Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

#fork

강사 : Innova Lee(이 상훈)

학생 : 김 시윤

1.배운내용 복습.

SYSCALL_FORK Driving

---

1. SYSCALL_DEFINE0(fork)

```
#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
}
```

systemcall 로 정의 되어있는 fork를 찾는다.
_do_fork 함수에 인자로 SIGCHLD와 0 , 0 , NULL, NULL, 0 이 넘어가 받아온 리턴값을 리턴한다. 그러므로 _do_fork를 확인한다.

2. _do_fork

```
long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
/*      clone_flags = SIGCHLD
        stack_start = 0
        stack_size = 0
        int__user *parent_tidptr = NULL
        int __user *child_tiptr = NULL
        tls = 0                         */
        struct task_struct *p;
        int trace = 0;
        long nr;
        /* trace 는 디버깅 용도
           task_struct *p 는 자식 프로세스
           nr 은 자식 프로세스의 pid 가 된다. */

        /*
         * 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.
         */
         /*#define CLONE_UNTRACED 0x00800000
           if문 값이 0일때 통과이기 때문에 통과 */
        if (!(clone_flags & CLONE_UNTRACED)) {
                /* #define CLONE_VFORK 0x00004000
                SIGCHLD = 17 과 &연산하면 0 이프문 통과 못함 */
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if ((clone_flags & CSIGNAL) != SIGCHLD)//여기서도 엘즈 통과함
                        trace = PTRACE_EVENT_CLONE;
                else
                        /*결국 fork 했을시에 PTRACE_EVENT_FORK통과한다 clone_flags에 들어오는
                        인자값에 의해 어떤 동작을 실행시켰는지 검사하는 로직 */
                        trace = PTRACE_EVENT_FORK;
                /* current = 현재 구동중인 task
                   trace = 1
                */
                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }
        /* trace = 0
           clone_flags = SIGCHLD
           stack_start = 0
           stack_size = 0
           child_tidptr = NULL
           NULL
           tls = 0
                        */
        /*복사된 프로세스가 페이 저장*/
        p = copy_process(clone_flags, stack_start, stack_size,
                         child_tidptr, NULL, trace, tls);
        /*
         * 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)) {
```

```
        if (!IS_ERR(p)) {
                struct completion vfork;
                struct pid *pid;

                trace_sched_process_fork(current, p);

                /*pid 값을 얻어옴 */
                pid = get_task_pid(p, PIDTYPE_PID);
                /*자식프로세스의 pid를 nr에 저장*/
                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);
                }
                /*run q 에 새로운 task 대기 */
                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;
        /*pid return */
}
```

long _do_fork(unsigned long clone_flags, unsigned long stack_start, unsigned long stack_size, int __user *parent_tidptr, int __user *child_tidptr, unsigned long tls)
do_fork 함수는 위와같은 인자를 갖는다. 이 이자로 넘어온 값들을 나열해보면

clone_flags = SIGCHLD
stack_start = 0
stack_size = 0
parent_tidptr = NULL
child_tidptr = NULL
tls = 0
가 되고 여기서 clone_flags 에 의해 fork 인지 vfork 인지 clone 인지 디버깅을 거쳐 trace 값이 바뀌고 그 trace 값에 의해 그에 해당하는 작업을 수행시켜준다.

p= copy_process(clone_flags, stack_start, stack_size, child_tidptr, NULL, trace, tls) 의 값이 넘어간다.
그럼 copy_process를 확인한다.
copy_process 는 복사할 프로세스의 데이터를 저장한다.
밑에 pid = get_task_pid(p,PIDTYPE_PID);
복사할 프로세스 즉 부모 프로세스의 pid 값을 얻어온다.

nr = pid_vnr(pid) 자식 프로세스의 pid 값을 얻어 nr에 저장한다.

- copy_process

```
/* trace = 0
   clone_flags = SIGCHLD
   stack_start = 0
   stack_size = 0
   child_tidptr = NULL
   pid *pid = NULL
   tls = 0
 */
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid,
                                        int trace,
                                        unsigned long tls)

{
        int retval;
        struct task_struct *p;
        void *cgrp_ss_priv[CGROUP_CANFORK_COUNT] = {};

        /* 정상적인 fork를 했을경우 밑에 있는 에러검출문은 다 통과하지 않는다. */

        /*에러일 경우 에러 내용
        새로운 mount namespace로 태스크 생성을 요청했지만 파일 시스템 정보를 공유할 수 없으므로 실패로 함수를 빠져나간다.*/
        if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
                return ERR_PTR(-EINVAL);
        /*에러일 경우 에러 내용
        새로운 user namespace로 태스크 생성을 요청했지만 파일 시스템 정보를 공유할 수 없으므로 실패로 함수를 빠져나간다.*/
        if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
                return ERR_PTR(-EINVAL);

        /*
         * Thread groups must share signals as well, and detached threads
         * can only be started up within the thread group.
         */
        /*thread 생성을 요청했지만 시그널 핸들러의 공유가 필요하다. 그렇지 않은 경우 실패로 함수를 빠져나간다.
          (thread 는 signal 공유 */
        if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
                return ERR_PTR(-EINVAL);

        /*
         * Shared signal handlers imply shared VM. By way of the above,
         * thread groups also imply shared VM. Blocking this case allows
         * for various simplifications in other code.
         */
         /*시그널 핸들러를 공유할 때 같은 가상 주소(VM)를 사용해야한다. 그렇지 않은 경우 실패로 함수를 빠져나간다.*/
        if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
                return ERR_PTR(-EINVAL);

        /*
         * Siblings of global init remain as zombies on exit since they are
         * not reaped by their parent (swapper). To solve this and to avoid
         * multi-rooted process trees, prevent global and container-inits
         * from creating siblings.
         */
         /* 부모 태스크에 SIGNAL_UNKILLABLE 시그널 플래그가 있는 경우 부모 태스크 클론을 요청한 경우 실패로 함수를 빠져나간다.*/
        if ((clone_flags & CLONE_PARENT) &&
                                current->signal->flags & SIGNAL_UNKILLABLE)
                return ERR_PTR(-EINVAL);
```

```
        if (clone_flags & CLONE_THREAD) {
                if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) ||
                    (task_active_pid_ns(current) !=
                                current->nsproxy->pid_ns_for_children))
                        return ERR_PTR(-EINVAL);
        }
        /* task_ops return 지금 만들어져 아무것도 없기 때문에 값은 0 */
        retval = security_task_create(clone_flags);
        /*retval = 0*/
        if (retval)
                goto fork_out;

        retval = -ENOMEM;
        /*current = 현재 구동중인 task */
        p = dup_task_struct(current);
        /* p 에 원래의 task가 복사됨 */
        if (!p)
                goto fork_out;

        ftrace_graph_init_task(p);

        rt_mutex_init_task(p);

#ifdef CONFIG_PROVE_LOCKING
        DEBUG_LOCKS_WARN_ON(!p->hardirqs_enabled);
        DEBUG_LOCKS_WARN_ON(!p->softirqs_enabled);
#endif
        retval = -EAGAIN;
        if (atomic_read(&p->real_cred->user->processes) >=
                        task_rlimit(p, RLIMIT_NPROC)) {
                if (p->real_cred->user != INIT_USER &&
                    !capable(CAP_SYS_RESOURCE) && !capable(CAP_SYS_ADMIN))
                        goto bad_fork_free;
        }
        /*define PF_NPROC_EXCEEDED 0x00001000 */
        current->flags &= ~PF_NPROC_EXCEEDED;
        /*새로운 namespace 할당? */
        retval = copy_creds(p, clone_flags);
        if (retval < 0)
                goto bad_fork_free;

        /*
         * If multiple threads are within copy_process(), then this check
         * triggers too late. This doesn't hurt, the check is only there
         * to stop root fork bombs.
         */
        retval = -EAGAIN;
        if (nr_threads >= max_threads)
                goto bad_fork_cleanup_count;

        delayacct_tsk_init(p);  /* Must remain after dup_task_struct() */
        p->flags &= ~(PF_SUPERPRIV | PF_WQ_WORKER);
        p->flags |= PF_FORKNOEXEC;
        INIT_LIST_HEAD(&p->children);
        INIT_LIST_HEAD(&p->sibling);
        rcu_copy_process(p);
        p->vfork_done = NULL;
        spin_lock_init(&p->alloc_lock);

        init_sigpending(&p->pending);
        //signal pending
```

```
        p->utime = p->stime = p->gtime = 0;
        p->utimescaled = p->stimescaled = 0;
        prev_cputime_init(&p->prev_cputime);

#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
        seqlock_init(&p->vtime_seqlock);
        p->vtime_snap = 0;
        p->vtime_snap_whence = VTIME_SLEEPING;
#endif

#if defined(SPLIT_RSS_COUNTING)
        memset(&p->rss_stat, 0, sizeof(p->rss_stat));
#endif

        p->default_timer_slack_ns = current->timer_slack_ns;

        task_io_accounting_init(&p->ioac);
        acct_clear_integrals(p);

        posix_cpu_timers_init(p);

        p->start_time = ktime_get_ns();
        p->real_start_time = ktime_get_boot_ns();
        p->io_context = NULL;
        p->audit_context = NULL;
        threadgroup_change_begin(current);
        cgroup_fork(p);
#ifdef CONFIG_NUMA
        p->mempolicy = mpol_dup(p->mempolicy);
        if (IS_ERR(p->mempolicy)) {
                retval = PTR_ERR(p->mempolicy);
                p->mempolicy = NULL;
                goto bad_fork_cleanup_threadgroup_lock;
        }
#endif
#ifdef CONFIG_CPUSETS
        p->cpuset_mem_spread_rotor = NUMA_NO_NODE;
        p->cpuset_slab_spread_rotor = NUMA_NO_NODE;
        seqcount_init(&p->mems_allowed_seq);
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
        p->irq_events = 0;
        p->hardirqs_enabled = 0;
        p->hardirq_enable_ip = 0;
        p->hardirq_enable_event = 0;
        p->hardirq_disable_ip = _THIS_IP_;
        p->hardirq_disable_event = 0;
        p->softirqs_enabled = 1;
        p->softirq_enable_ip = _THIS_IP_;
        p->softirq_enable_event = 0;
        p->softirq_disable_ip = 0;
        p->softirq_disable_event = 0;
        p->hardirq_context = 0;
        p->softirq_context = 0;
#endif

        p->pagefault_disabled = 0;

#ifdef CONFIG_LOCKDEP
        p->lockdep_depth = 0; /* no locks held yet */
        p->curr_chain_key = 0;
        p->lockdep_recursion = 0;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        p->blocked_on = NULL; /* not blocked yet */
#endif
#ifdef CONFIG_BCACHE
        p->sequential_io        = 0;
        p->sequential_io_avg    = 0;
#endif

        /* Perform scheduler related setup. Assign this task to a CPU. */
        retval = sched_fork(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_policy;

        retval = perf_event_init_task(p);
        if (retval)
                goto bad_fork_cleanup_policy;
        retval = audit_alloc(p);
        if (retval)
                goto bad_fork_cleanup_perf;
        /* copy all the process information */
        shm_init_task(p);
        retval = copy_semundo(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_audit;
        retval = copy_files(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_semundo;
        retval = copy_fs(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_files;
        retval = copy_sighand(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_fs;
        retval = copy_signal(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_sighand;
        retval = copy_mm(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_signal;
        retval = copy_namespaces(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_mm;
        retval = copy_io(clone_flags, p);
        if (retval)
                goto bad_fork_cleanup_namespaces;
        retval = copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);
        if (retval)
                goto bad_fork_cleanup_io;

        if (pid != &init_struct_pid) {
                pid = alloc_pid(p->nsproxy->pid_ns_for_children);
                if (IS_ERR(pid)) {
                        retval = PTR_ERR(pid);
                        goto bad_fork_cleanup_io;
                }
        }
```

```
                        list_add_tail_rcu(&p->thread_node,
                                          &p->signal->thread_head);
                }
                attach_pid(p, PIDTYPE_PID);
                nr_threads++;
        }

        total_forks++;
        spin_unlock(&current->sighand->siglock);
        syscall_tracepoint_update(p);
        write_unlock_irq(&tasklist_lock);

        proc_fork_connector(p);
        cgroup_post_fork(p, cgrp_ss_priv);
        threadgroup_change_end(current);
        perf_event_fork(p);

        trace_task_newtask(p, clone_flags);
        uprobe_copy_process(p, clone_flags);

        return p;

bad_fork_cancel_cgroup:
        cgroup_cancel_fork(p, cgrp_ss_priv);
bad_fork_free_pid:
        if (pid != &init_struct_pid)
                free_pid(pid);
bad_fork_cleanup_io:
        if (p->io_context)
                exit_io_context(p);
bad_fork_cleanup_namespaces:
        exit_task_namespaces(p);
bad_fork_cleanup_mm:
        if (p->mm)
                mmput(p->mm);
bad_fork_cleanup_signal:
        if (!(clone_flags & CLONE_THREAD))
                free_signal_struct(p->signal);
bad_fork_cleanup_sighand:
        __cleanup_sighand(p->sighand);
bad_fork_cleanup_fs:
        exit_fs(p); /* blocking */
bad_fork_cleanup_files:
        exit_files(p); /* blocking */
bad_fork_cleanup_semundo:
        exit_sem(p);
bad_fork_cleanup_audit:
        audit_free(p);
bad_fork_cleanup_perf:
        perf_event_free_task(p);
bad_fork_cleanup_policy:
#ifdef CONFIG_NUMA
        mpol_put(p->mempolicy);
bad_fork_cleanup_threadgroup_lock:
#endif
        threadgroup_change_end(current);
        delayacct_tsk_free(p);
bad_fork_cleanup_count:
        atomic_dec(&p->cred->user->processes);
        exit_creds(p);
bad_fork_free:
```

- dup_task_struct

```
static struct task_struct *dup_task_struct(struct task_struct *orig)
{
/*부모 프로세스가 인자로 함수 진입 */
        struct task_struct *tsk;
        struct thread_info *ti;
        /* 접근 가능한 node(메모리)를 할당해준다. */
        int node = tsk_fork_get_node(orig);
        int err;
        /*위에서 할당한 노드가 들어감 */
        /*실제 할당된 slab메모리 리턴 */
        tsk = alloc_task_struct_node(node);
        if (!tsk)
                return NULL;
        /*원래 태스크의 메모리가 복사됨 페이지크기보다 클때는 다른페이지 연결리스트로 관리됨 */
        ti = alloc_thread_info_node(tsk, node);
        if (!ti)
                goto free_tsk;

        err = arch_dup_task_struct(tsk, orig);
        if (err)
                goto free_ti;

        tsk->stack = ti;
#ifdef CONFIG_SECCOMP
        /*
         * We must handle setting up seccomp filters once we're under
         * the sighand lock in case orig has changed between now and
         * then. Until then, filter must be NULL to avoid messing up
         * the usage counts on the error path calling free_task.
         */
        tsk->seccomp.filter = NULL;
#endif

        setup_thread_stack(tsk, orig);
        clear_user_return_notifier(tsk);
        clear_tsk_need_resched(tsk);
        set_task_stack_end_magic(tsk);

#ifdef CONFIG_CC_STACKPROTECTOR
        tsk->stack_canary = get_random_int();
#endif

        /*
         * One for us, one for whoever does the "release_task()" (usually
         * parent)
         */
        atomic_set(&tsk->usage, 2);
#ifdef CONFIG_BLK_DEV_IO_TRACE
        tsk->btrace_seq = 0;
#endif
        tsk->splice_pipe = NULL;
        tsk->task_frag.page = NULL;
        tsk->wake_q.next = NULL;

        account_kernel_stack(ti, 1);
        /*복사된 task가 리턴됨 */
        return tsk;

free_ti:
        free_thread_info(ti);
free_tsk:
```

```c
/* called from do_fork() to get node information for about to be created task */
int tsk_fork_get_node(struct task_struct *tsk)
{
/*부모 task 들어옴 */
#ifdef CONFIG_NUMA
        if (tsk == kthreadd_task)
                return tsk->pref_node_fork;
                /*NUMA 일때 노드가 여러개이므로 task_struct 에서 노드를 얻어온다.
                부모프로세스 태스크와 kthreadd_task(task_struct)가 같을때, */
#endif
        return NUMA_NO_NODE;
        /*그게 아니면 우마이기 때문에 노드는 한개이므로 리턴 */
}
```

```c
static inline struct task_struct *alloc_task_struct_node(int node)
{
        /* origin이 사용하는 노드
           task가 갖고있는 시피유 캐시와, GFP_KERNEL = 메모리 할당 옵션 , 원래 태스크가 쓰던 노드*/
        return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
}
```

```c
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
{
        /* task_cpu_cache , GFP_KERNEL 메모리 할당 옵션이 넘어감 */
        void *ret = slab_alloc(cachep, flags, _RET_IP_);

        trace_kmem_cache_alloc(_RET_IP_, ret,
                               cachep->object_size, cachep->size, flags);

        return ret;
}
EXPORT_SYMBOL(kmem_cache_alloc);
```

```c
slab_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid,
                unsigned long caller)
{
        unsigned long save_flags;
        void *ptr;
        /* NUMA ID 값 설정 */
        int slab_node = numa_mem_id();

        /*flags = GFP_KERNEL */
        flags &= gfp_allowed_mask;
        lockdep_trace_alloc(flags);

        if (slab_should_failslab(cachep, flags))
                return NULL;
        /* slab을 할당받아 반환해줌 */
        cachep = memcg_kmem_get_cache(cachep, flags);

        cache_alloc_debugcheck_before(cachep, flags);
        /*Interrupt 를 끄고 flags가 리턴됨 */
        local_irq_save(save_flags);

        if (nodeid == NUMA_NO_NODE)
                nodeid = slab_node;

        /*get_node 원래 노드 s에 노드 할당 */
        if (unlikely(!get_node(cachep, nodeid))) {
                /* Node not bootstrapped yet */
                ptr = fallback_alloc(cachep, flags);
                goto out;
        }
        /* nodeid 와 slab_node 는 같지 않다 */
        if (nodeid == slab_node) {
                /*
                 * Use the locally cached objects if possible.
                 * However ____cache_alloc does not allow fallback
                 * to other nodes. It may fail while we still have
                 * objects on other nodes available.
                 */
                ptr = ____cache_alloc(cachep, flags);
                if (ptr)
                        goto out;
        }
        /* ___cache_alloc_node can fall back to other nodes */
        /*obj return 실제 메모리 할당 */
        ptr = ____cache_alloc_node(cachep, flags, nodeid);
 out:
        local_irq_restore(save_flags);
        ptr = cache_alloc_debugcheck_after(cachep, flags, ptr, caller);
        kmemleak_alloc_recursive(ptr, cachep->object_size, 1, cachep->flags,
                                 flags);

        if (likely(ptr)) {
                kmemcheck_slab_alloc(cachep, flags, ptr, cachep->object_size);
                if (unlikely(flags & __GFP_ZERO))
                        memset(ptr, 0, cachep->object_size);
        }

        memcg_kmem_put_cache(cachep);
        return ptr;
```

```
struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep)
{
        struct mem_cgroup *memcg;
        struct kmem_cache *memcg_cachep;
        int kmemcg_id;

        VM_BUG_ON(!is_root_cache(cachep));

        /* current = task_struct
           CONFIG_MEMCG 가 설정되있으면
           memcg_kmem_skip_account:1;
           1비트만 사용한다.    fork를 하는경우 설정 안되있다.  */
        if (current->memcg_kmem_skip_account)
                return cachep;
        /*retrun memcg */
        memcg = get_mem_cgroup_from_mm(current->mm);
        /*kernel memory cg id 값 구해옴 */
        kmemcg_id = READ_ONCE(memcg->kmemcg_id);
        if (kmemcg_id < 0)
                goto out;

        /*cache index return */
        /*여기서 slab을 할당 받음*/
        memcg_cachep = cache_from_memcg_idx(cachep, kmemcg_id);
        if (likely(memcg_cachep))
                return memcg_cachep;

        /*
         * If we are in a safe context (can wait, and not in interrupt
         * context), we could be be predictable and return right away.
         * This would guarantee that the allocation being performed
         * already belongs in the new cache.
         *
         * However, there are some clashes that can arrive from locking.
         * For instance, because we acquire the slab_mutex while doing
         * memcg_create_kmem_cache, this means no further allocation
         * could happen with the slab_mutex held. So it's better to
         * defer everything.
         */
        memcg_schedule_kmem_cache_create(memcg, cachep);
out:
        css_put(&memcg->css);
        return cachep;
}
```

```
static struct mem_cgroup *get_mem_cgroup_from_mm(struct mm_struct *mm)
{
        struct mem_cgroup *memcg = NULL;

        rcu_read_lock();
        do {
                /*
                 * Page cache insertions can happen withou an
                 * actual mm context, e.g. during disk probing
                 * on boot, loopback IO, acct() writes etc.
                 */
                 /*mm 의 유무 확인
                   mm 이 존재하므로 else 로 갑니다.*/
                if (unlikely(!mm))
                        memcg = root_mem_cgroup;
                else {
                        /*css 있으면 css리턴 없으면 NULL리턴
                          정보 있기때문에 css리턴되서 if문 통과 못함*/
                        memcg = mem_cgroup_from_task(rcu_dereference(mm->owner));
                        if (unlikely(!memcg))
                                memcg = root_mem_cgroup;
                }
        } while (!css_tryget_online(&memcg->css));
        rcu_read_unlock();
        return memcg;
        /*memcg 값 있어서 리턴 */
}
```

```
struct mem_cgroup *mem_cgroup_from_task(struct task_struct *p)
{
        /*
         * mm_update_next_owner() may clear mm->owner to NULL
         * if it races with swapoff, page migration, etc.
         * So this can be called with p == NULL.
         */
        if (unlikely(!p))
                return NULL;

        /*css 있으면 css리턴 없으면 NULL리턴 */
        return mem_cgroup_from_css(task_css(p, memory_cgrp_id));
}
EXPORT_SYMBOL(mem_cgroup_from_task);
```

```
cache_from_memcg_idx(struct kmem_cache *s, int idx)
{
        struct kmem_cache *cachep;
        struct memcg_cache_array *arr;

        rcu_read_lock();
        /* s = cachep */
        /* 부노가 쓰던 메모리를 복사했기 때문에 s에 값이 있다 그걸 arr에 넣고*/
        arr = rcu_dereference(s->memcg_params.memcg_caches);

        /*
         * Make sure we will access the up-to-date value. The code updating
         * memcg_caches issues a write barrier to match this (see
         * memcg_create_kmem_cache()).
         */
         /*arr 를 cachep로 가져온다*/
        cachep = lockless_dereference(arr->entries[idx]);
        rcu_read_unlock();

        return cachep;
}
```

```
static void *____cache_alloc_node(struct kmem_cache *cachep, gfp_t fla
                                  int nodeid)
{
        struct list_head *entry;
        struct page *page;
        struct kmem_cache_node *n;
        void *obj;
        int x;

        VM_BUG_ON(nodeid < 0 || nodeid >= MAX_NUMNODES);
        n = get_node(cachep, nodeid);
        /*n = node 의 번호값을 가져옴 */
        BUG_ON(!n);
        /*BUG_ON = debug 할때만 사용 */
retry:
        check_irq_off();
        spin_lock(&n->list_lock);
        entry = n->slabs_partial.next;
        if (entry == &n->slabs_partial) {
                n->free_touched = 1;
                entry = n->slabs_free.next;
                if (entry == &n->slabs_free)
                        goto must_grow;
        }
        /*offset 으로 시작 값 가져옴 */
        /*entry - offset 페이지 값 셋팅 */
        page = list_entry(entry, struct page, lru);
        check_spinlock_acquired_node(cachep, nodeid);

        STATS_INC_NODEALLOCS(cachep);
        STATS_INC_ACTIVE(cachep);
        STATS_SET_HIGH(cachep);

        BUG_ON(page->active == cachep->num);

        /*retrun objp*/
        obj = slab_get_obj(cachep, page, nodeid);
        /*할당완료 했기 때문에 프리한 오브젝트 하나 지움 */
        n->free_objects--;
        /* move slabp to correct slabp list: */
        list_del(&page->lru);

        if (page->active == cachep->num)
                list_add(&page->lru, &n->slabs_full);
        else
                list_add(&page->lru, &n->slabs_partial);

        spin_unlock(&n->list_lock);
        goto done;

must_grow:
        spin_unlock(&n->list_lock);
        x = cache_grow(cachep, gfp_exact_node(flags), nodeid, NULL);
        if (x)
                goto retry;

        return fallback_alloc(cachep, flags);

done:
```

-security_task_create

```
int security_task_create(unsigned long clone_flags)
{
        /* clone_flags = SIGCHLD
        return security_ops->task_create(clone_flags);*/
        return call_int_hook(task_create, 0, clone_flags);
        /*task ops retrun*/
}
```

결국 do_fork에서 리턴되는건 자식 프로세스의 pid 값이며,
그 pid 값을 가진 프로세스 정보는 부모프로세스에서 복사된 정보로 차있다.