

# TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)  
[gcccompil3r@gmail.com](mailto:gcccompil3r@gmail.com)

학생 – GJ (박현우)  
[uc820@naver.com](mailto:uc820@naver.com)

# 1. 리눅스 커널 내부 구조 - Chapter 3 (태스크 관리)

Chapter 3 태스크 관리

③ 프로세스와 스레드의 생성과 수행

- 프로세스는 어떻게 생성??  $\text{fork() or vfork}$
- 두 함수의 차이??  $\text{fork()}$  vs  $\text{vfork()}$ 
  - $\text{fork()}$  후  $\text{exec()}$ 을 하면  $\text{fork()}$ 를 할 때 부모의 memory layout이 복사된다. 기다가  $\text{exec()}$ 을 하면 메모리를 덮어 씌움.
  - 즉) 이중으로 덮어쓰는 불필요한 행위를 함.
  - 그럼이 때문에,  $\text{vfork()}$ 가 생김. 복사는 1번만!!
  - $\text{no cow!}$
- 프로세스와 스레드의 생성시 차이점!!
  - $\Rightarrow$  프로세스는  $\text{fork()}$ 시에 각각 다른 메모리를 사용하므로 변수의 값이 다르게 출력!
  - But, 스레드는 메모리를 공유하므로 변수에 영향을 끼친다.

- $\text{clone()}$  함수 사용법!

$\text{clone}(\text{sub\_func}, (\text{void} *) (\text{child\_stack} + 4095),$   
           $\text{CLONE\_VM} \mid \text{CLONE\_THREAD} \mid \text{CLONE\_SIG}$   
           $\text{HAND}, \text{NULL});$

$\text{sub\_func}$                        $\text{child\_stack}$ 가 쓸 스택공간 (4KB)  
          가상메모리                      스레드(메모리 공유)                      시그널 처리  
          용선(시그)

$\Rightarrow$  프로세스나 스레드가 될 수 있다.

추가  $\#define \_GNU\_SOURCE$

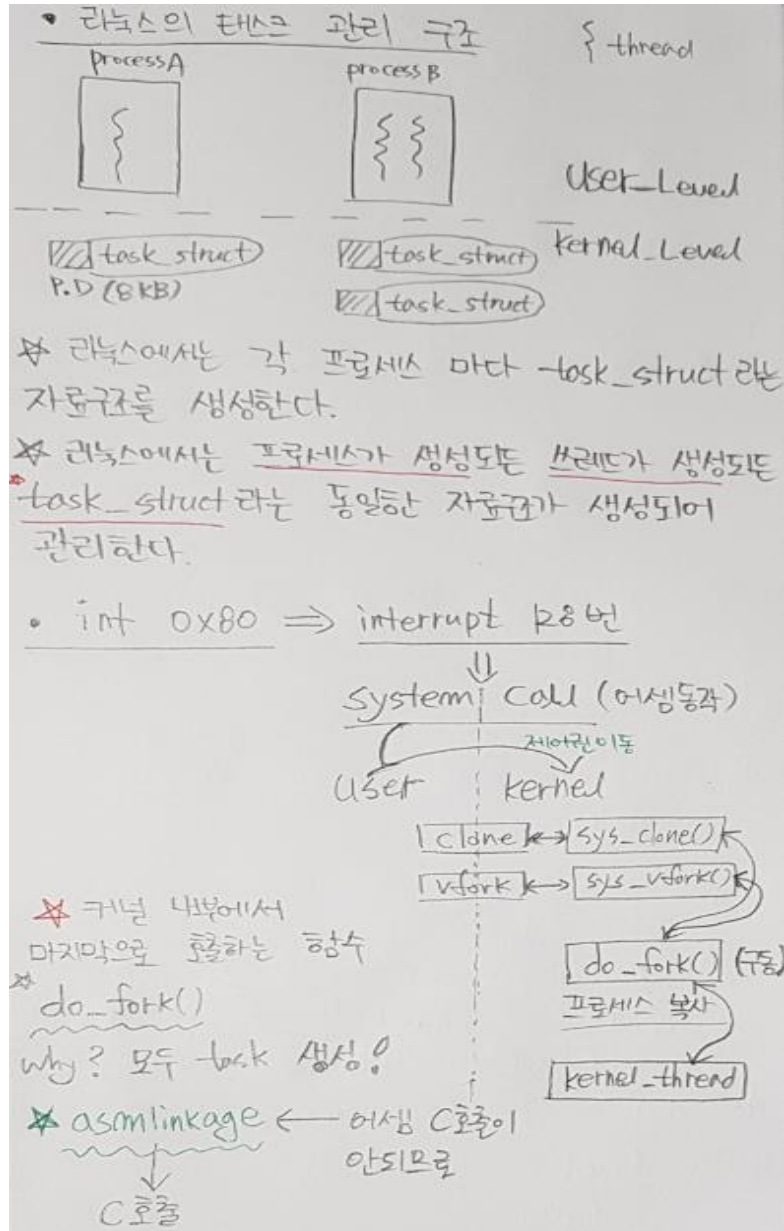
$\star$  새로운 스레드를 생성하면 서로 같은 주소공간을 공유한다.

$\Downarrow$   
critical section  
방지

세마포어, 스핀락, 뮤텁스

$\star$  프로세스 간 메모리 공유  
 $\Rightarrow$  Shared Memory, Message Queue 등

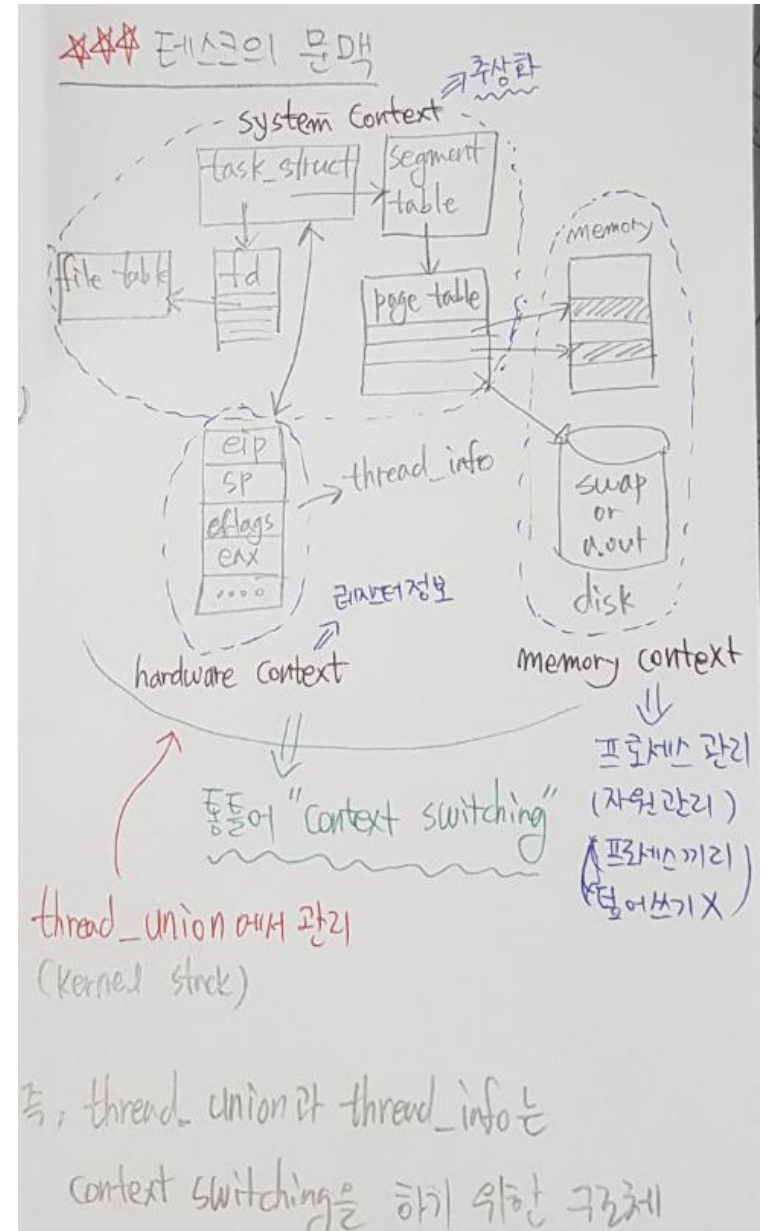
# 1. 리눅스 커널 내부 구조 - Chapter 3 (태스크 관리)



• do\_fork() 역할  
⇒ 새로 생성되는 태스크를 위해 pid 이름표 준비,  
부모님 이름, 소지품 등 자세한 정보 기록 = task\_struct  
ppid Memory Layout

• pid와 tgid  
tgid (thread group id)  
⇒ getpid() 함수는 사실 tgid 값임.  
pid (process id)  
⇒ gettid() 함수이다. (syscall(\_\_NR\_gettid))

• 커널 내의 getpid()  
⇒ current는 (thread\_info → task) 형제  
구조체인 task이 \*task\_struct이다.  
SYSCALL\_DEFINE0(getpid)  
{  
return task\_tgid\_vnr(current);  
}  
getpid = tgid를 리턴하는 함수



# 1. 리눅스 커널 내부 구조 - Chapter 3 (태스크 관리)

• task\_struct 변수 이름 정리

1) task identification  
uid (제정), euid (sudo) (권한이 있는지?)

2) state  
TASK\_RUNNING(0), TASK\_INTERRUPTIBLE(1)  
구동 인터럽트 수신  
TASK\_UNINTERRUPTIBLE(2)  
인터럽트 허용 X  
TASK\_STOPPED(4), TASK\_TRACED(8)  
일시정지 디버깅 상태  
EXIT\_DEAD(16), EXIT\_ZOMBIE(32)  
정상종료 return 0 parent가 child 처리 X  
시그널

3) scheduling information  
prio, policy, cpus\_allowed, time\_slice,  
rt\_priority  
↓  
우선순위  
• sched\_rt\_entity ⇒ real-time schedule  
• sched\_entity ⇒ user

4) signal information  
• signal\_struct (시그널 번호 관리)  
• sigband\_struct (어떤 시그널 인지)  
• blocked, real\_blocked ⇒ masking 0, 1  
비트연산  
• pending (지연)

7) cpu\_context\_save  
hardware context

8) format  
• elf → linux\_binfmt  
• dwarf  
↓  
thread\_info (preemption syscall 등)

• 상태전이 (state transition)와 실행수준 변화

1) 부모가 wait 함수 호출 → 자식 태스크  
TASK\_DEAD (EXIT\_DEAD)  
return 0;

2) 부모가 wait 함수 호출 전 종료  
⇒ 부모없는 ZOMBIE 태스크 → init 태스크  
1번 process

3) 리눅스는 여러 태스크들이 CPU를 '공정하게'  
사용하게 해주기 위해 일반 태스크는 'CFS'  
기법을 사용.

4) 디버깅되고 있는 태스크는 TASK\_TRACED  
상태로 전이.

5) 사용자 수준 실행 → 커널 수준 실행  
↑  
과거 방법 시스템 호출, 인터럽트

6) 커널도 (스택)을 필요로 함.

2) kernel\_stack ⇒ thread\_union  
thread\_info  
(프로세스 디스크립터) (PD)  
8KB (ARM) 16KB (Intel)



# 1. 리눅스 커널 내부 구조 - Chapter 3 (태스크 관리)

