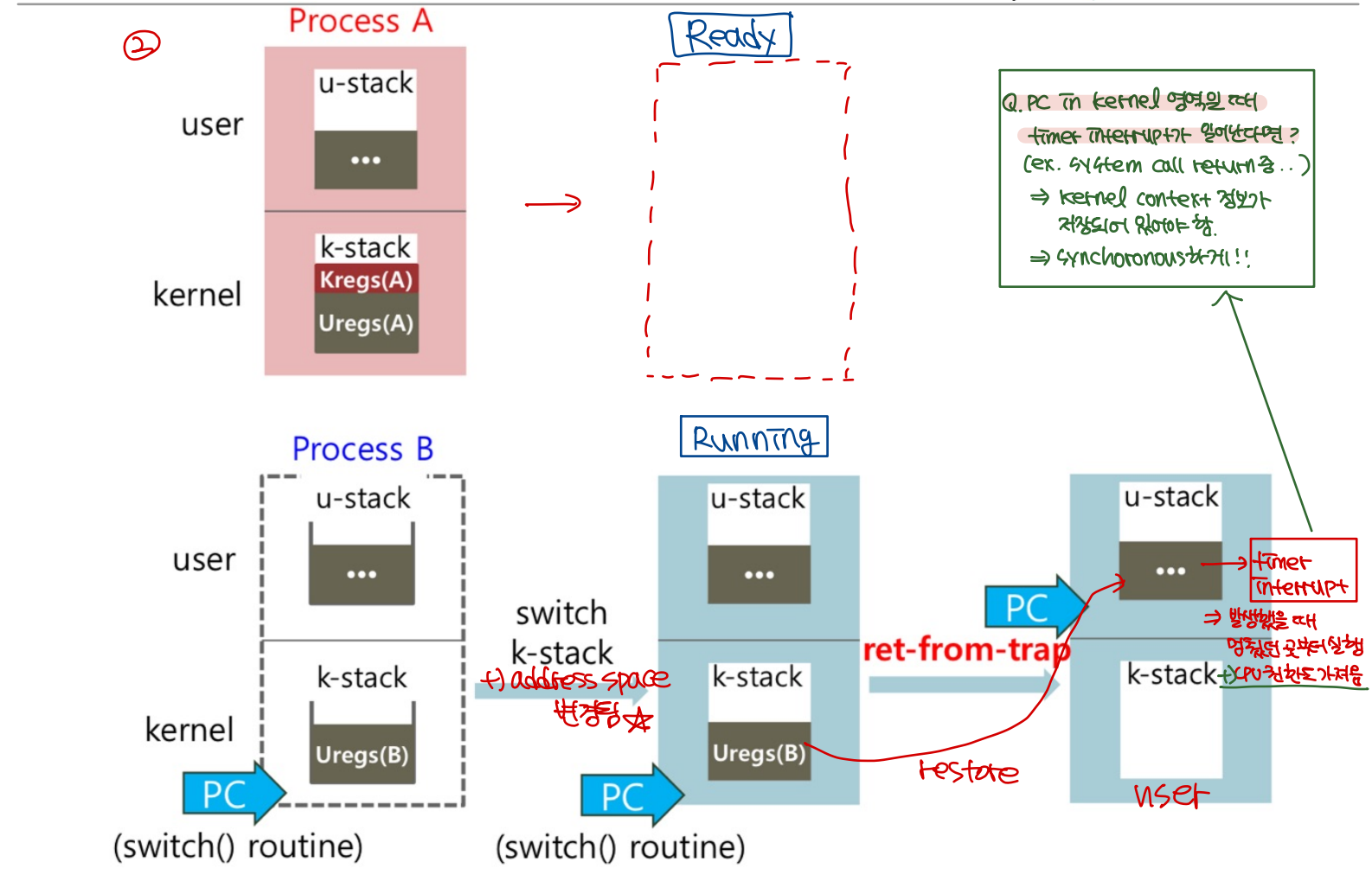
- **Limited Direct Execution Protocol**

부팅될 때 적용 → process마다 실행 X

OS	Hardware (CPU)	Program
Initialize trap table (최초)	Remember address of syscall handler timer handler (CPU 상실)	
Start interrupt timer	Start timer (interrupt CPU in X *ms*) (timer H/W)	
...		Process A ⇒ running
	Timer interrupt (kernel stack에 저장) Save Uregs(A) to k-stack(A) Move to kernel mode Jump to trap handler → 일종의 interrupt 종류	
Handle the trap Call switch() routine save Kregs(A) to k-stack(A)* restore Kregs(B) from k-stack(B)* switch to k-stack(B) (process A의 context로 기억해서 실행)		timer interrupt
Return-from-trap (into B)	Restore Uregs(B) from k-stack(B) Move to user mode Jump to B's PC	
		Process B ⇒ ready

* Can be either k-stack or PCB (stack 혹은 PCB에 저장)


①
Running
Process A
user
kernel
u-stack
k-stack
PC
trap
(timer)
Uregs(A)
save
Kregs(A)
Kregs(A)
Uregs(A)
context 정보 저장
user mode
kernel mode
(Interrupt handler or
Scheduler)
(switch()
routine)
Ready
Process B
context switch
restore
Kregs(B)
Uregs(B)
PC 이동
but. address
정보는 없음.
Restore
(switch()
routine)


②
Process A
Ready
user
kernel
u-stack
k-stack
Kregs(A)
Uregs(A)
Q. PC in kernel 영역일 때
timer interrupt가 일어난다면?
(ex. system call return중..)
⇒ kernel context 정보가
저장되어 있어야 함.
⇒ synchronous하게!!
Process B
Running
switch
k-stack
+) address space
변경됨☆
Uregs(B)
ret-from-trap
timer
interrupt
⇒ 발생했을 때
명령어 그대로실행
+) CPU 권한도 가져옴
restore
user
(switch() routine)
(switch() routine)