TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/4/9
수업일수	33 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

(3) 프로세스와 쓰레드의 생성과 수행

프로세스는 fork(), vfork()를 통해 생성된다.

* fork()와 vfork()의 차이점

exec()함수를 실행하게 될 때의 문제점은 기존 메모리가 날라가게 되어 exec()이후의 명령들을 실행하지 못한다는 점이다.
그래서 fork()와 exec()을 같이 사용하게 되었는데, 이 때의 문제점은 fork()하여 부모의 메모리를 복사하고 자식 프로세스에 exec()하여 메모리를 덮어쓰게 되어 쓸데없이 메모리를 낭비하게 된다. 이를 보완하기 위해 vfork()를 만든 것이다. fork()하고 exec()하여 가상 메모리 레이아웃을 제외한 정보들, 프로세스 구성 정보만 복사하고 싶을 때 vfork()를 사용한다. 사용법은 fork()와 같다.

-쓰레드의 생성 (p55)

~소스코드

```
#define _GNU_SOURCE
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sched.h>

int g=2;

int sub_func(void *arg)
{
        g++;
        printf("PID(%d) : Child g=%d₩n", getpid(), g);
        sleep(2);

        return 0;
}

int main(void)
{
        int pid;
```

```
        int child_stack[4096];
        int l =3;
        printf("PID(%d) : Parent g=%d, l=%d₩n", getpid(), g, l);
        clone(sub_func, (void *)(child_stack+4095), CLONE_VM | CLONE_THREAD |
CLONE_SIGHAND, NULL);
       /*   첫번째 인자 : 태스크가 sub_func 이라는 함수를 구동 시킬 것이다.
            두번째 인자 : thread 도 sub_func 을 구동시키는 주체여서 스택공간이 필요. 물리메모리의
                          최소단위가 4K 이므로 4096byte 를 할당해 준다. 4K 할당받으면 성능상으로
                          좋음.
            세번째 인자 : 옵션 */
        sleep(1);
        printf("PID(%d) : Parent g = %d, l=%d₩n", getpid(), g, l);
        return 0;
}
```

~결과
PID(4082) : Parent g=2, l=3
PID(4082) : Child g=3
PID(4082) : Parent g = 3, l=3

쓰레드는 메모리를 공유하여 전역변수 g 가 sub_func 를 수행하는 쓰레드에서 3으로 증가했을 때 이가 갱신되어 쓰레드 리더에서도 g 값이 갱신된 3이 나오게 된다.
쓰레드를 생성하면 자식쓰레드와 부모쓰레드는 서로 같은 주소공간을 공유한다.
쓰레드가 아닌 프로세스였다면 프로세스는 pid 값이 자식 프로세스에서는 달랐을 것 이고, 메모리를 공유하지 못하니 값이 갱신되지 못했을 것이다.

➔ 새로운 쓰레드를 생성하면 생성된 쓰레드(자식 쓰레드)와 생성한 쓰레드(부모 쓰레드)는 서로 같은 주소공간을 공유한다. 이 때 이 주소 공간이 critical section 이 되고 이를 막기 위해 mutex 와 semaphore, spin lock 을 걸어준다.

clone()

: clone 은 옵션을 주는 것에 따라 쓰레드가 될 수 있고 프로세스가 될 수 있다.

-flags
CLONE_VM : 가상 메모리 사용
CLONE_THREAD : 쓰레드 생성
CLONE_SIGHAND : 시그널 처리
CLONE_CHILD, CLONE_CLEARID, CLONE_SETTID : 태스크 생성

(4) 리눅스의 태스크 모델

리눅스에서는 프로세스와 리눅스를 관리하기 위해 각 프로세스 또는 쓰레드마다 task_struct 라는 자료구조를 생성한다. task_struct 가 있으면 프로세스와 관련된 정보를 알 수 있다.

-태스크 생성과 관계된 함수의 흐름
user app 에서 생성된 프로세스 또는 쓰레드는 C Library 가 돌아가고 어셈블리로 선언된 int 0x80(128번)을 호출하는데 이는 system call 이다.

User App	Library	System call		실행할 때	커널 내부에서 관리
fork() vfork() clone() pthread_create()	fork() vfork() clone() pthread_create()	clone() vfork() clone() clone()	sys_clone() sys_vfork() sys_fork()	do_fork()	kernel_thread()

커널 내부에서는 fork(), vfork() clone(), pthread_create()모두 kernel_thread 로 관리되지만 이렇게 분리해 놓은 이유는 옵션의 차이 때문이다.
옵션이 많이 때문에 분리해 놓음.

-do_fork()내부
새로 생성되는 태스크를 위한 이름표 - pid, tgid
태어난 시간
부모님 이름 - ppid
소지품 - 메모리 레이아웃(stack_start, stack_size), 전역변수
들의 정보들로 구성되어 있다.

```c
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
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)
{
        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.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if ((clone_flags & CSIGNAL) != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }

        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)) {
                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;
}
```

(do_fork 내부)

tgid 와 pid 가 같으면 프로세스 tgid 는 같고 pid 는 다르면 같은 프로세스의 쓰레드들, tgid 와 pid 모두 다르면 다른 프로세스

61p

```c
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<linux/unistd.h>

int main(void)
```

```
{

        int pid;
        printf("before fork ₩n₩n");
        if((pid = fork()) < 0)
        {
                printf("fork error₩n");
                exit(-2);
        }
        else if(pid==0)
        {
                printf("TGID(%d), PID(%d) : Child₩n",getpid(), syscall(__NR_gettid));
        }
        else
        {

                printf("TGID(%d), PID(%d) : Parent₩n",getpid(), syscall(__NR_gettid));
                sleep(2);
        }
        printf("after fork₩n₩n");

        return 0;
}
```

```
before fork

TGID(5555), PID(5555) : Parent
TGID(5556), PID(5556) : Child
after fork

after fork
```

P62

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<linux/unistd.h>

int main(void)
{

        int pid;
        printf("before vfork ₩n₩n");
        if((pid = vfork()) < 0)
        {
                printf("fork error₩n");
                exit(-2);
        }
        else if(pid==0)
        {
                printf("TGID(%d), PID(%d) : Child₩n",getpid(), syscall(__NR_gettid));
                _exit(0);
        }
        else
        {

                printf("TGID(%d), PID(%d) : Parent₩n",getpid(), syscall(__NR_gettid));
        }
        printf("after vfork₩n₩n");

        return 0;
}
```

```
before vfork
```

```
TGID(5583), PID(5583) : Child
TGID(5582), PID(5582) : Parent
after vfork
```

P63

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <pthread.h>


void *t_function(void* data)
{

    int id;
    int i=0;
    pthread_t t_id;
    id = *((int*)data);
    printf("TGID(%d), PID(%lu), pthread_self(%ld): Child₩n", getpid(), syscall(__NR_gettid),
pthread_self());
    sleep(2);
}

int main(void)
{

    int pid, status;
    int a=1;
```

```
    int b=2;

    pthread_t p_thread[2];
    printf("before pthread create₩n₩n");
    if((pid=pthread_create(&p_thread[0], NULL, t_function, (void*)&a))<0)
    {
        perror("thread create error: ");
        exit(1);
    }

    if((pid=pthread_create(&p_thread[1], NULL, t_function, (void*)&b))<0)
    {
        perror("thread create error: ");
        exit(2);
    }

    pthread_join(p_thread[0], (void**)&status);
    printf("pthread_join(%d)₩n", status);
    pthread_join(p_thread[1], (void**)&status);
    printf("pthread_join(%d)₩n", status);
    printf("TGID(%d), PID(%lu): Parent₩n", getpid(), syscall(__NR_gettid));

    return 0;
}
```

~결과

```
xeno@xeno-270E5G-270E5U:~/Downloads$ gcc 63p.c -lpthread
xeno@xeno-270E5G-270E5U:~/Downloads$ ./a.out
before pthread create

TGID(4400), PID(4402), pthread_self(140683338651392): Child
TGID(4400), PID(4401), pthread_self(140683347044096): Child
```

```
pthread_join(0)
pthread_join(0)
TGID(4400), PID(4400): Parent
```

p64

~소스코드

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <sched.h>



int sub_func(void* arg)
{
    printf("TGID(%d), PID(%lu): Child₩n", getpid(), syscall(__NR_gettid));
    sleep(2);
    return 0;

}

int main(void)
{
    int pid;
    int child_a_stack[4096], child_b_stack[4096];
```

```
    printf("before clone ₩n₩n");
    printf("TGID(%d), PID(%lu): Parent₩n", getpid(), syscall(__NR_gettid));

     clone(sub_func,  (void*)(child_a_stack+4095),  CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID,
NULL);
     clone(sub_func,  (void*)(child_a_stack+4095),  CLONE_VM|CLONE_THREAD|CLONE_SIGHAND,
NULL);

    sleep(1);
    printf("after clone₩n₩n");
    return 0;


}
```

~결과

```
before clone

TGID(4421), PID(4421): Parent
TGID(4421), PID(4423): Child
TGID(4422), PID(4422): Child
after clone
```

current 라는 매크로는 커널 내부에 정의되어 있는 매크로로써 현재 태스크의 task_struct 구조체를 가리킬 수 있게 해준다. task_tigd_vnr()은 해당 task_struct 구조체의 tgid 필드를 리턴한다.

```
/*
 * how to get the current stack pointer in C
 */
register unsigned long current_stack_pointer asm ("sp");

/*
 * how to get the thread information struct from C
 */
static inline struct thread_info *current_thread_info(void) __attribute_const__;

static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)
                (current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

(5) 태스크 문맥

:태스크과 관련된 모든 정보들, 파일을 열었을 때 리턴받는 파일 디스크립터들, 스케줄링을 위한 우선순위, CPU 사용량, 태스크의 가족관계, 태스크에 전달된 시그널, 태스크가 사용하고 있는 자원 등의 정보들.

thread_union 에서 그림 3.14 태스크의 문맥(p66) 에 나와있는 것들을 모두 관리한다.

thread_info 구조체인 thread_info 가 선언되어 있다.

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

1. hardware context - thread_info

```
/*
 * low level task data that entry.S needs immediate access to.
 * __switch_to() assumes cpu_context follows immediately after cpu_domain.
 */
struct thread_info {
        unsigned long           flags;          /* low level flags */
        int                     preempt_count;  /* 0 => preemptable, <0 => bug */
        mm_segment_t            addr_limit;     /* address limit */
        struct task_struct      *task;          /* main task structure */
        __u32                   cpu;            /* cpu */
        __u32                   cpu_domain;     /* cpu domain */
        struct cpu_context_save cpu_context;    /* cpu context */
        __u32                   syscall;        /* syscall number */
        __u8                    used_cp[16];    /* thread used copro */
        unsigned long           tp_value[2];    /* TLS registers */
#ifdef CONFIG_CRUNCH
        struct crunch_state     crunchstate;
#endif
        union fp_state          fpstate __attribute__((aligned(8)));
        union vfp_state         vfpstate;
#ifdef CONFIG_ARM_THUMBEE
        unsigned long           thumbee_state;  /* ThumbEE Handler Base register */
#endif
};
```

2. system context

files_struct

```
/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
        atomic_t count;
        bool resize_in_progress;
        wait_queue_head_t resize_wait;

        struct fdtable __rcu *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        spinlock_t file_lock ____cacheline_aligned_in_smp;
        int next_fd;
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        unsigned long full_fds_bits_init[1];
        struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

fdtable

```
/*
 * The default fd array needs to be at least BITS_PER_LONG,
 * as this is the granularity returned by copy_fdset().
 */
#define NR_OPEN_DEFAULT BITS_PER_LONG

struct fdtable {
        unsigned int max_fds;
        struct file __rcu **fd;      /* current fd array */
        unsigned long *close_on_exec;
        unsigned long *open_fds;
        unsigned long *full_fds_bits;
        struct rcu_head rcu;
};
```

**fd 의 file

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

3. memory context

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;

        /*
         * Largest free memory gap in bytes to the left of this VMA.
         * Either between this VMA and vma->vm_prev, or between one of the
         * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
         * get_unmapped_area find a free area of the right size.
         */
        unsigned long rb_subtree_gap;

        /* Second cache line starts here. */

        struct mm_struct *vm_mm;        /* The address space we belong to. */
        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */

        /*
         * For areas with an address space and backing store,
         * linkage into the address_space->i_mmap interval tree.
         */
        struct {
                struct rb_node rb;
                unsigned long rb_subtree_last;
        } shared;

        /*
         * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
         * list, after a COW of one of the file pages.  A MAP_SHARED vma
         * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
         * or brk vma (with NULL file) can only be in an anon_vma list.
         */
        struct list_head anon_vma_chain; /* Serialized by mmap_sem &
                                          * page_table_lock */
        struct anon_vma *anon_vma;      /* Serialized by page_table_lock */

        /* Function pointers to deal with this struct. */
        const struct vm_operations_struct *vm_ops;

        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;         /* was vm_pte (shared mem) */

#ifndef CONFIG_MMU
        struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
        struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
};
```

-task identification

: 태스크를 인식하기 위한 변수들이다. pid, tgid, pid 를 통해 해당 태스크의 task_struct 를 빠르게 찾기 위해 커널이 유지하고 있는 해쉬 관련 필드 등의 변수가 있다. audit-context 구조체를 통해 이 태스크에 대한 사용자 접근 권한을 제어할 때 이용되는 uid(사용자 ID), euid(유효 사용자 ID) 등이 있다.

-state

: 태스크를 생성하고 소멸할 때 까지 관리하기 위한 state 변수 존재.

TASK_RUNNIG(0) - 현재 태스크가 run queue 에 있다.

TASK_OMTERRUPTIBLE(1) - 인터럽트 수신 가능

TASK_UNINTERRUPTIBLE(2) - 인터럽트 수신 불가능

TASK_STOPPED(4) - 일시적으로 정지(ctrl+z)

TASK_TRACED(8) - 디버깅 할 때 걸리는 상태

EXIT_DEAD(16) - 리턴 0, exit(0),시그널 맞아 죽었을 때

EXIT_ZOMBIE(32) - 좀비 프로세스 상태

-task relationship

태스크는 생성하면서 가족관계를 갖는데 현재 부모 태스크의 task_stuct 를 가리키는 real_parent 와 현재 부모 태스크의 task_struct 구조체를 가리키는 parent 필드가 존재. 또한 자식과 형제를 리스트로 연결한 뒤 그 리스트의 헤드를 각각 children, sibling 필드에 저장했다. 형제관계에 있는 .next와 .prev는 list_head 라는 구조체 안에 이중 연결리스트로 연결되어 있다.

```
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->real_parent->pid)
         */
        struct task_struct __rcu *real_parent; /* real parent process */
        struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
        /*
         * children/sibling forms the list of my natural children
         */
        struct list_head children;      /* list of my children */
        struct list_head sibling;       /* linkage in my parent's children list */
        struct task_struct *group_leader;       /* threadgroup leader */
```

```
struct list_head {
        struct list_head *next, *prev;
};
```

-scheduling information

task_struct 에서 스케줄링과 관련된 변수 : prio, policy, cpus_allowed, time_slice, rt_priority 등이 있다.

-signal information

task_struct 에서 시그널과 관련된 변수 : signal, sighand, blocked, pending 등이 있다.

-memory information

태스크는 자신의 명령어와 데이터를 텍스트, 데이터, 스택, 힙 공간에 저장. 이 공간에 대한 위치와 크기, 접근제어 정보 등을 관리하는 변수들이 존재한다. 가상 주소를 물리주소로 변환하기 위한 페이지 디렉터리와 페이지 테이블 등의 주소 변환 정보도 task_struct 에 존재. 이 정보들은 mm_struct 에서 접근 가능하다.

```
/*
 * A region containing a mapping of a non-memory backed file under NOMMU
 * conditions.  These are held in a global tree and are pinned by the VMAs that
 * map parts of them.
 */
struct vm_region {
        struct rb_node  vm_rb;          /* link in global region tree */
        vm_flags_t      vm_flags;       /* VMA vm_flags */
        unsigned long   vm_start;       /* start address of region */
        unsigned long   vm_end;         /* region initialised to here */
        unsigned long   vm_top;         /* region allocated to here */
        unsigned long   vm_pgoff;       /* the offset in vm_file corresponding to vm_start */
        struct file     *vm_file;       /* the backing file or NULL */

        int             vm_usage;       /* region usage count (access under nommu_region_sem) *
/
        bool            vm_icache_flushed : 1; /* true if the icache has been flushed for
                                                * this region */
};
```

-file information

태스크가 오픈한 파일들은 task_stuct 에서 files_struct 구조체 형태인 files 라는 이름의 변수로 접근 가능. 루트 디렉터리의 inode 와 현제 디렉터리의 inode 는 fs_struct 구조체 형태인 fs 라는 변수로 접근 가능

-CPU context

쓰레드 구조는 context switching 할 때 태스크가 현재 어디까지 실행되었는지 기억해 놓는 공간

-time information

태스크의 시간 정보를 위한 변수. start_time, real_time 등이 있다.

-format

현재에는 바뀌어서 task_struct 안의 personality 변수 밖에 남지 않았다.

(6) 상태전이(State Transition)와 실행 수준 변화

태스크는 동작하며 CPU 이외의 자원을 요청하기도 하는데 당장 제공할 수 없는 자원을 요청하면 태스크를 잠시 대기하도록 만든 뒤 요청했던 자원이 사용 가능해지면 수행시켜 줌으로써 높은 시스템 활용률을 제공하려 하고 상태전이라는 특징을 가진다.

CPU 는 한번에 하나의 동작밖에 수행하지 못하고 n 개의 CPU 는 최대 n 개 까지의 태스크가 실행 상태에 있을 수 있다.

실행 상태에 있는 태스크들은 발생하는 사건에 따라 상태전이 하는 경우

1. 태스크가 할 일을 끝내고 exit()나 kill 이면 TASK_DEAD 상태로 전이. 정확히 말하면 부모 태스크에게 알려주기 위해 유지되고있는 TASK_DEAD(EXIT_ZOMBIE)상태가 됨.

   만일 고아 프로세스일 경우에는 init 태스크가 wait()함수를 호출할 때 최종 소멸된다.

2. 실행(TASK_RUNNING(running))상태에서 CPU 가 할당된 시간을 다 사용하거나 높은 우선순위의 태스크로 인해 TASK_RUNNING(ready)로 전환되는 경우. CPU 를 공정하게 사용될 수 있게 CFS 기법을 사용한다.

3. SIGSTOP, SIGSTP,SIGTTIN, SIGTTOU 등의 시그널을 받은 태스크는 TASK_STOPPED 상태로 전이되며, SIGCONT 시그널을 받아 다시 TASK_RUNNING(ready)상태로 전환된다.

   디버거의 ptrace() 호출에 의해 디버깅되고 있는 태스크는 시그널을 받는 경우 TASK_TRACED 상태로 전이될 수 있다.

4. 실행 TASK_RUNNING(running)상태에 있던 태스크가 특정한 사건을 기억해야할 때 대기상태(TASK_INTTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_KILLABLE)로 전이한다.

사용자 수준 실행상태에서 커널 수준 실행상태로 전이할 수 있는 방법

1. 시스템 호출(system call) - 태스크가 시스템 호출을 요청하는 것. system call handler 가 처리함.
2. 인터럽트 발생(하드웨어 인터럽트) - 인터럽트가 발생되면 리눅스 커널에 인터럽트가 걸림. 하드웨어마다 핸들러가 있어 그 핸들러가 처리.

리눅스도 소프트웨어이기 때문에 스택을 필요로 한다. ARM 은 8KB, intel 은 16KB 의 스택을 태스크 별로 할당함. 태스크가 생성되면 리눅스는 task_struct 구조체와 커널스택(thread_union)을 할당하게 된다. (커널스택은 thread_info 구조체를 포함하며 thread_info 는 프로세스 디스크립터라고 부름)

커널이 시스템 호출의 서비스를 완료하거나 인터럽트 처리를 완료하면 커널 수준 실행 상태에서 사용자 수준 실행상태로 전이한다. 이때 리눅스 커널이 하는 중요한 일들

1. 커널이 현재 실행중인 태스크가 시그널을 받았는지 확인하고 받았다면 필요한 경우 시그널 핸들러를 호출.
2. 다시 스케줄링 해야 할 필요가 있다면 (need_resched 플래그가 1 로 set 되어 있는 경우 = = flags=1 이면) 스케줄러 호출.
3. 커널 내에서 연기된 루틴들이 존재하면 이를 수행한다.

   이때 연기된 루틴은 non-block 일 때 들어온 것들을 리스트로 만들어서 처리하는데 논블록킹 방식으로 처리할 것들 + bottom half 가 연기된 루틴이다.

(8) 문맥교환(context switch)

만약 A 라는 태스크에 할당되어 있던 타임 슬라이스가 모두 소진되거나 잠들어야 하는 경우 리눅스 커널은 새로이 수행할 태스크 B 를 선택하여 CPU 자원을 배정해준다. 수행 중이던 태스크의 동작을 멈추고 다른 태스크로 전환하는 과정을 context switch 라고 부른다. context switch 를 할 때에는 현재 수행 중이던 태스크의 위치 정보를 CPU 레지스터(H/W context)에 저장하고 이 H/W context 는 thread_struct 로 관리된다.