기본

1. user app - fork, vfork, clone (우리가 만든것)이를 구동 시키면 → c liberary 가 돌아간다. fork,vfork,clone (커널개발자가 만든것) 에 어셈블리 코드가 들어가 가지고 int 0x80 적혀있다.→ 인터럽트 128번을 호출해라. 인터럽트 128번을 호출하는 순간 제어권이 커널로 된다. (sys_call이 보이징?) 시스템콜 처리를 하는데, Sys_ 붙은 애들을 모두 처리하게 된다. sys_clone, sys_vfork, sys_fork는 어디에 연결되있을까? do_fork에 연결되어 있고, 이들은 모두 커널 쓰레드를 생성한다.

= 결론적으로 vfork,fork, clone이든 뭐든 커널 내부에서는 커널 쓰레드로 관리한다는 것이고, 이들을 구동 시키는것은 옵션의 차이를 둔 do_fork이다. fork()와 clone() 둘 모두 커널 입장에서는 모두 태스크를 생성하기 때문이다. fork는 비교적 부모 태스크와 덜 공유 하는 태스크이고, clone()으로 생성되는 태스크는 비교적 부모 태스크와 많이 공유하는 태스크이다. 즉, do_fork()를 호출할 때 이 함수의 인자로 부모 태스크와 얼마나 공유할 지를 정해 줌으로써 fork()와 clone()함수 둘 다를 지원 할 수 있는 것이다.

do_fork하는일 : 새로 생성되는 태스크를 위해 일종의 이름표를 하나 준비한다. 이 이름표에 새로 생성된 태스크의 이름과 태어난 시간, 부모님 이름, 소지품 등 자세한 정보를 기록해 둔다. 나중에 생성된 태스크를 쉽게 찾고, 정보를 알려 하는 것이다. 여기서 이 이름표는 **task_struct 구조체**이다. 태스크가 수행되기 위해 필요한 자원 등을 할당하고, 수행 가능한 상태로 만드는 역할을 한다.

----------sys_fork 분석(code)----------

grep -rn "fork" ./ | grep "SYSCALL"

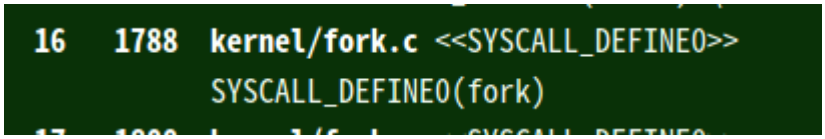
```
./arch/s390/kernel/syscalls.S:13:SYSCALL(sys_fork,sys_fork)
./arch/s390/kernel/syscalls.S:201:SYSCALL(sys_vfork,sys_vfork)                  /* 190 */
./arch/arm/include/uapi/asm/unistd.h:30:#define __NR_fork                    (__NR_SYSCALL_BASE+  2)
./arch/arm/include/uapi/asm/unistd.h:218:#define __NR_vfork                   (__NR_SYSCALL_BASE+190)
./kernel/fork.c:1788:SYSCALL_DEFINE0(fork)
./kernel/fork.c:1800:SYSCALL_DEFINE0(vfork)
```

```
#define SYSCALL_DEFINE0(sname)
        SYSCALL_METADATA(_##sname, 0);
```

SYSCALL_METADATA(##sname,0);

asmlinkage long sys_##sname(void): 시스템콜은 어셈블리어로 동작하기때문에 어셈블리어는 c 코드를 불러올 수 없기 때문에 c를 호출하려고 넣어준다.

→ SYSCALL_DEFINE0은 뭘까?




```
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
        return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
#else
        /* can not support in nommu mode */
        return -EINVAL;
#endif
}
#endif
```

do_fork 가 보인다. sys_fork, sys_vfork, sys_clone 모두 연결되어 있다. do_fork를 분석하면 된다.

```
long _do_fork(unsigned long clone_flags,  // SIGCHLD, 0x11~14
          unsigned long stack_start, // 0
          unsigned long stack_size,  // 0
          int __user *parent_tidptr,  // NULL
          int __user *child_tidptr,     // NULL
          unsigned long tls)            // 0
{
      struct task_struct *p;
      int trace = 0;
      long nr;

      /*
       * Determine whether and which event to report to ptracer.  When
       * called from kernel_thread or CLONE_UNTRACED is explicitly
       * requested, no event is reported; otherwise, report if the event
       * for the type of forking is enabled.
       */
```

```
    // SIGCHLD
    if (!(clone_flags & CLONE_UNTRACED)) {// 0x11 & 0x008000000
        if (clone_flags & CLONE_VFORK)  // 0x11 & 0x00004000
            trace = PTRACE_EVENT_VFORK; //1
        else if ((clone_flags & CSIGNAL) != SIGCHLD) //0x11& 0x000000ff
            trace = PTRACE_EVENT_CLONE; //3
        else
            trace = PTRACE_EVENT_FORK;

/* current 현재 구동중인 task_struct, likely를 주면 컴파일러에게 해당 분기문에서 참인 경우가 더 많을 것
이라는 정보를 주어 성능을 추가적으로 성능 시키는 함수고,  ptrace는 디버깅을 하는지 판단,
ptrace_event_enable → currnt현재 구동중인 태스크에 문제가 있는지 없는지 확인하고, 없다면 trace= 0
        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }

    // 자식프로세스
    p = copy_process(clone_flags,  // 0x11
                     stack_start,  // 0
                      stack_size,  // 0
                     child_tidptr,  // NULL
                             NULL,
                              trace, // 0
                                 tls); // 0
    /*
     * Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
     */
         if (!IS_ERR(p)) {
        struct completion vfork;
        struct pid *pid;

        trace_sched_process_fork(current, p);

        pid = get_task_pid(p, PIDTYPE_PID);
        nr = pid_vnr(pid);

        if (clone_flags & CLONE_PARENT_SETTID)
            put_user(nr, parent_tidptr);

        if (clone_flags & CLONE_VFORK) {
            p->vfork_done = &vfork;
            init_completion(&vfork);
            get_task_struct(p);
        }
```

```
        wake_up_new_task(p);

        /* forking complete and child started to run, tell ptracer */
        if (unlikely(trace))
            ptrace_event_pid(trace, pid);

        if (clone_flags & CLONE_VFORK) {
            if (!wait_for_vfork_done(p, &vfork))
                ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
        }

        put_pid(pid);
    } else {
        nr = PTR_ERR(p);
    }
    return nr;
}

..
```