Linux Task Management

단국대학교
컴퓨터학과
2009
백승재
ibanez1383@dankook.ac.kr
http://embedded.dankook.ac.kr/~ibanez1383

강의 목표

- Linux의 task 개념 이해
- Task 자료구조 파악
- Linux의 task 관리 방법 파악

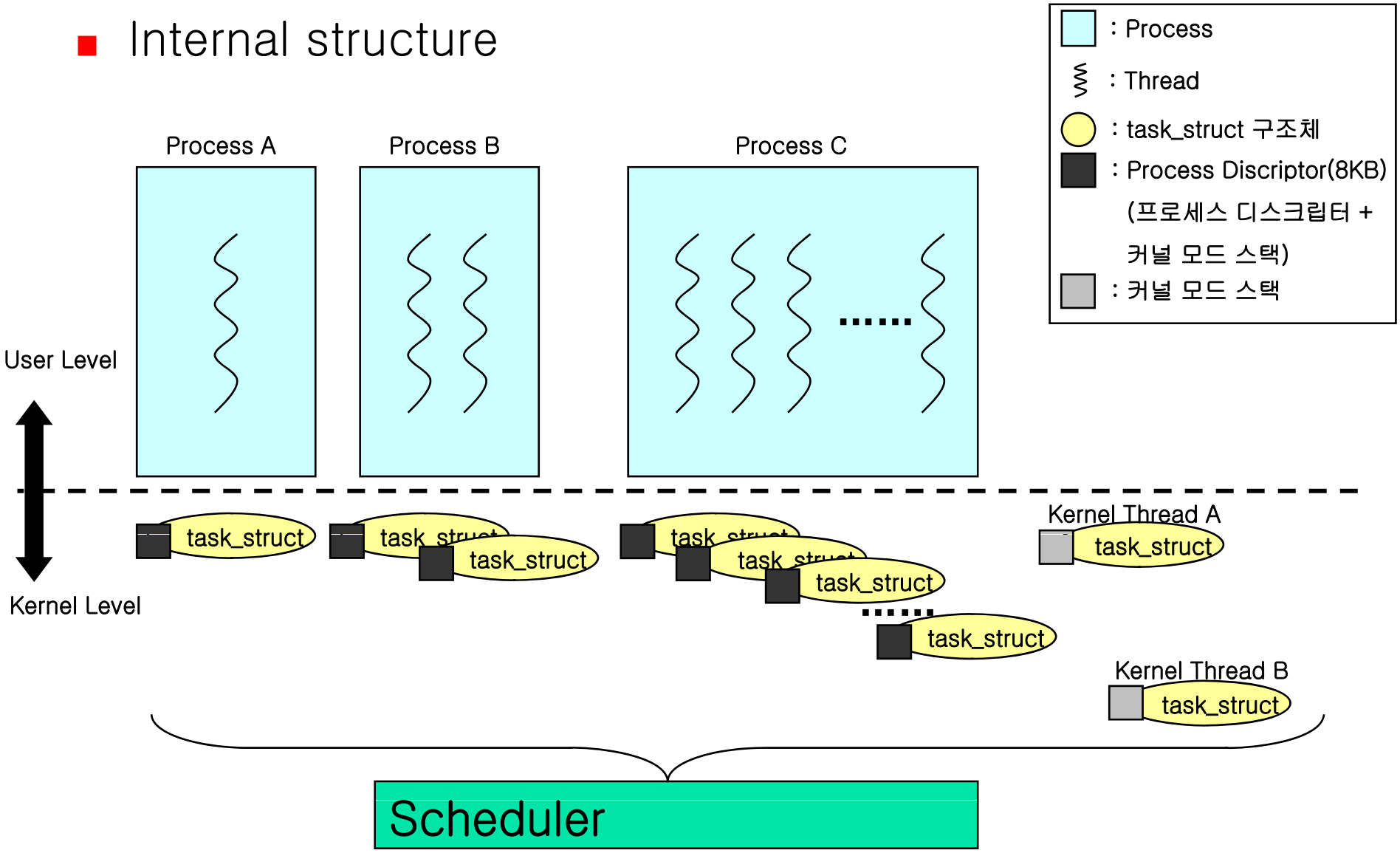
Task, Process and Thread

- Process
  - ✓ 실행 상태에 있는 프로그램의 instance …
  - ✓ 자원 소유권의 단위
- Thread
  - ✓ 디스패칭의 단위, 실행 흐름 …
  - ✓ 수행의 단위

- In Linux source code
  - ✓ Process is Task and Thread is also Task

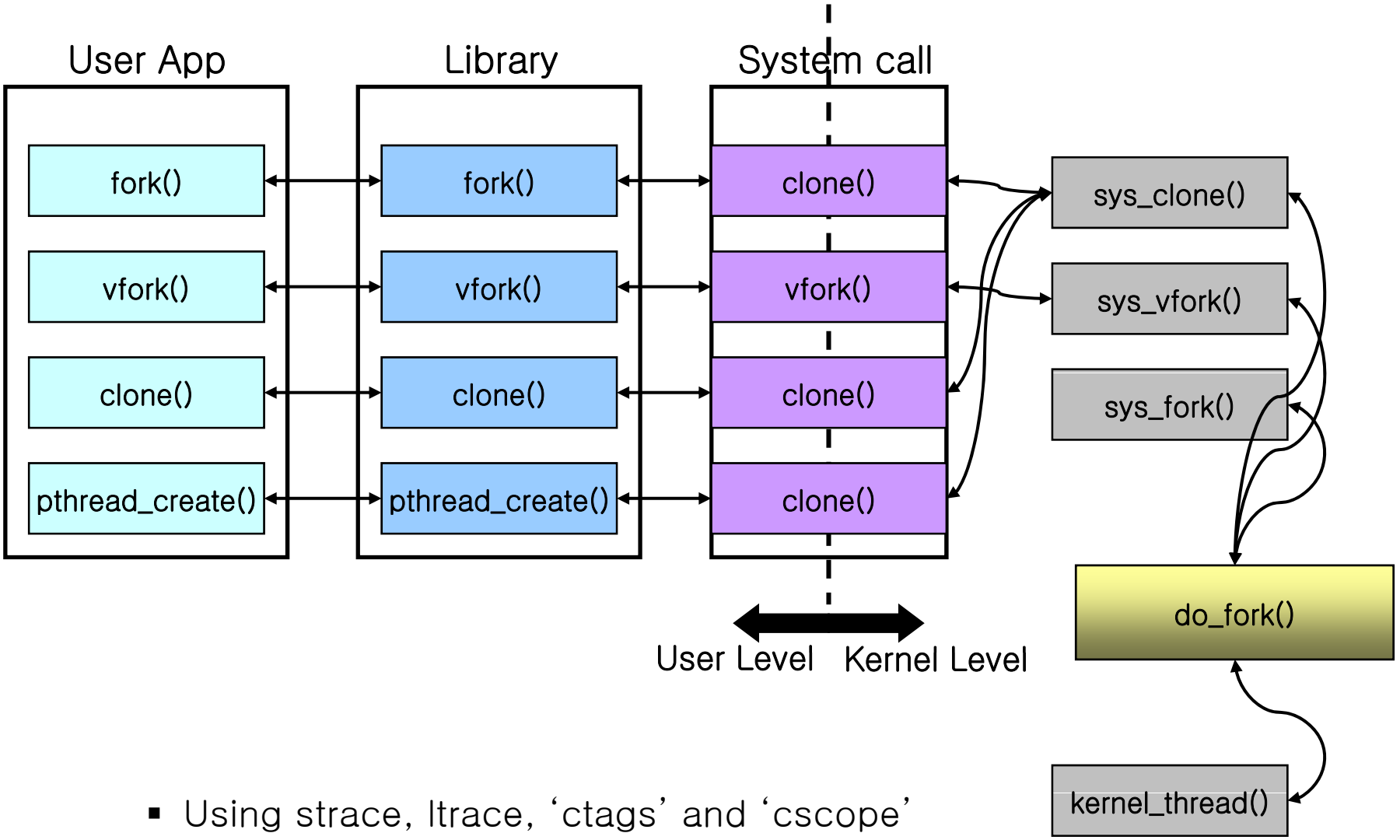
Linux Task Model

- Internal structure



Task 관련 함수 흐름

- Flow controls



- Using strace, ltrace, ‘ctags’ and ‘cscope’

Thread example(1/5)

```
_syscall0(pid_t, gettid);

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("child, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
        }else{
                printf("parent, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
                sleep(2);
        }
        exit(0);
}
```

```
[root@localhost 3_4_process]# ./fork
before fork

child, tgid=(3486), pid=(3486)
parent, tgid=(3485), pid=(3485)
[root@localhost 3_4_process]#
```

Thread example(2/5)

```
_syscall0(pid_t, gettid);

void *t_function(void *data)
{
        int id;         int i=0;        pthread_t t_id;
        id = *((int *)data);
        printf("child, tgid=(%d), pid=(%d), pthread_self()=(%d)₩n", getpid(), gettid(), pthread_self());
        sleep(2);
        return (void *)(id*id);
}
int main()
{
        pthread_t p_thread[2];
        int thr_id;     int status;
        int a = 1;      int b = 2;

        printf("before pthread_create()₩n");

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

        if((thr_id = 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("thread join : %d₩n", status);
        pthread_join(p_thread[1], (void **)&status);
        printf("thread join : %d₩n", status);

        printf("parent, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
        return 0;
}
```

Thread example(3/5)


```
[root@localhost 5_6_thread]# ./pthread
before pthread_create()
child, tgid=(3619), pid=(3620), pthread_self()=(1082132832)
child, tgid=(3619), pid=(3621), pthread_self()=(1090525536)
thread join : 1
thread join : 4
parent, tgid=(3619), pid=(3619)
```

Thread example(4/5)


```
_syscall0(pid_t, gettid);

int main(void)
{
        pid_t pid;

        printf("before vfork₩n");

        if((pid = vfork()) < 0){
                printf("fork error₩n");
                exit(-2);
        }else if (pid == 0){
                printf("child, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
                _exit(0);
        }else{
                printf("parent, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
        }

        exit(0);
}
```

```
[root@localhost 3_4_process]# ./vfork
before vfork
child, tgid=(3515), pid=(3515)
parent, tgid=(3514), pid=(3514)
```

Thread example(5/5)

```
_syscall0(pid_t, gettid);

int             sub_a(void *);

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

                printf("main:Before clone₩n");
                printf("parent, tgid=(%d), pid=(%d)₩n", getpid(), gettid());

                clone (sub_a, (void *)(child_a_stack+4095), CLONE_VM, NULL);
                clone (sub_a, (void *)(child_b_stack+4095), CLONE_VM, NULL);

                exit(0);
}

int sub_a(void *arg)
{
                printf("child, tgid=(%d), pid=(%d)₩n", getpid(), gettid());
                exit(1);
}
```

CLONE_THREAD option 추가 시 같은 tgid를 가짐

```
[root@localhost 5_6_thread]# ./clone
main:Before clone
parent, tgid=(3576), pid=(3576)
[root@localhost 5_6_thread]# child, tgid=(3577), pid=(3577)
child, tgid=(3578), pid=(3578)

[root@localhost 5_6_thread]#
```

Task in Linux

- POSIX interface 기반

- 구현은 버전에 따라 약간씩 다름
  - ✓ Linux 1.X
    - User / kernel mode 지원
    - Thread 생성시 thread의 attribute를 지정 해 줌으로써 thread mode를 설정 할 수 있음
  - ✓ Linux 2.X
    - Kernel mode 만 지원 (sys_clone)
    - LinuxThreads라고 불림
    - 각 thread마다 task 자료구조 할당 (MACH의 thread와는 다름)
    - POSIX incompatible 문제 (pid/tid, signal handling, …)
  - ✓ Linux 2.6
    - NPTL (Native POSIX Thread Library)

Task in Linux

- LinuxThreads vs NPTL

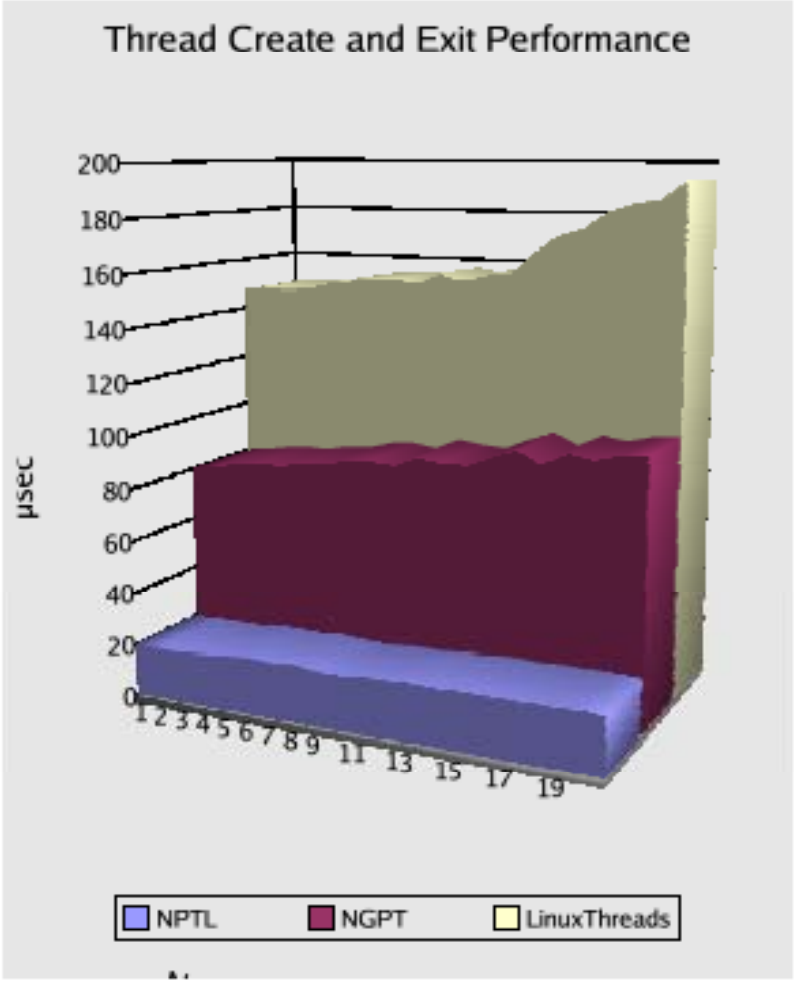


Figure 1: Varying number of Toplevel Threads

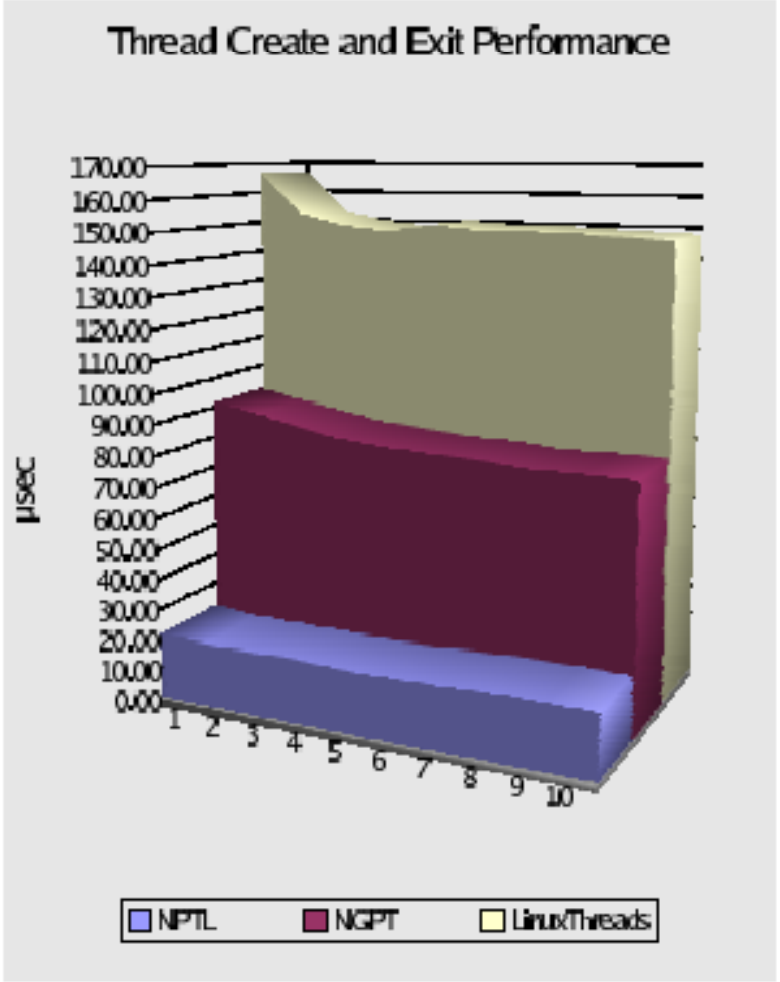


Figure 2: Varying number of Concurrent Children

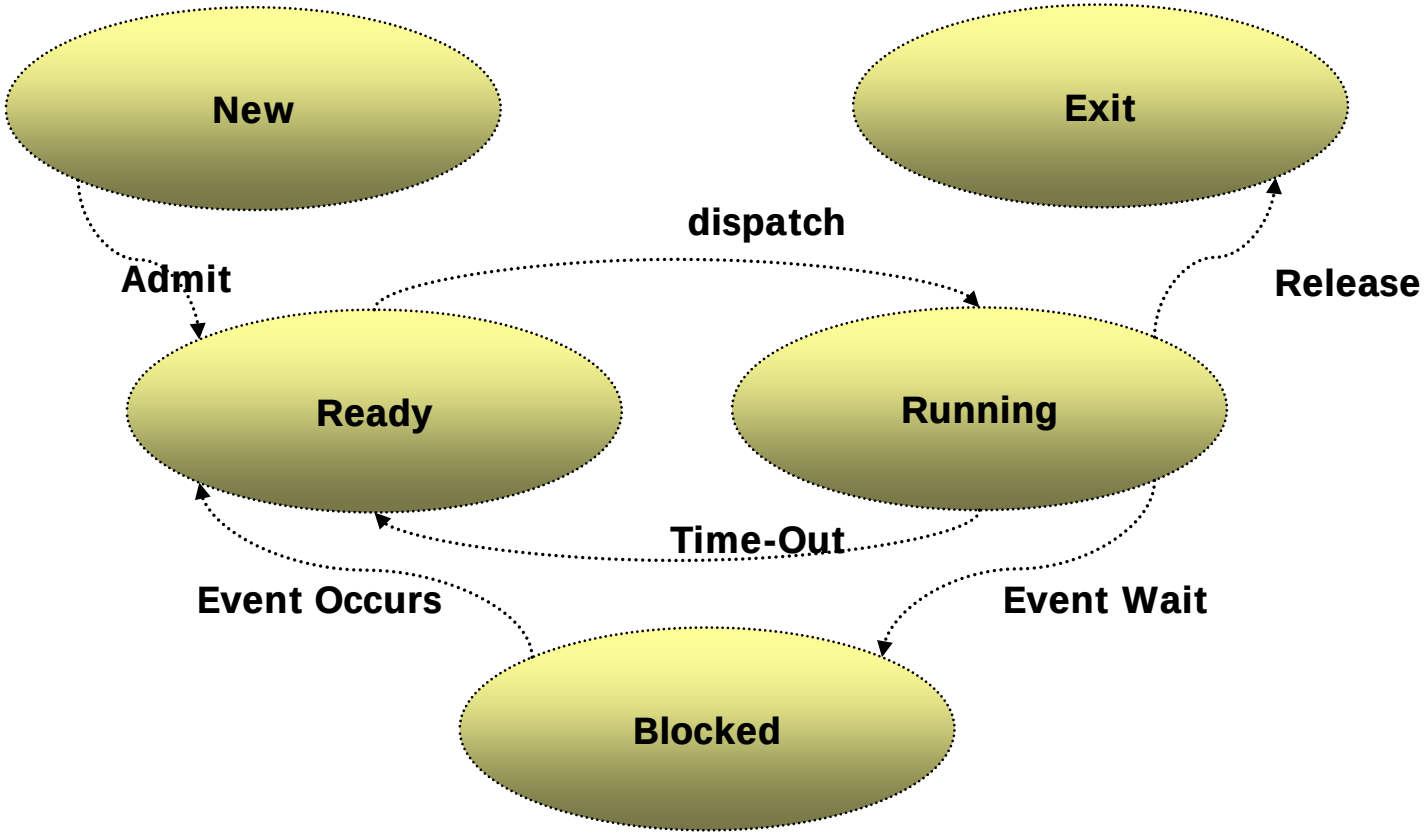
Task, Process and Thread

- Task는
  - ✓ 사용자의 요청을 대리하며 시스템 자원을 두고 서로 경쟁한다
  - ✓ 자신의 메모리 공간(코드, 변수, 스택)과 하드웨어 레지스터를 갖는다
  - ✓ 태스크 계층구조를 갖는다
  - ✓ 상태와 전이를 갖는다.

```
/* test.c */

int             glob = 6;
char            buf[] = "a write to stdout₩n";

int main(void)
{
                int  var;
                pid_t pid;

                var = 88;
                write(STDOUT_FILENO, buf, sizeof(buf)-1);
                printf("before fork₩n");

                if ((pid = fork()) == 0) {      /* child */
                                glob++; var++;
                } else
                                sleep(2);                       /* parent */

                printf("pid = %d, glob = %d, var = %d₩n", getpid(), glob, var);
                exit (0);
}
```

(Source : Adv. programming in the UNIX Env., pgm 8.1)

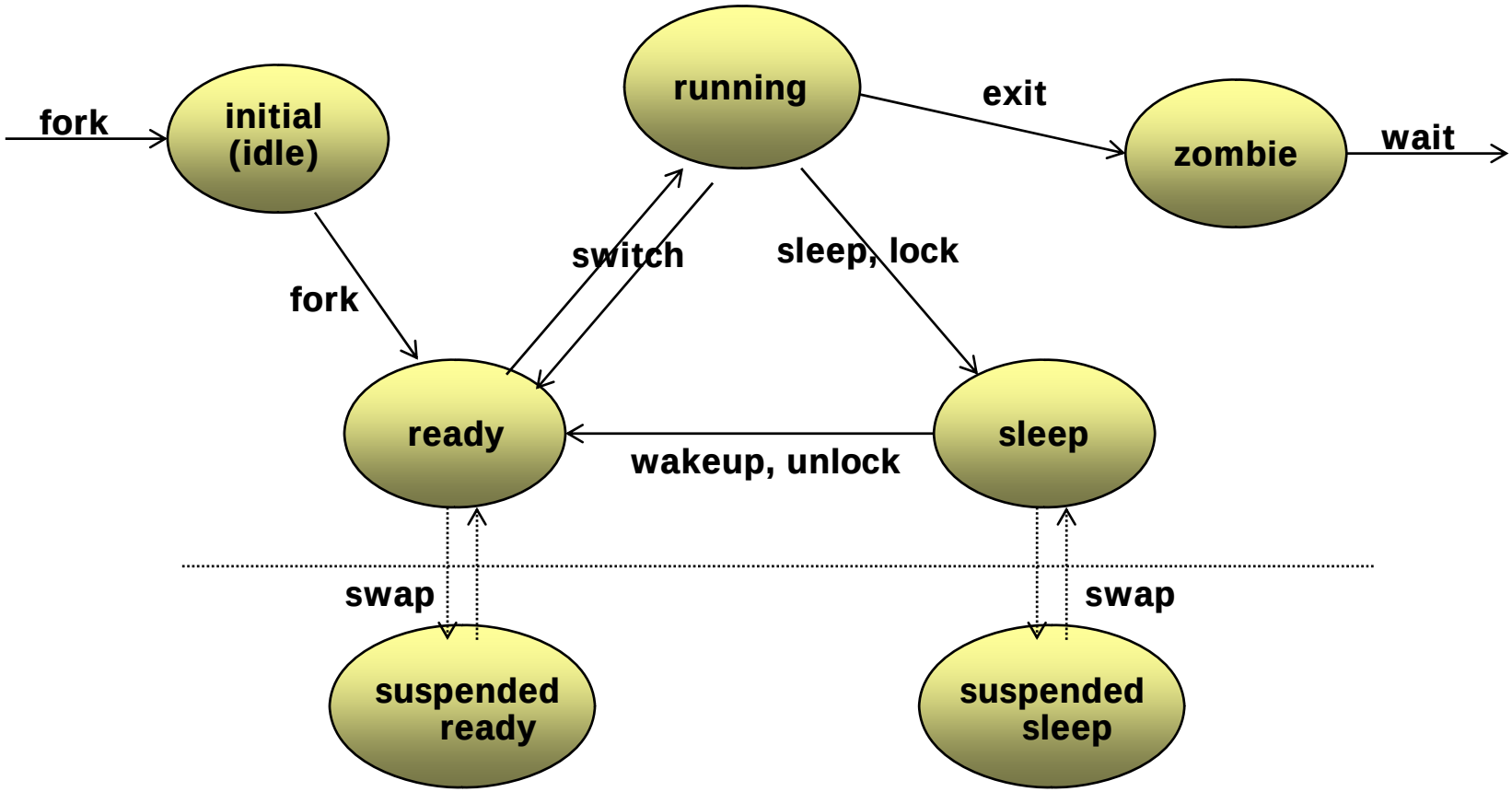
Process State Diagram(1/2)



Five-State Process Model

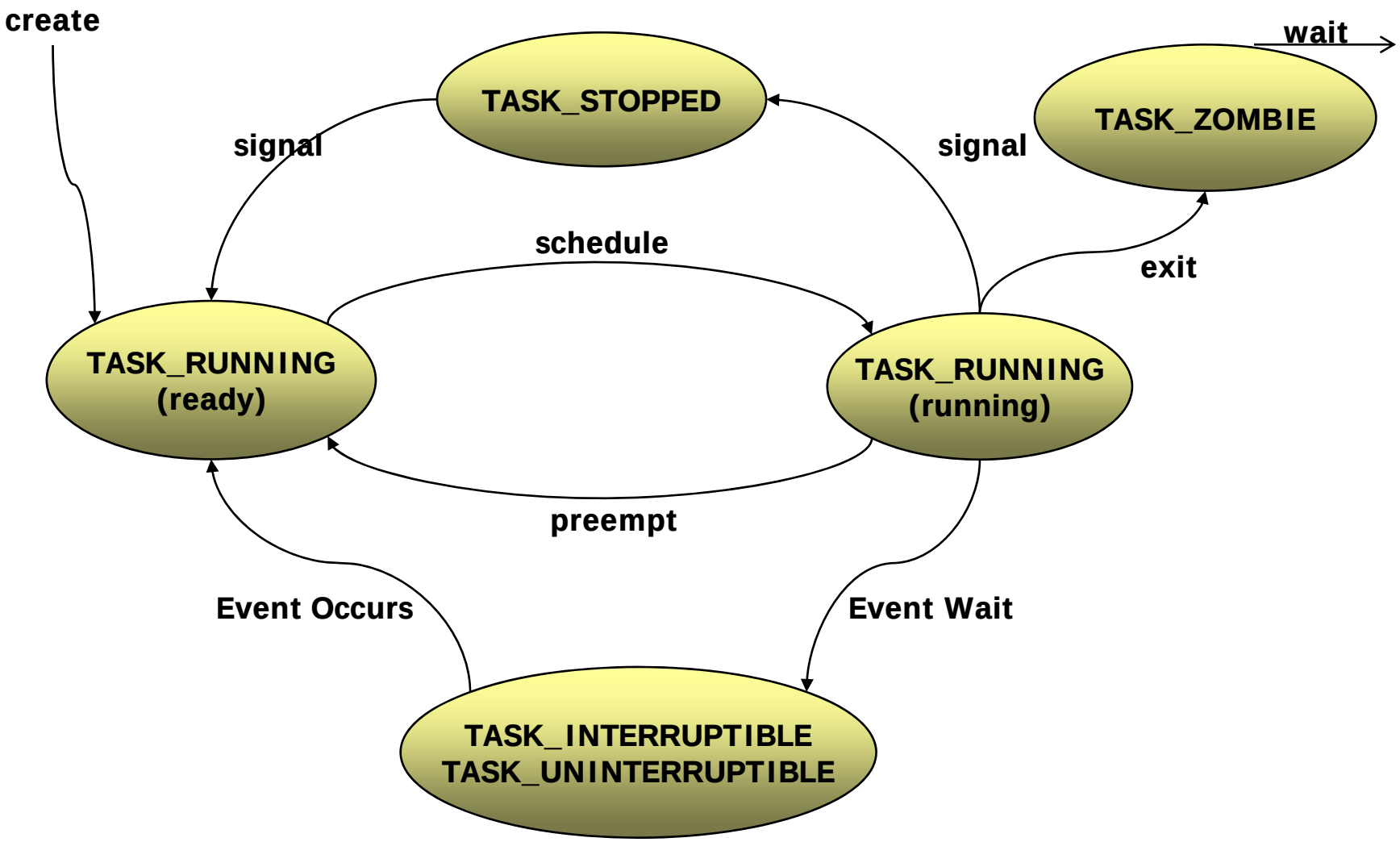
(Source : OS(stallings))

Process State Diagram(2/2)

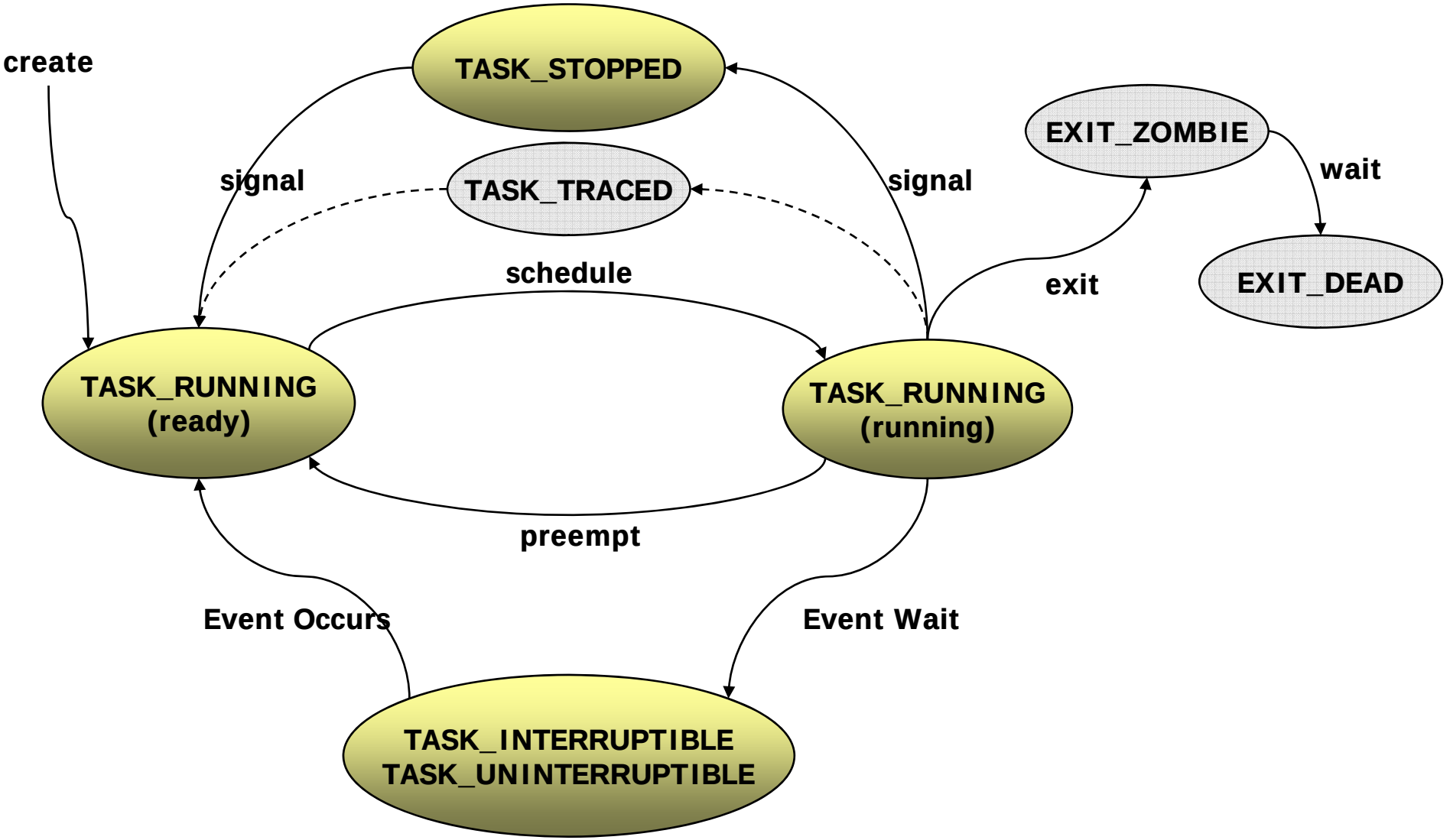


(Source : UNIX Internals)

Task State Diagram of LINUX(2.4)

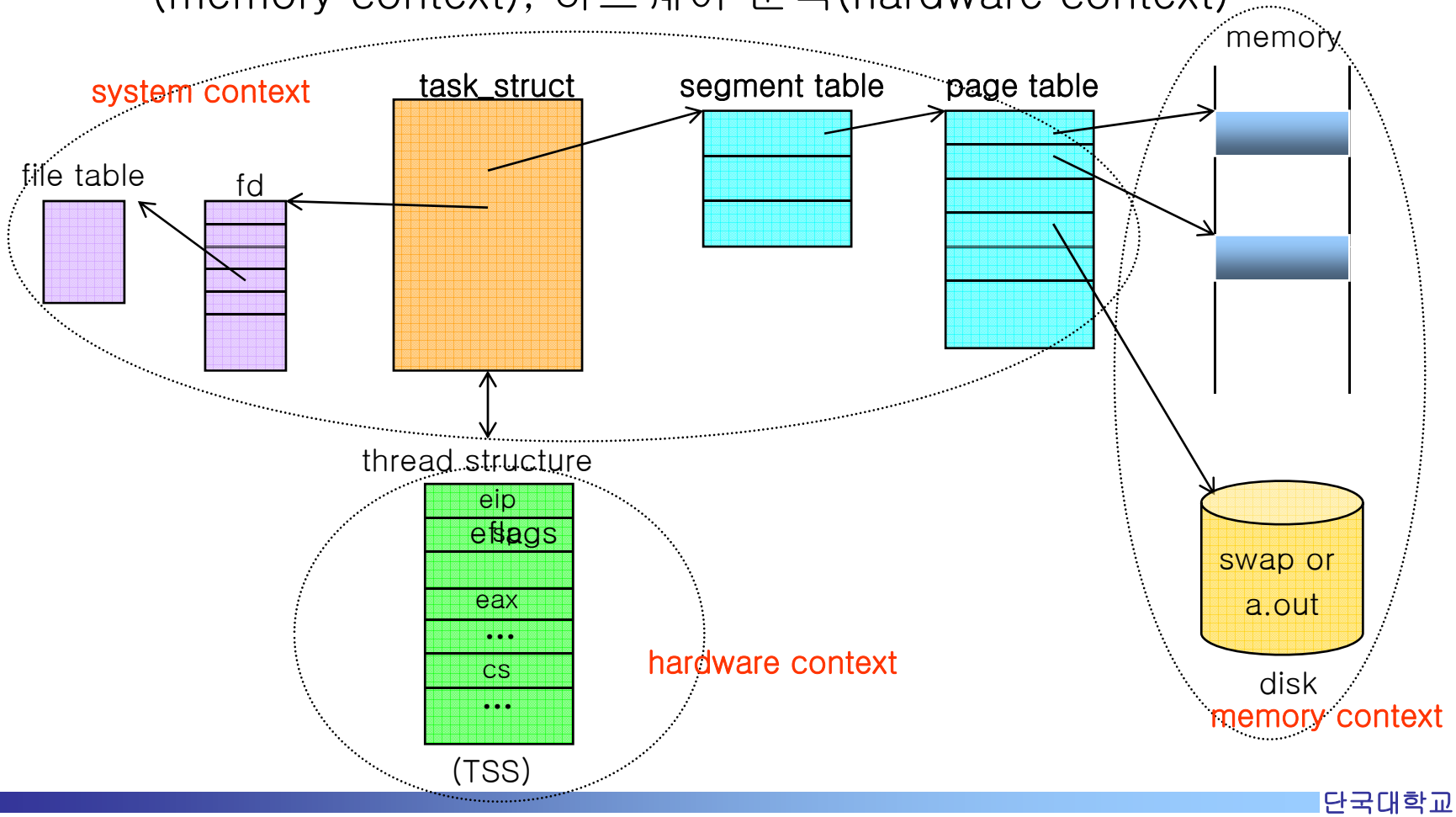


Task State Diagram of LINUX(2.6)



Context

- 문맥 (Context) :
  - ✓ 커널이 관리하는 태스크의 자원과 수행 환경 집합
  - ✓ 3 부분으로 구성 :시스템 문맥 (system context), 메모리 문맥 (memory context), 하드웨어 문맥(hardware context)

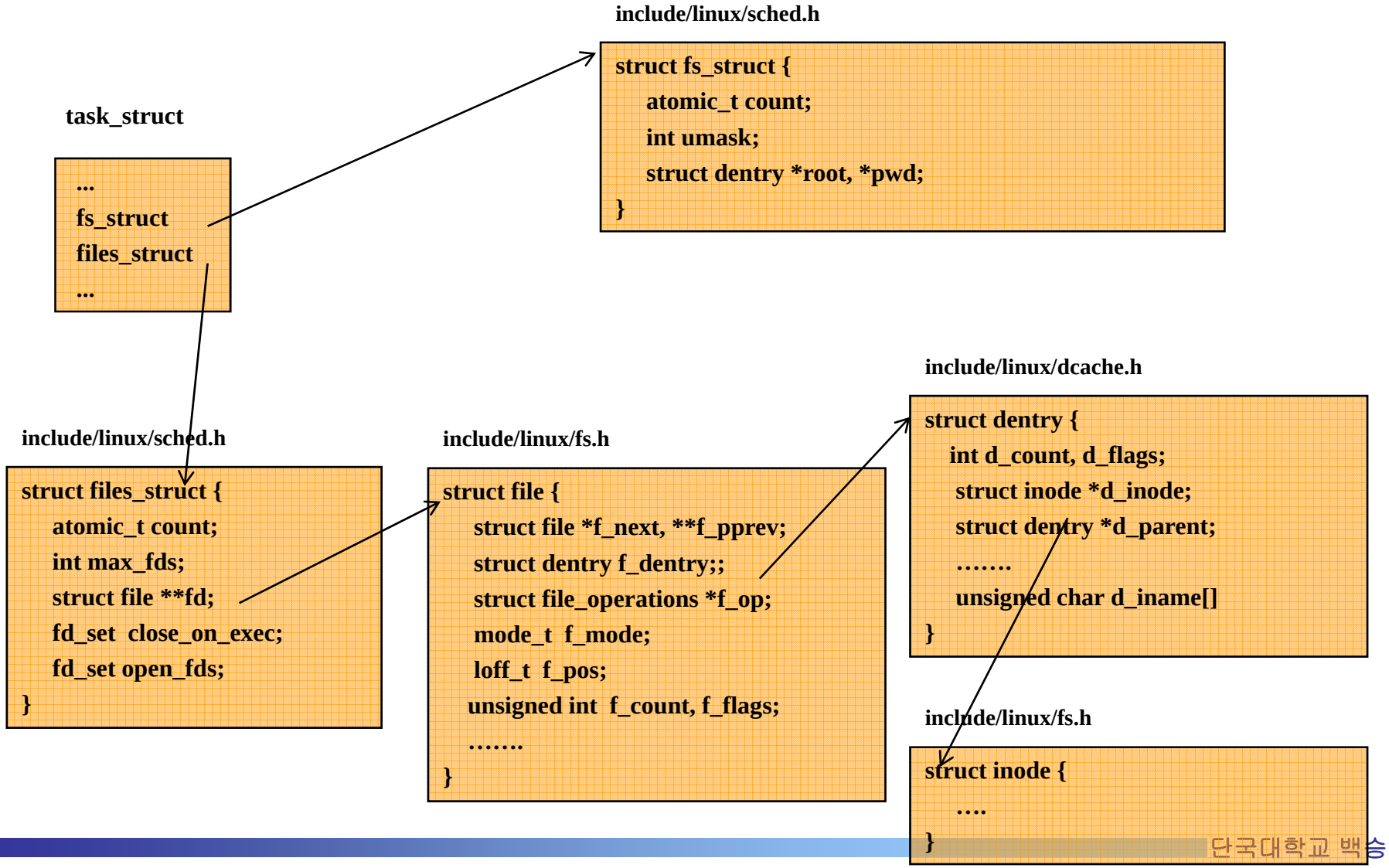


System Context

- 태스크를 관리하는 정보 집합
  - ✓ 태스크 정보: pid, uid, euid, suid, …
  - ✓ 태스크 상태: 실행 상태, READY 상태, 수면 상태, …
  - ✓ 태스크의 가족 관계: p_pptr, p_cptr, next_task, next_run
  - ✓ 스케줄링 정보: policy, priority, counter, rt_priority, need_resched
  - ✓ 태스크의 메모리 정보: 세그먼트, 페이지
  - ✓ 태스크가 접근한 파일 정보: file descriptor
  - ✓ 시그널 정보
  - ✓ 쓰레드 정보
  - ✓ 그 외: 수행 시간 정보, 수행 파일 이름, 등등 (kernel dependent)

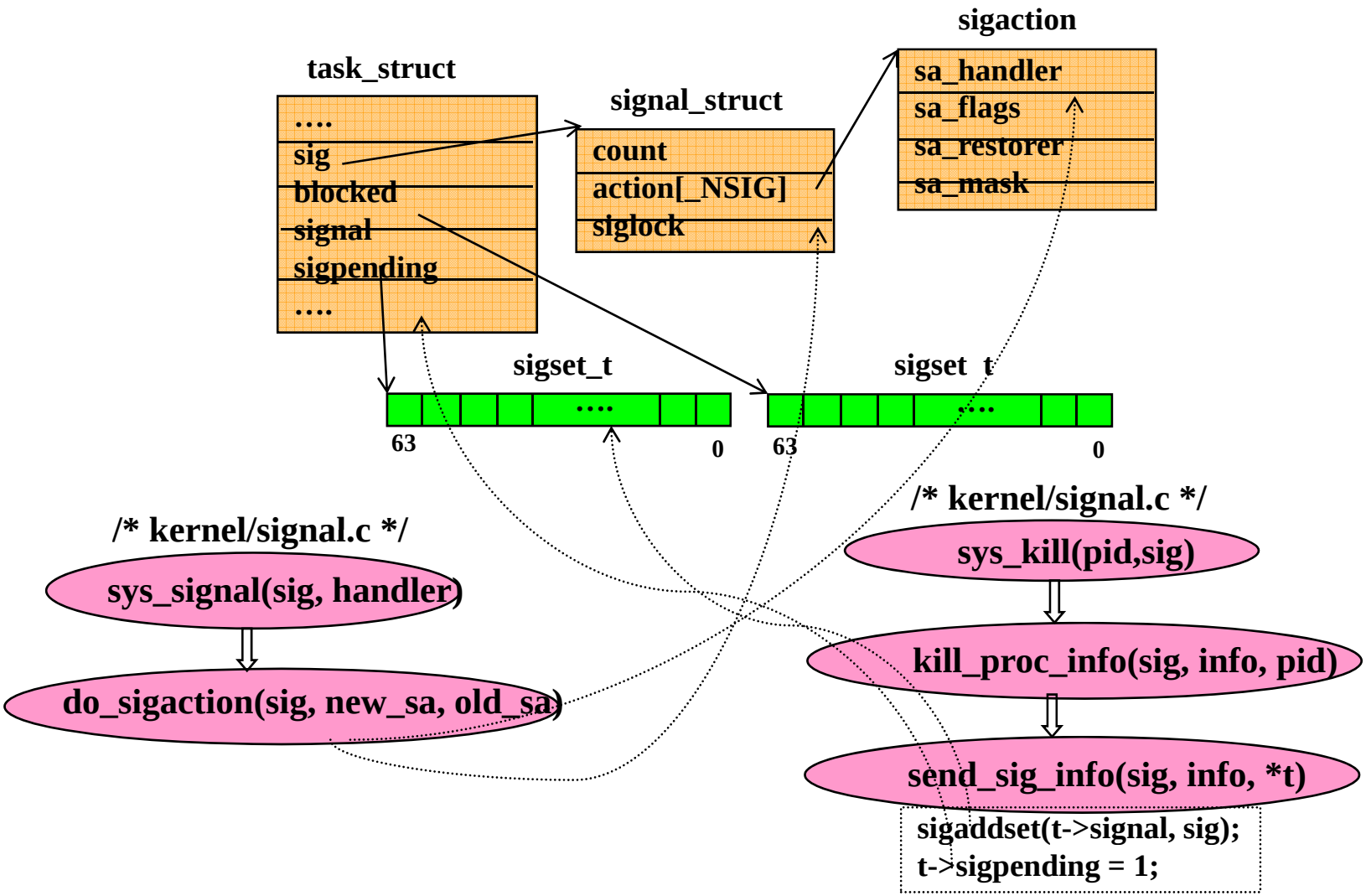
Task와 file

- 파일 관리 자료 구조 : fd, inode



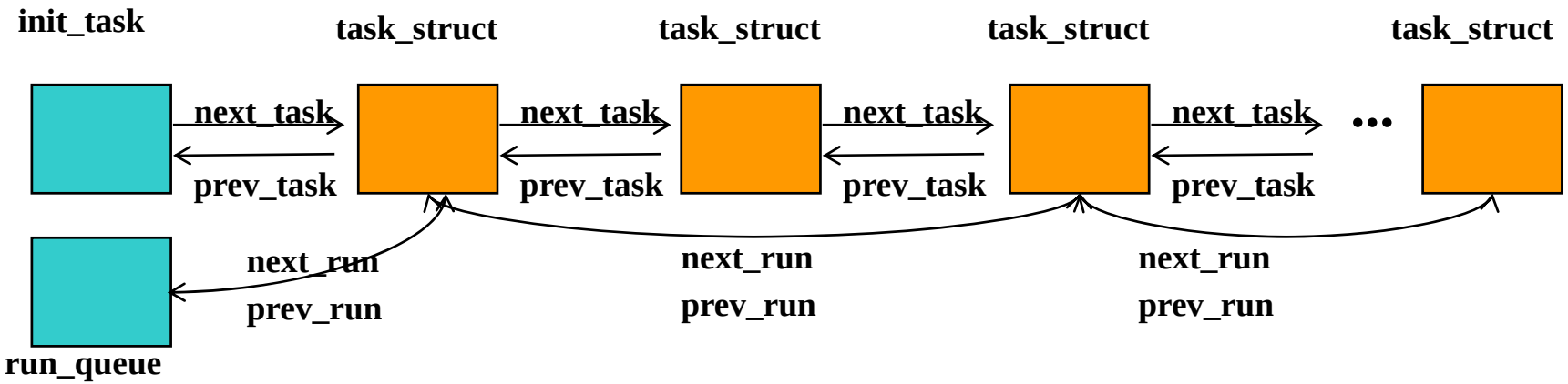
Task와 signal

- 시그널 처리 자료 구조



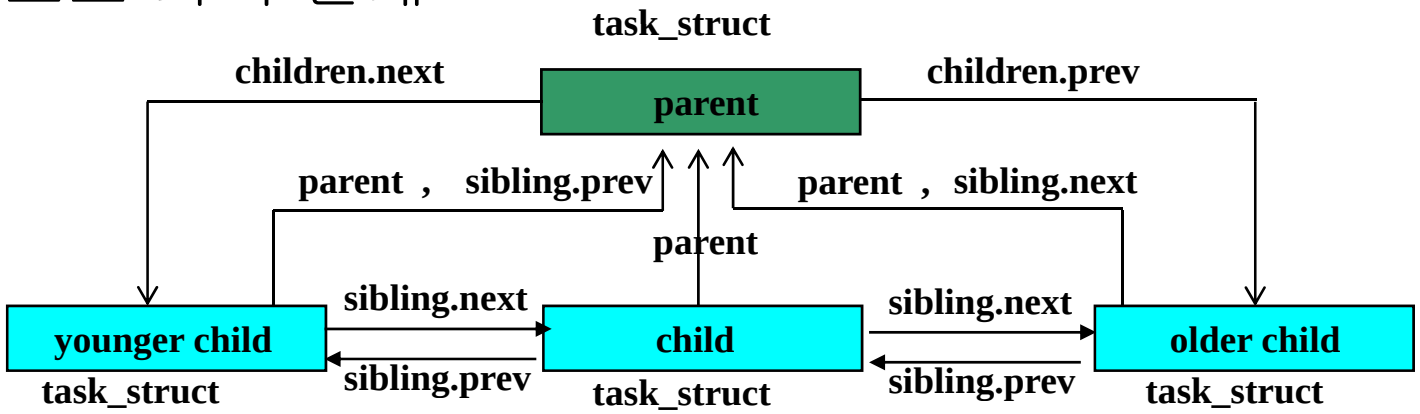
Task 연결 구조

태스크 연결 구조

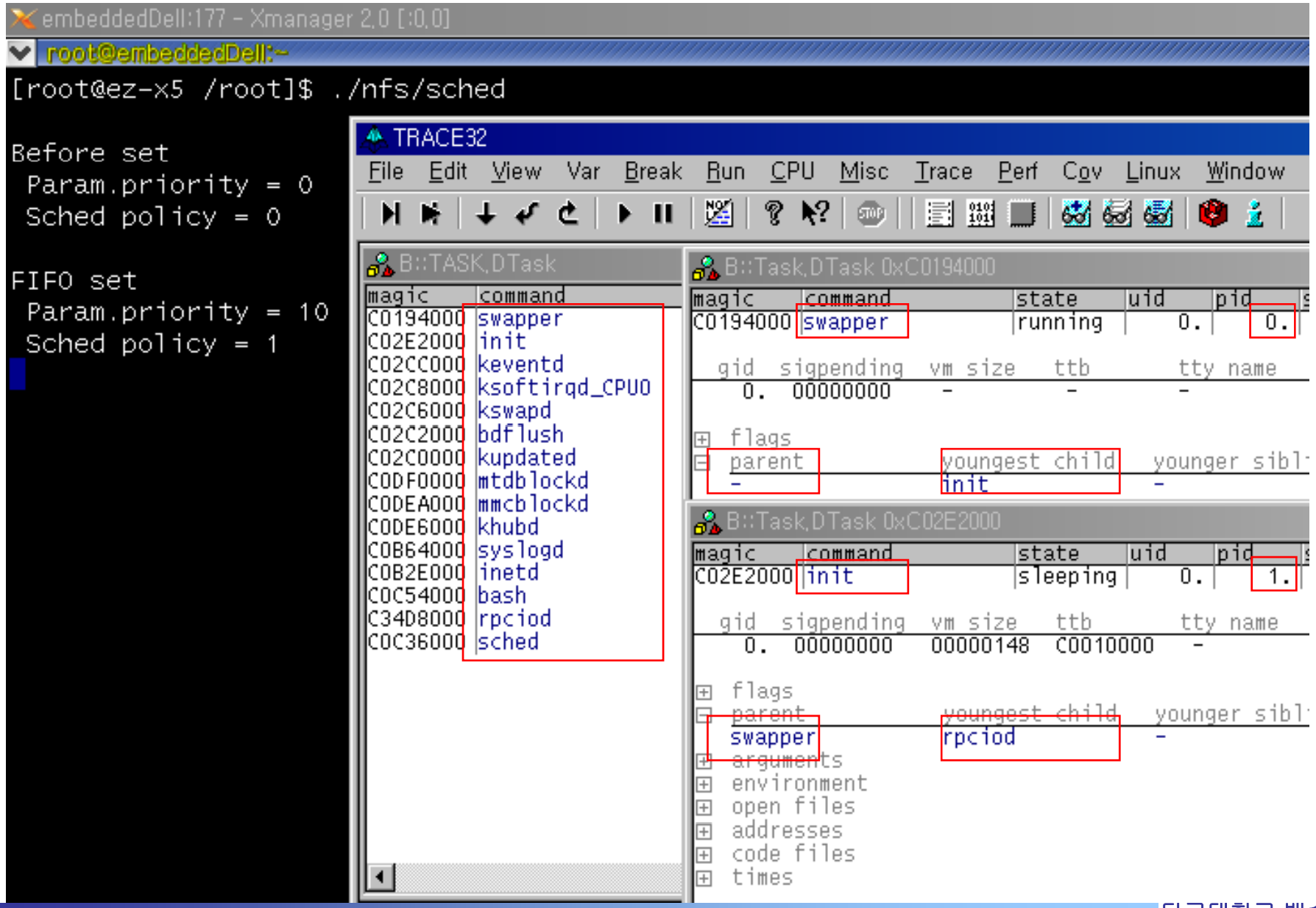


where is run_queue ? prev_task? next_task?

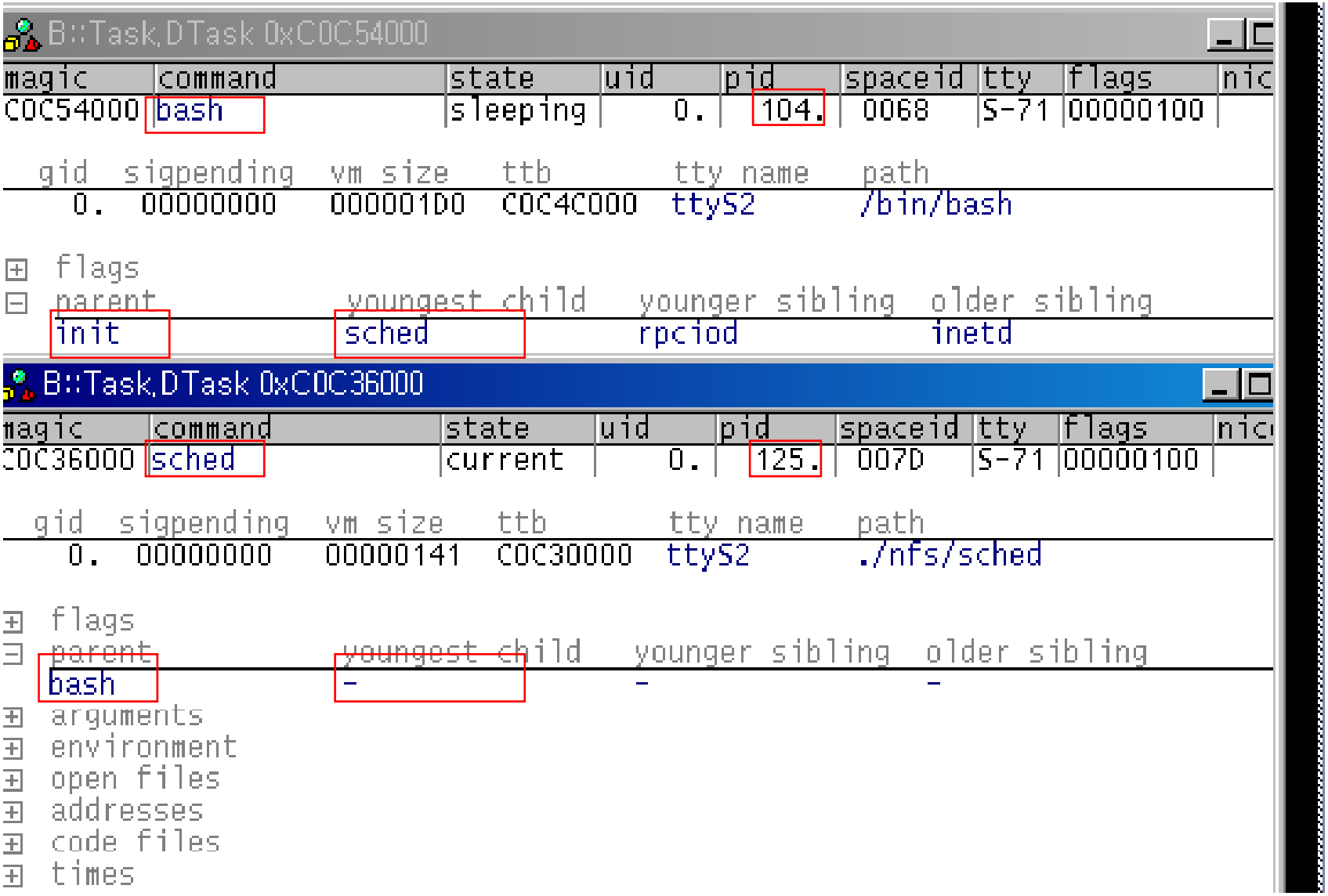
태스크 가족 관계



Task 계층 구조 확인(1/2)

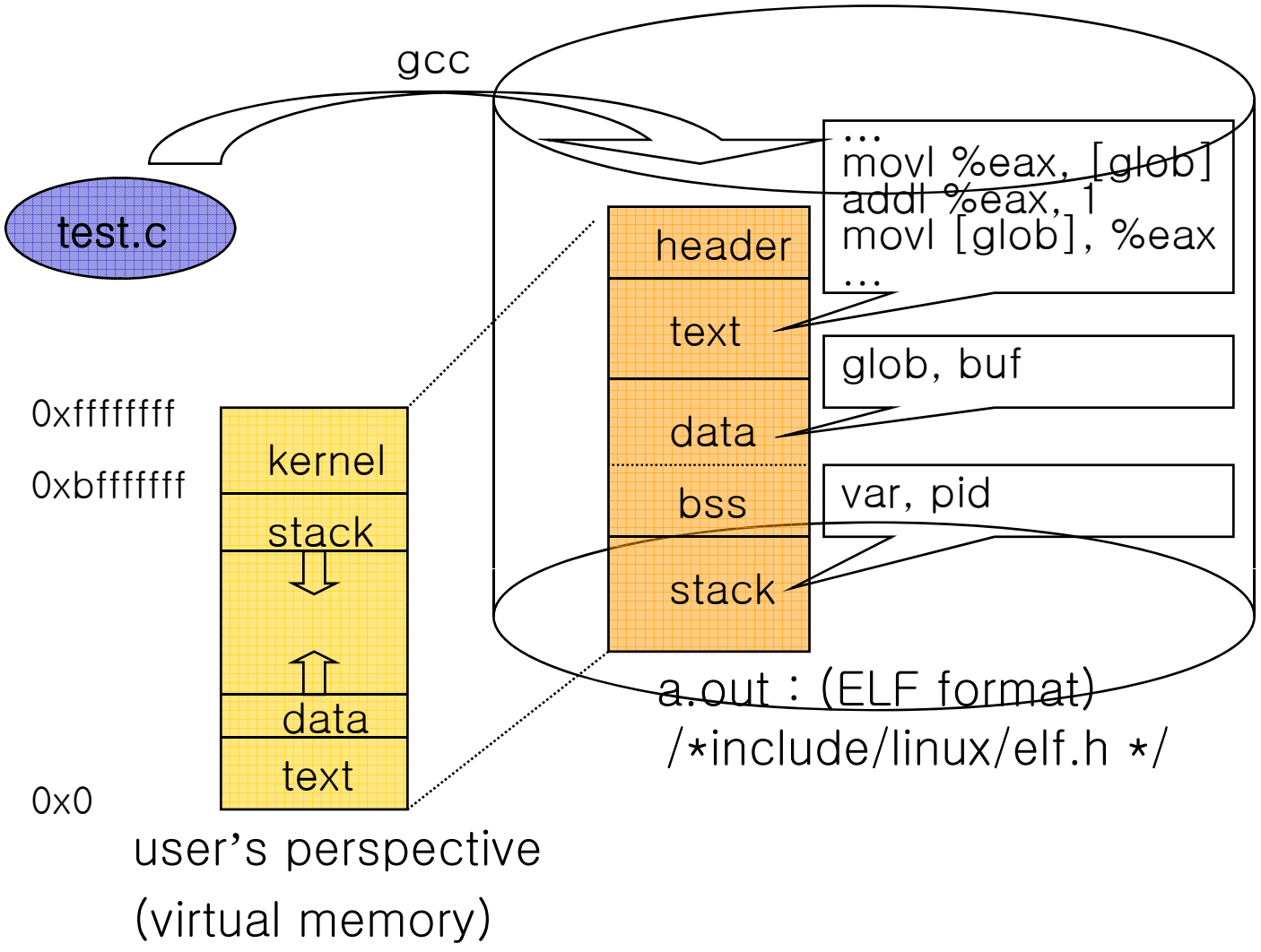


Task 계층 구조 확인(2/2)



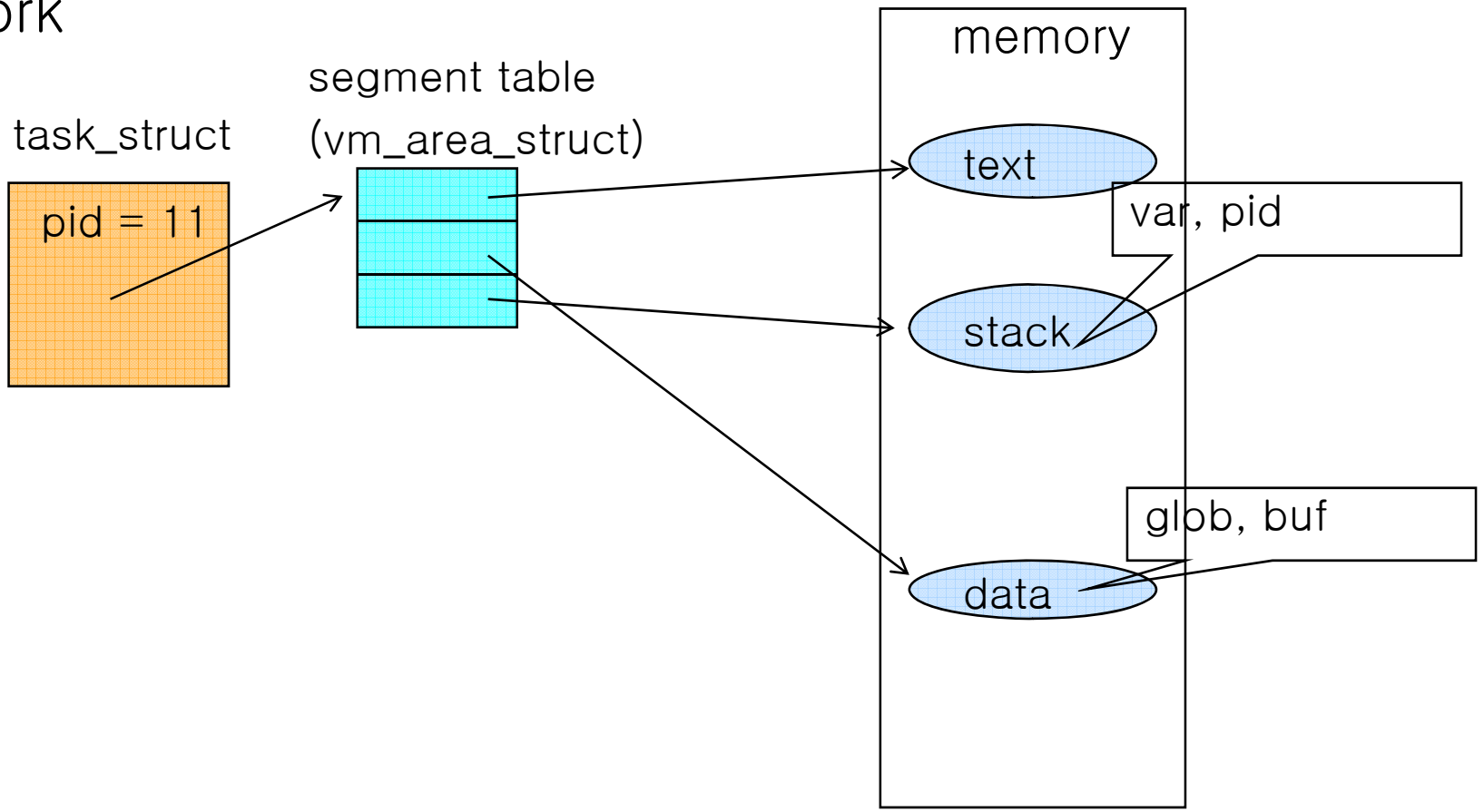
Memory Context (1/5)

- fork internal : compile results



Memory Context (2/5)

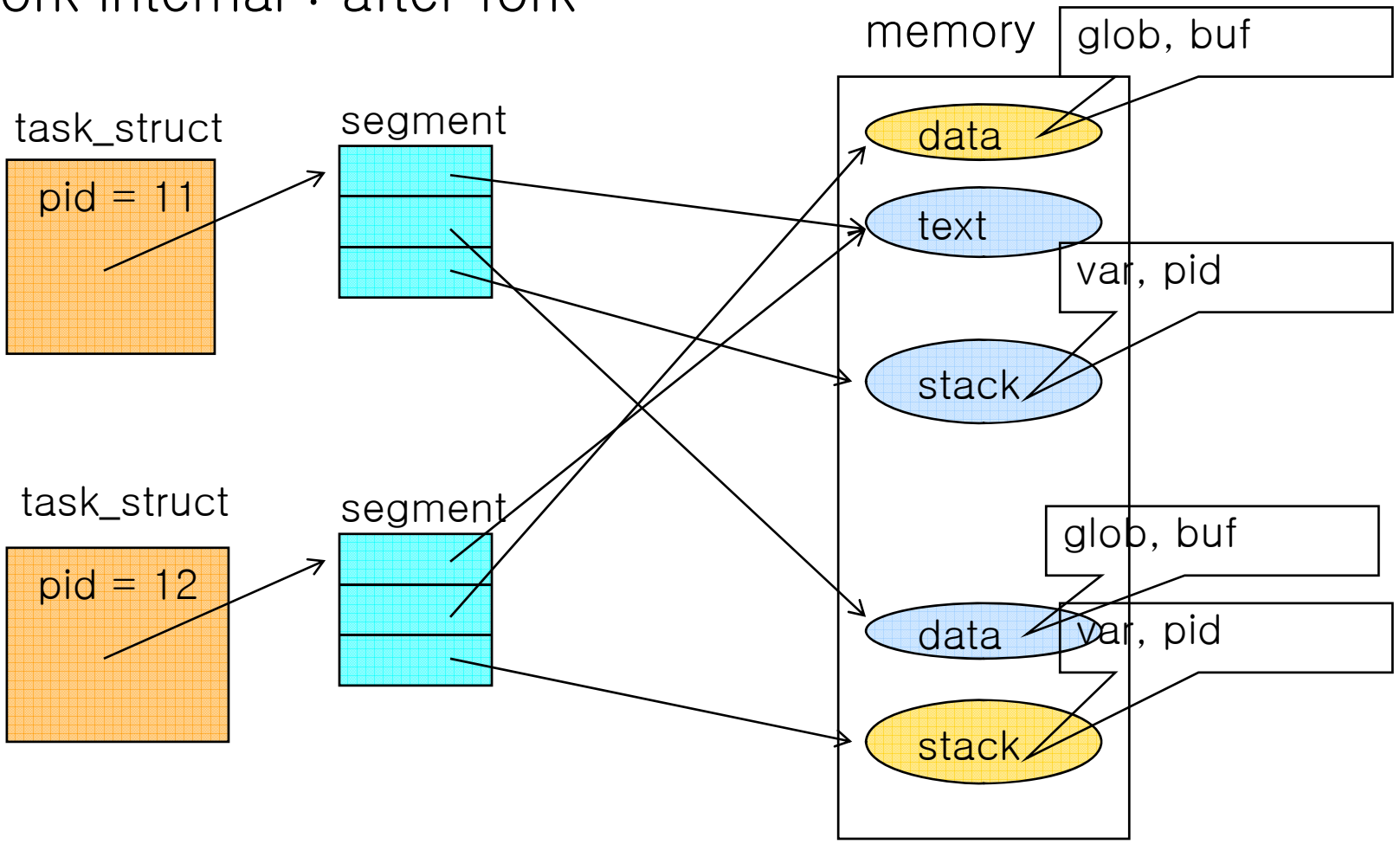
- fork internal : after loading (after run a.out) & before fork



✓ In this figure, we assume that there is no paging mechanism.

Memory Context (3/5)

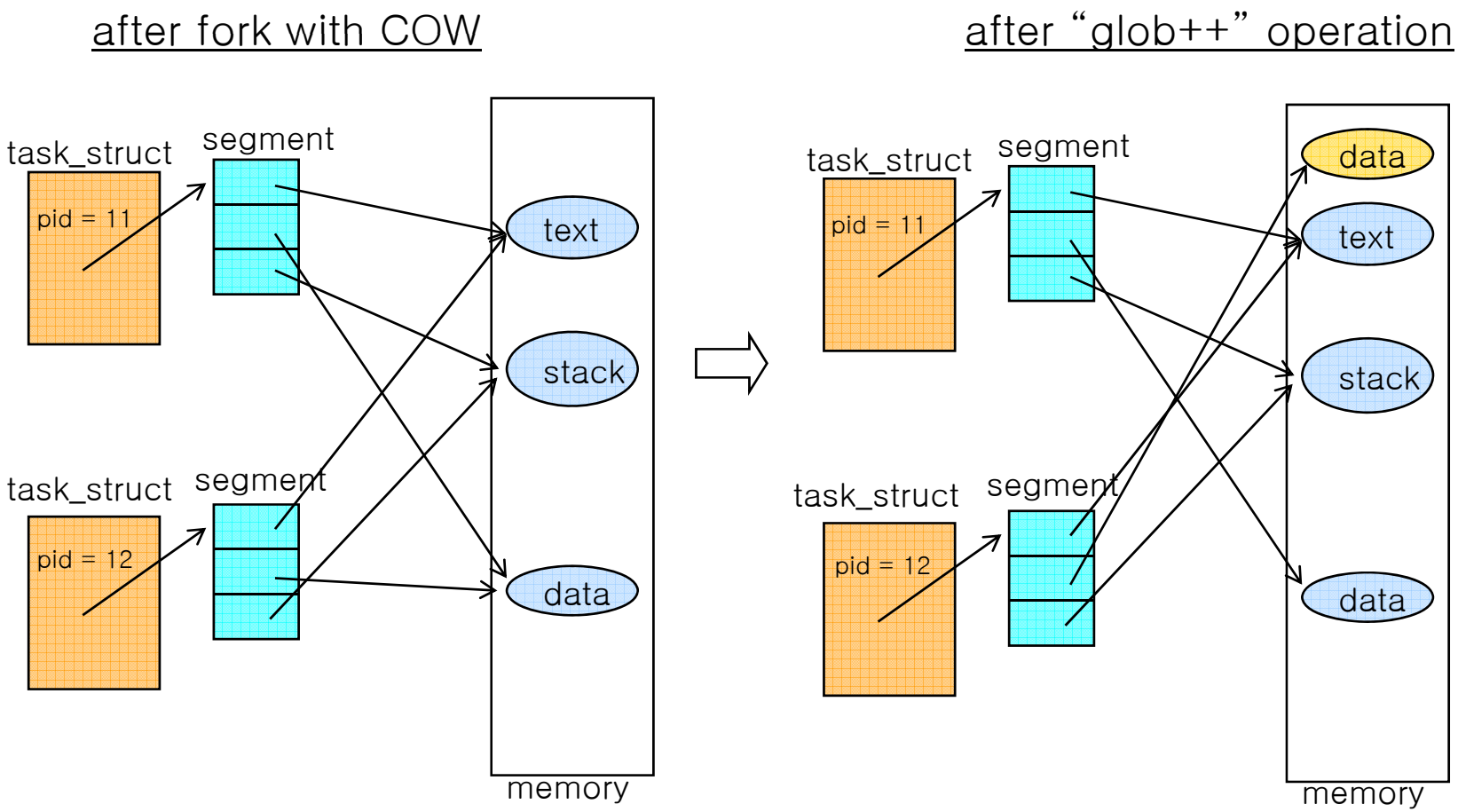
- fork internal : after fork



✓ address space : basic protection barrier

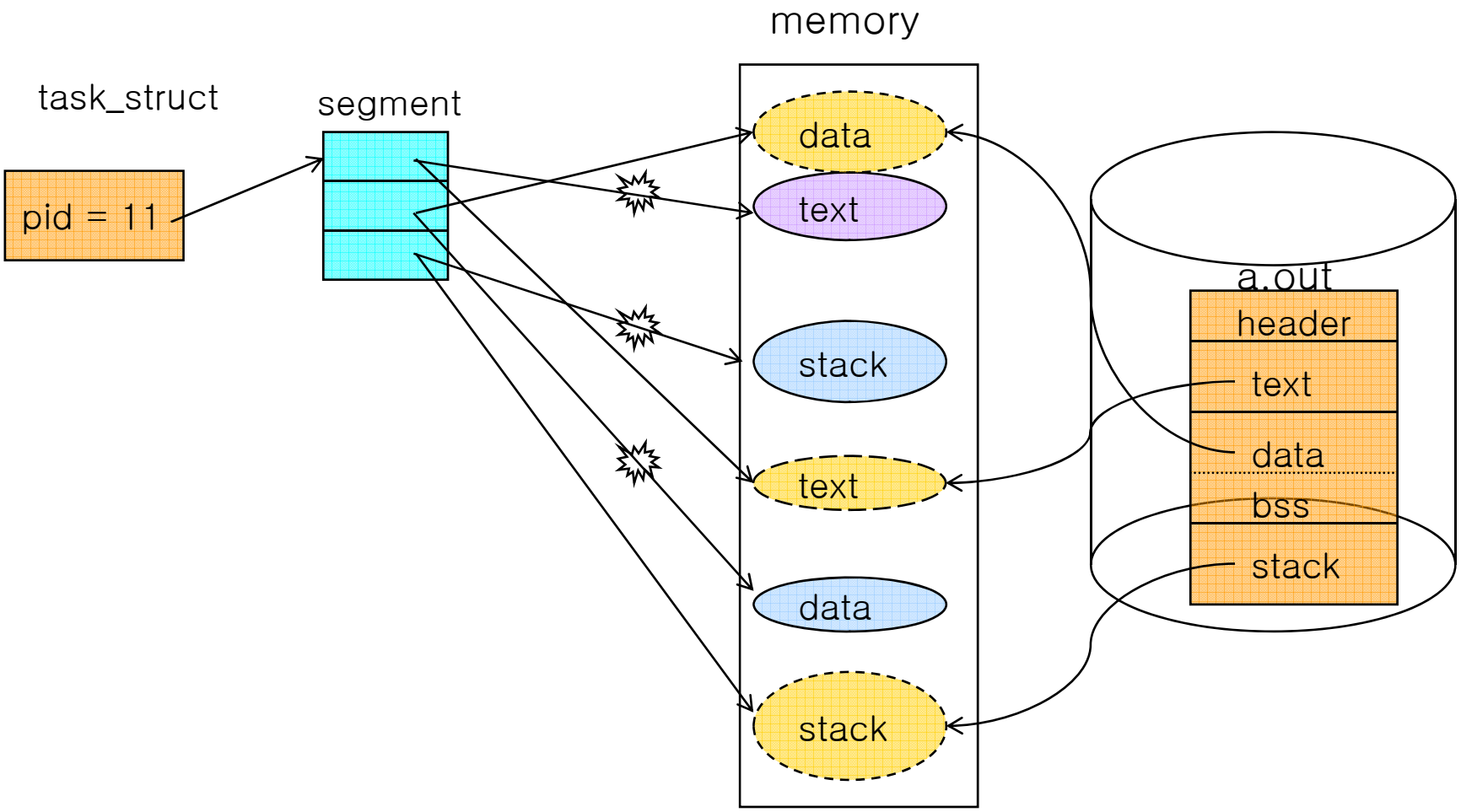
Memory Context (4/5)

- fork internal : with COW (Copy on Write) mechanism



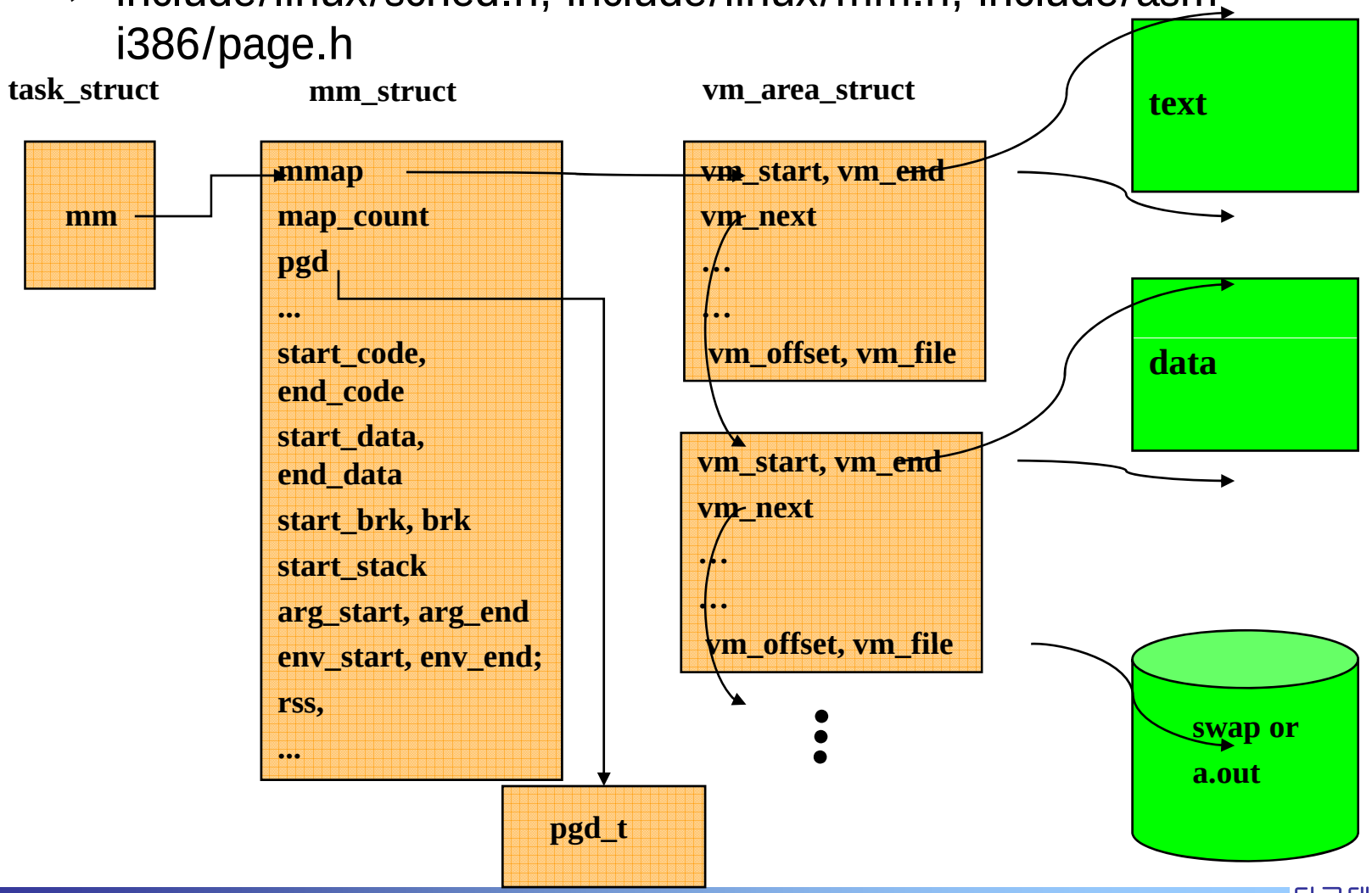
Memory Context (5/5)

- 새로운 프로그램 수행: execve() internal



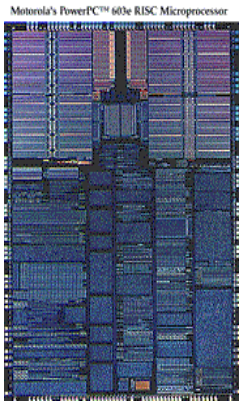
Task의 memory context

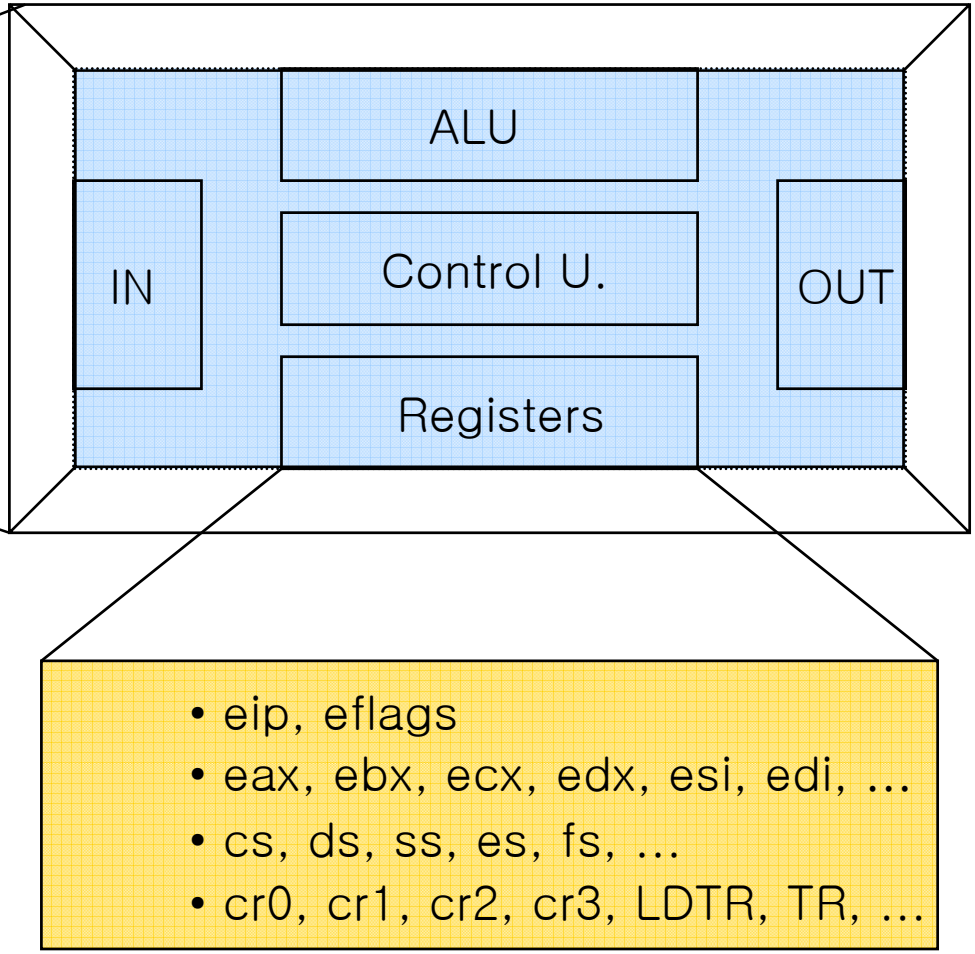
- 메모리 관리 자료 구조
  - ✓ include/linux/sched.h, include/linux/mm.h, include/asm-i386/page.h



Hardware Context (1/4)

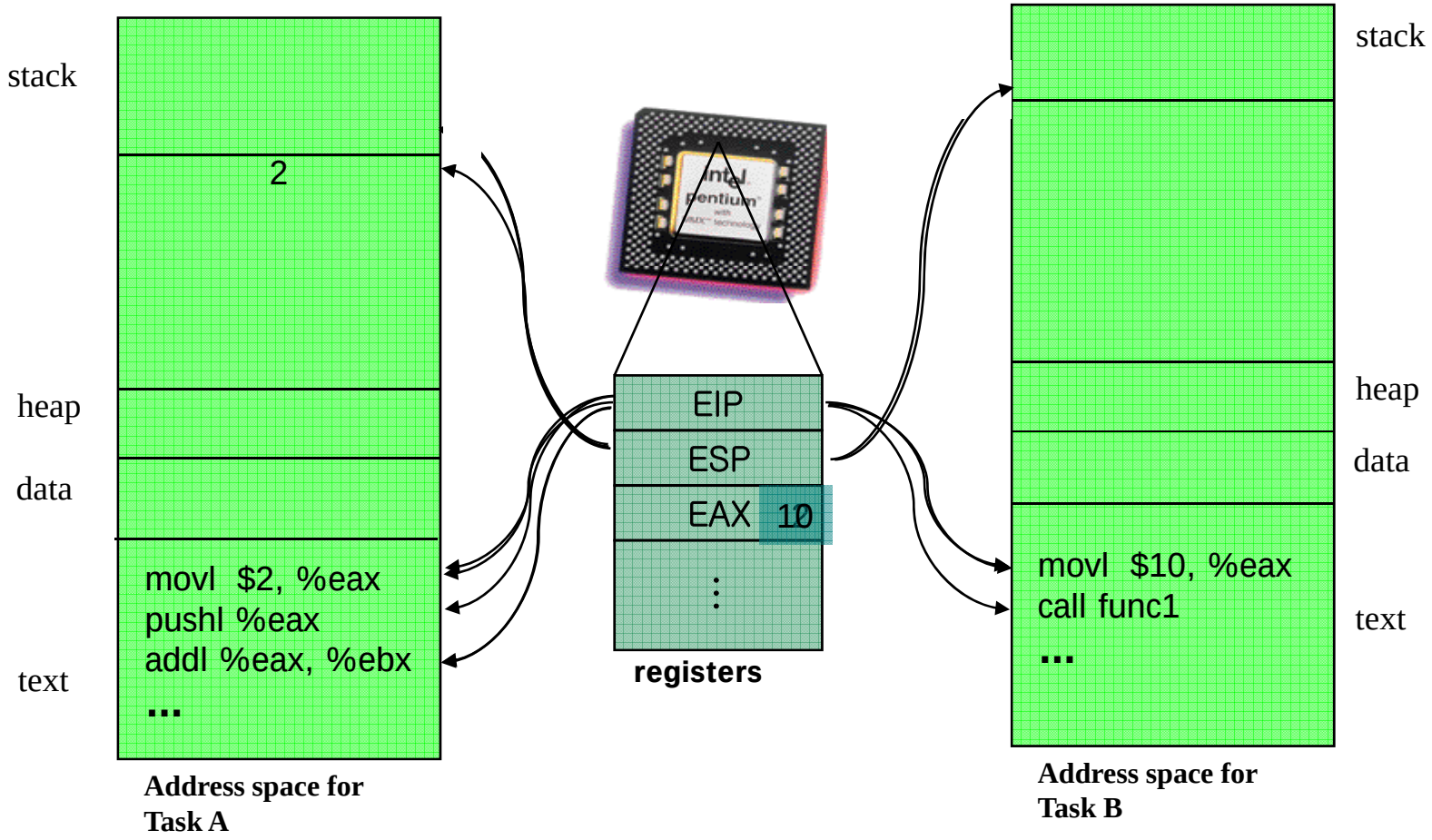
- brief reminds the 80x86 architecture





Hardware Context(2/4)

- 쓰레드 자료 구조 : CPU 추상화

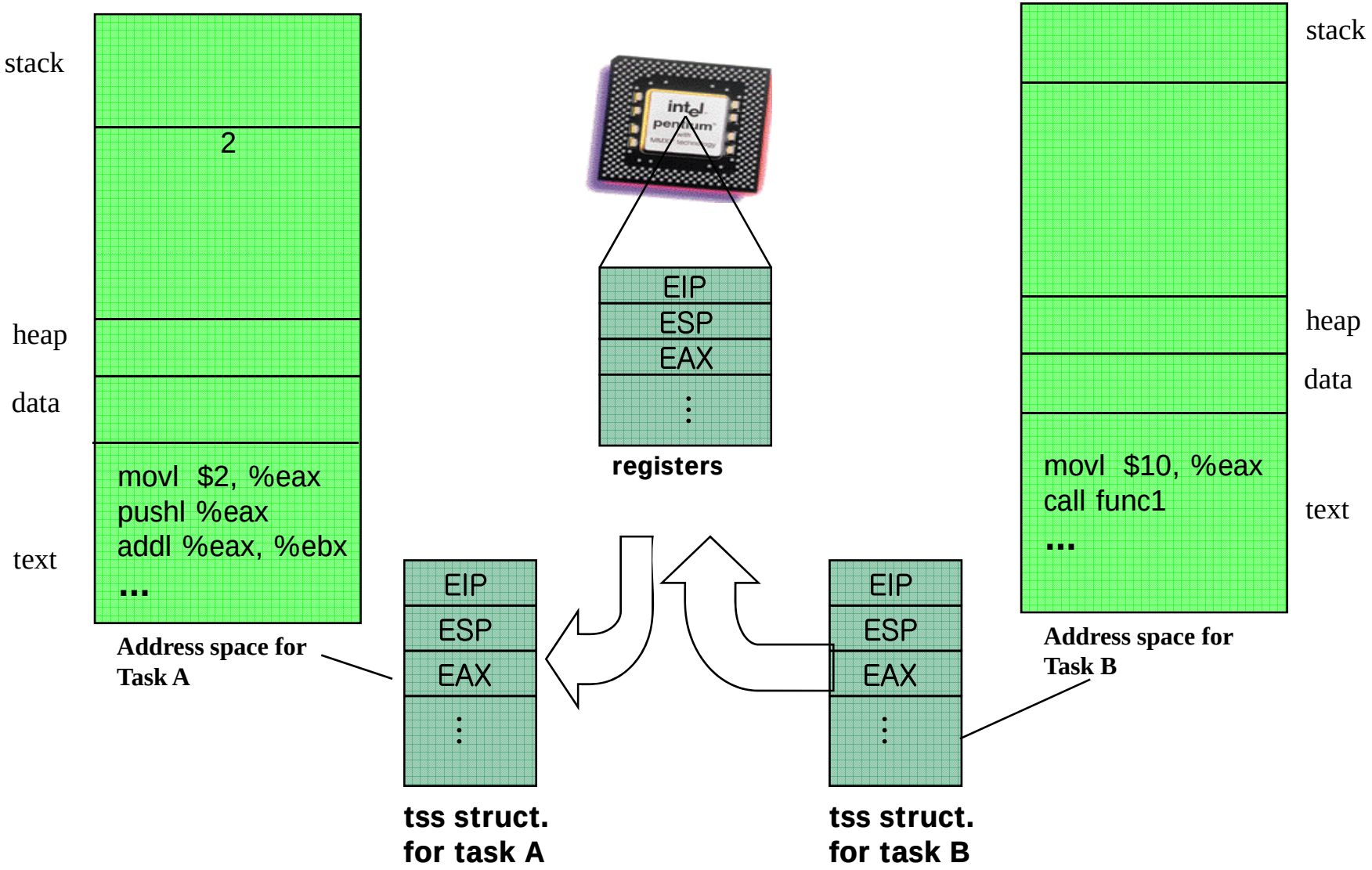


다시 **Task A**가 수행되려면?

문맥 교환**(Context Switch)** ➔ **thread structure**

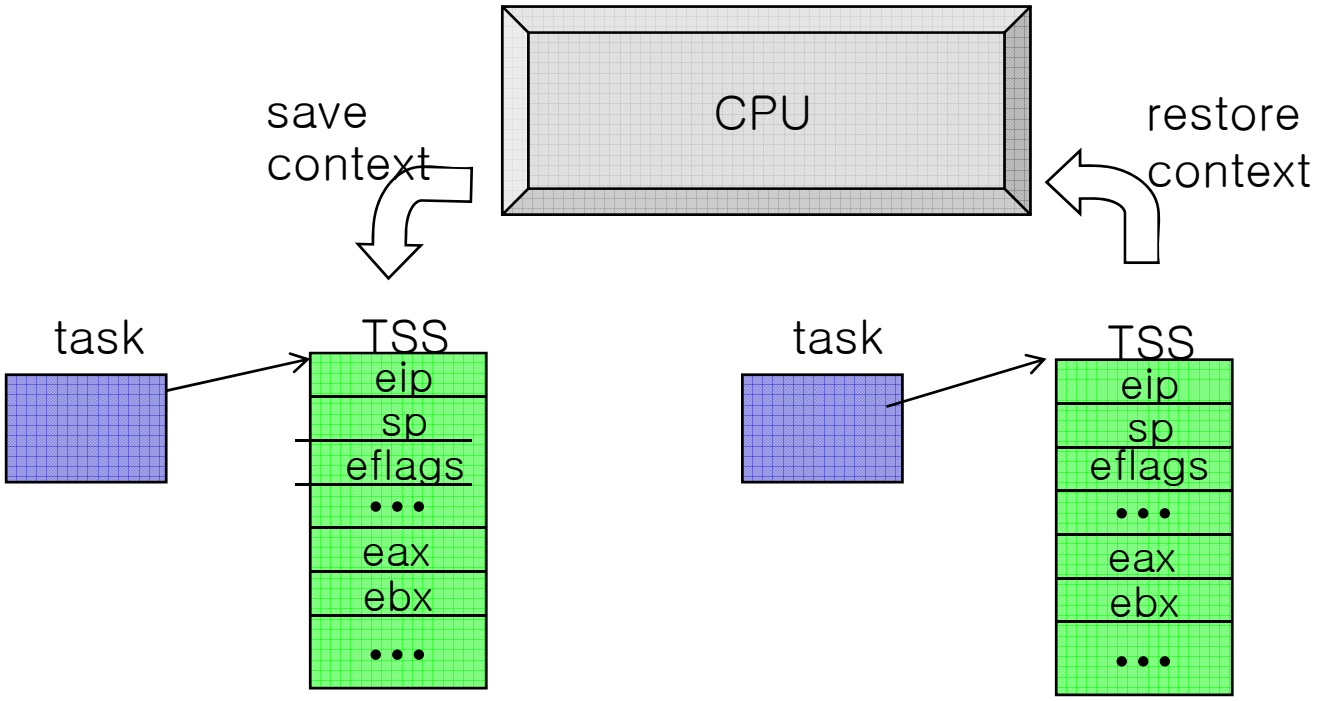
Hardware Context(3/4)

- 쓰레드 자료 구조 : CPU 추상화



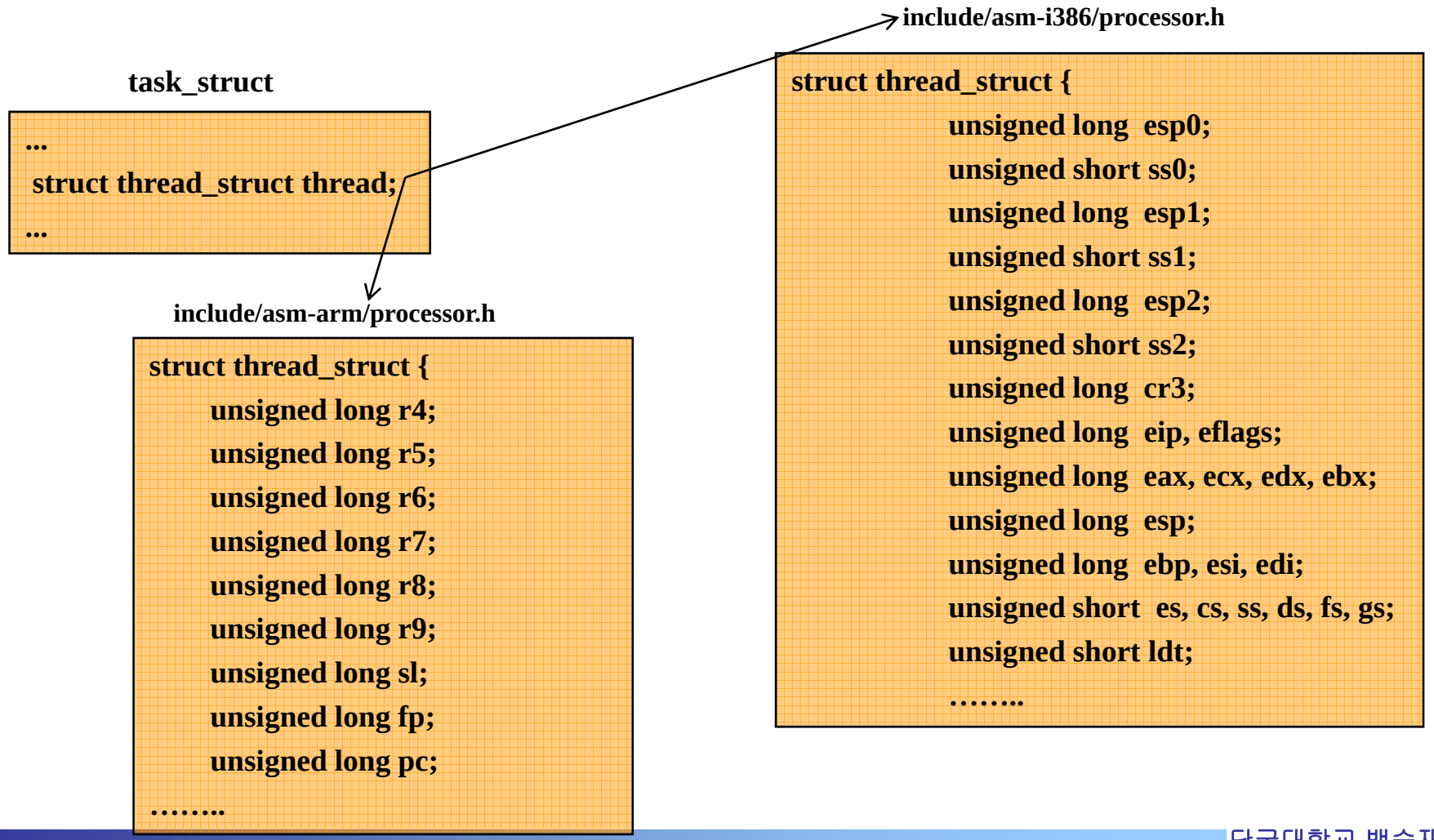
Hardware Context(4/4)

- 문맥 교환 (Context Switch)



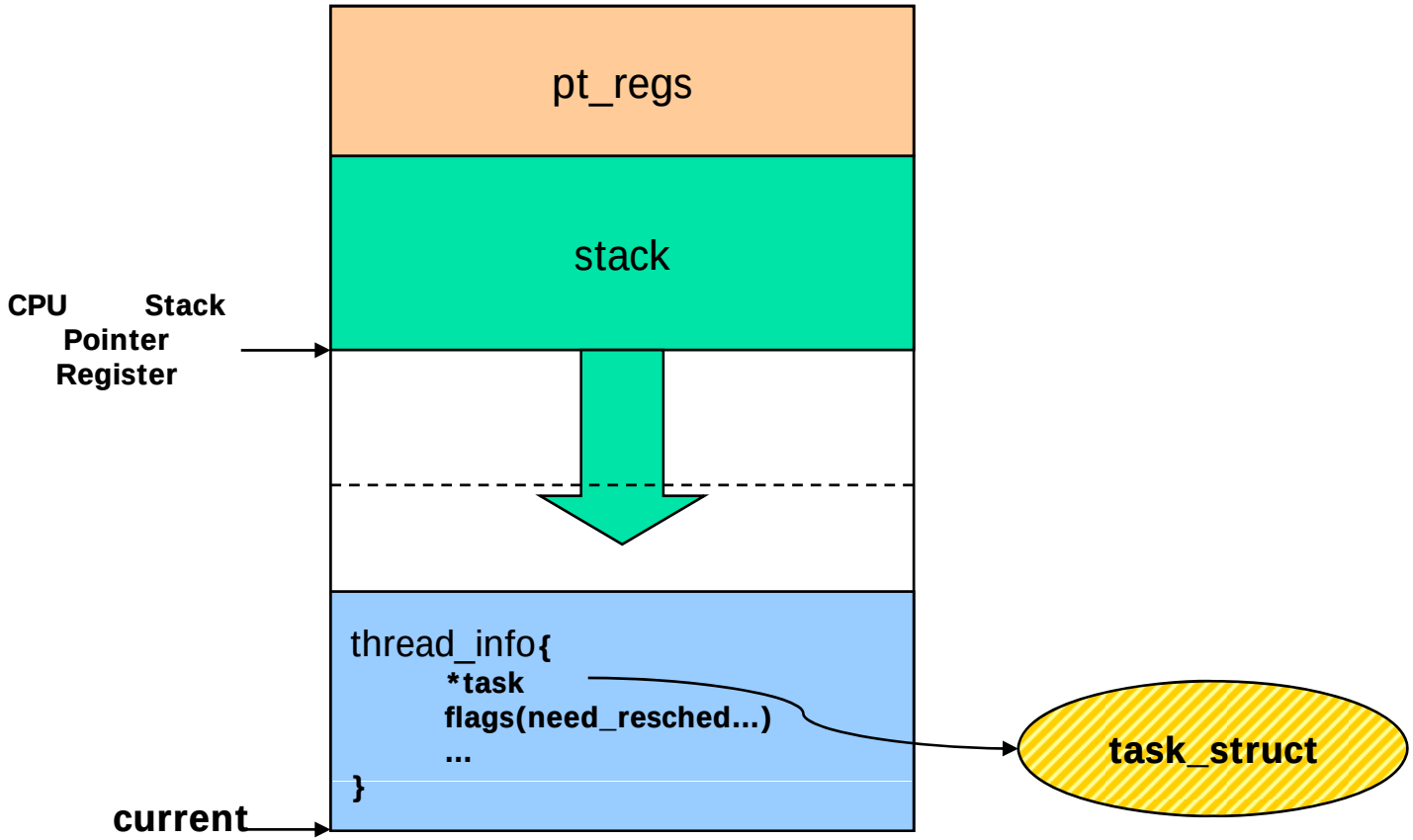
Task의 H/W context

- 쓰레드 자료 구조 : CPU 추상화



Kernel Stack

- Thread union
  - ✓ 커널은 각 태스크 별로 8KB메모리 할당
  - ✓ thread_info 구조체와 kernel stack
  - ✓ alloc_thread_struct, free_thread_struct



Kernel mode 진입

- INT, syscall ➔ kernel mode로 전환
  - ✓ control path : kernel mode로 진입하기 위한 일련의 명령어
- struct pt_regs ?
  - ✓ 현재 P의 상태를 커널 스택에 저장하기 위해 사용

```
asmlinkage long sys_open(const char * filename, int flags, int mode)
```

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

변환한 IRQ번호

- **sys_open()**처럼 인자가 임의의 개수인 경우 저장된 **pt_regs**구조체의 정보 중 위에서 부터 차례로 전달됨
- **sys_fork()**처럼 **pt_regs**를 통째로 받는 경우도 있음

```
asmlinkage int sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.esp, &regs, 0);
}
```

Linux의 Task 생성

- 프로세스 관련 Interface
  - ✓ fork(), vfork(), clone() ➔ do_fork() ➔ copy_process()
    - dup_task_struct(): 새 커널 스택, 부모의 task_struct복사
    - get_pid()를 호출해 새로운 PID할당
    - 플래그에 따라 열린 파일, 파일시스템 정보, 시그널 핸들러, 프로세스 주소 영역, namespace등을 복제 / 공유
      - 하위 바이트는 자식 프로세스 종료 시 부모 프로세스에 전달할 시그널 신호(보통 SIGCHLD), 나머지 3바이트는 아래와 같음
        - CLONE_VM : 메모리 디스크립터와 모든 PT공유
        - CLONE_FS : 루트 dir과 CWD나타내는 table과 'umask'값 공유
        - CLONE_FILES : 열린 file나타내는 테이블 공유
        - CLONE_PARENT : 새로 만들어지는 자식 P의 부모를 호출P의 부모로 설정
        - CLONE_PID : PID공유
        - CLONE_PTRACE : ptrace()로 부모 P 추적중 이라면, 자식 P도 추적가능
        - CLONE_SIGHAND : 시그널 핸들러 나타내는 table공유
        - CLONE_THREAD : 자식P를 부모P와 같은 스레드 그룹에 넣고, 자식P의 tgid필드도 이에 알맞게 설정, CLONE_PARENT자동 설정됨
        - CLONE_SIGNAL : ➔ CLONE_SIGHAND + CLONE_THREAD ➔ 멀티스레드 APP에 있는 모든 스레드에 시그널 송신 가능
        - CLONE_VFORK : vfork() syst call서 사용함
    - 남은 time slice를 부모와 자식간에 분배
    - 새로운 프로세스의 포인터 반환

Linux의 Task 제거

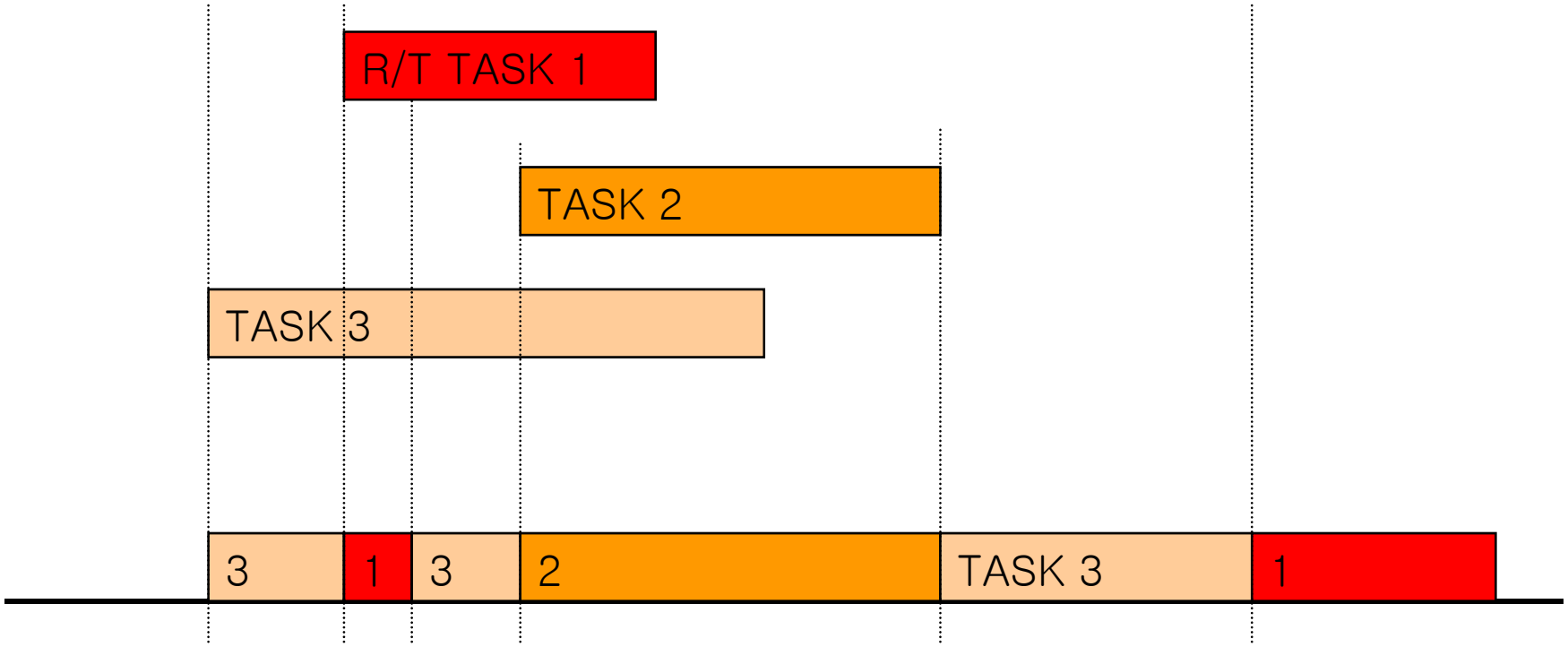
- exit() ➔ do_exit()
  - ✓ task_struct구조체의 flags멤버에 PF_EXITING플래그 설정
  - ✓ __exit_mm(), sem_exit(), __exit_files(), __exit_fs(), exit_namespace(), exit_sighand()
  - ✓ 태스크의 종료 코드 ➔ task_struct 구조체의 exit_code 멤버
  - ✓ exit_notify() ➔ 부모에게 시그널
  - ✓ 자식들의 부모를 같은 스레드 그룹의 다른 스레드나 혹은 init프로세스로 지정
    - 고아라면?
      - forget_original_parent()를 사용함
      - Reaper를 프로세스의 스레드그룹에 있는 다른 태스크로 설정
      - 다른 태스크가 없으면 reaper를 child_reaper로 설정 -> 이것이 바로 init프로세스이다
  - ✓ 태스크의 상태를 TASK_ZOMBIE로 설정
    - 이후에는 부모에게 정보를 전달하기 위한 커널 스택과 슬랩 객체 즉, thread_info와 task_struct구조체가 남아있다.
  - ✓ schedule()함수를 호출해 새 프로세스로 스위칭

스케줄링 기법

- 선점 지원 여부에 따른 분류
  - ✓ 선점 스케줄링
  - ✓ 비선점 스케줄링
- 우선 순위 변경 지원 여부에 따른 분류
  - ✓ 정적 우선 순위 스케줄링
  - ✓ 동적 우선 순위 스케줄링

- 구체적인 스케줄링 알고리즘
  - ✓ FIFO 스케줄링
  - ✓ 다단계 피드백 큐 스케줄링
  - ✓ SJT, SRT 스케줄링
  - ✓ EDF? RM? RR?...

Priority Inversion Problem

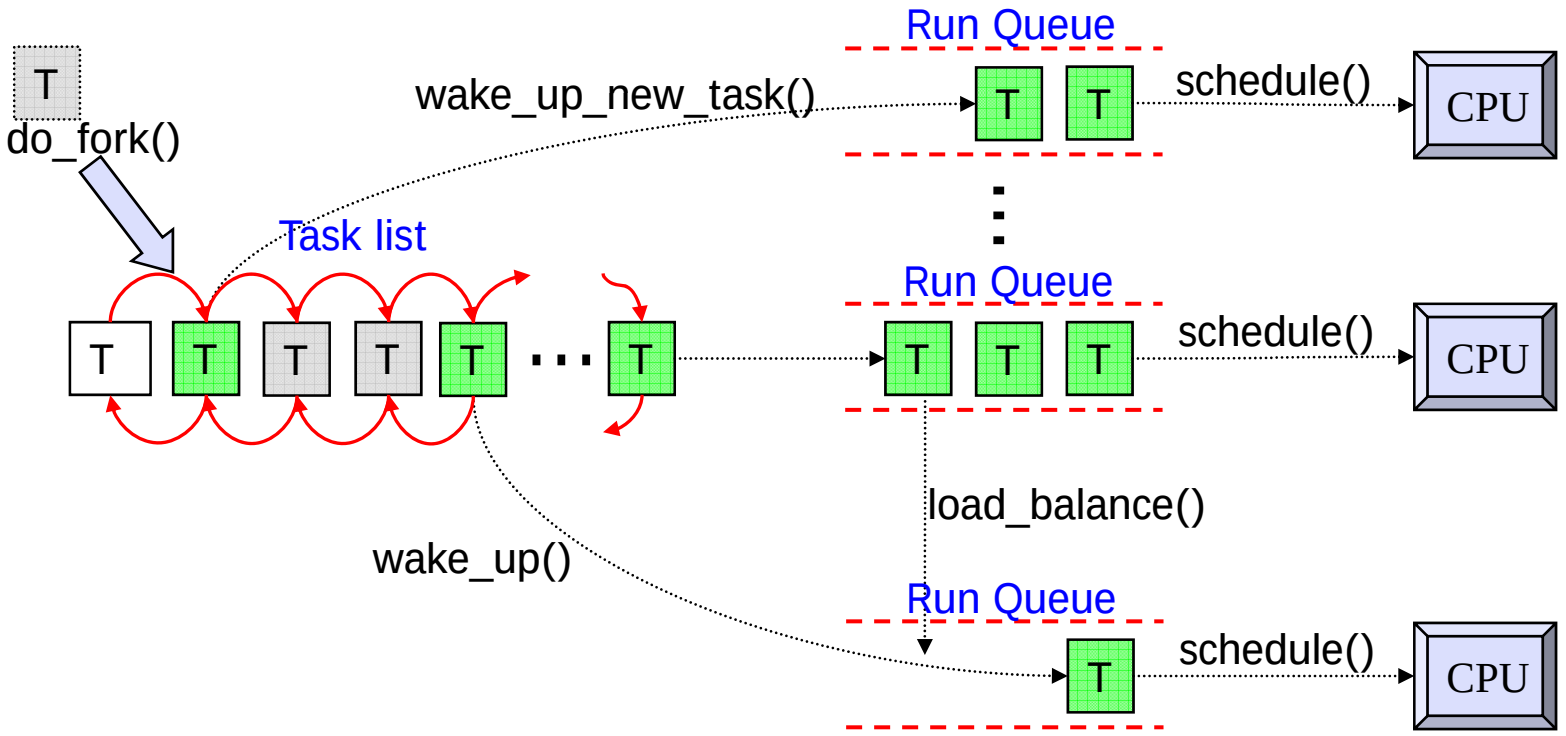


- 해결책은?
  - ✓ Priority inheritance→ task 3의 우선순위를 R/T task 1의 우선순위와 같게 하여 수행 후, R/T task 1이 기다리는 자원에 대한 처리가 끝나면 원래 우선순위로 복귀 시킴

  ➔ 동적 우선 순위 변경이 가능해야 한다

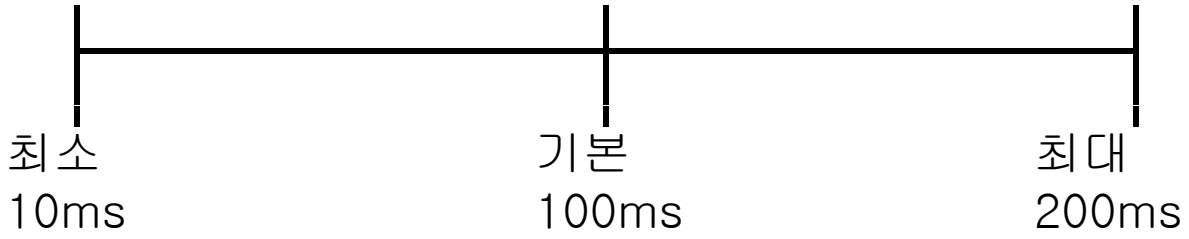
Multi-CPU를 위한 리눅스 특징

- 스케줄링
  - ✓ Run queue per each CPU
  - ✓ Effective priority based O(1) scheduler
    - Priority, affinity, interactivity, …
  - ✓ Load balancing



Linux Scheduling

- Multi-level feed back queue based on priority
- Task의 종류
  - ✓ 실시간 태스크(SCHED_FIFO, SCHED_RR)
    - 정적 우선 순위
    - 0 ~ 99
  - ✓ 일반 태스크(SCHED_OTHER)동적 우선 순위
    - Nice ➔ -20 ~ 19 ➔ 100 ~ 139
- 타임 슬라이스
  - ✓ Interactivity를 고려한 time slice와 priority의 재 계산



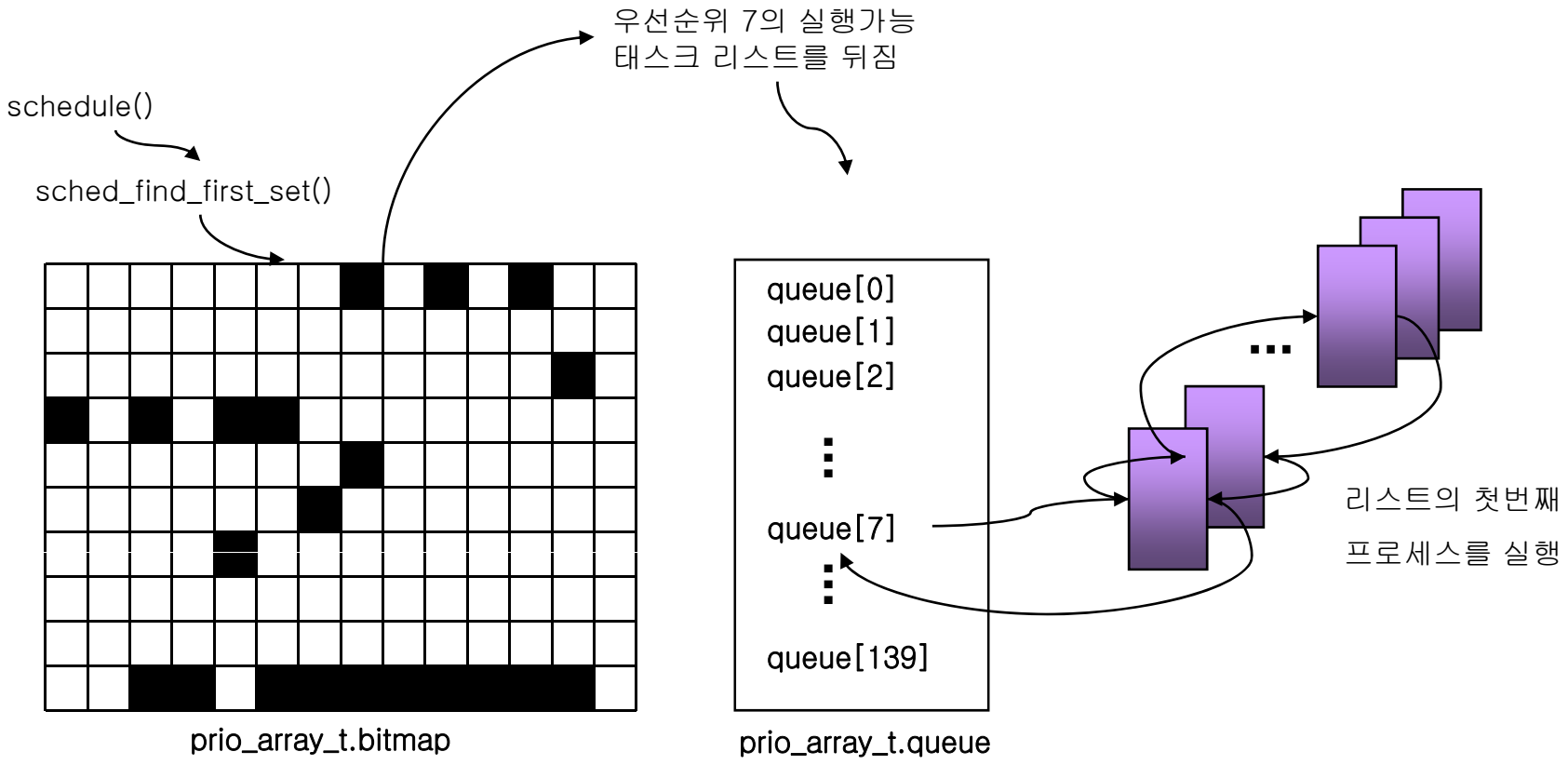
Run Queue

- 실행 큐
  - ✓ 프로세서에 있는 실행 가능 한 프로세스의 목록

- 우선순위 배열 → O(1) scheduling
  - ✓ 각 실행 큐는 2개의 우선순위 배열을 가짐
  - ✓ 활성(active), 비활성(expired)
  - ✓ 각 우선순위 레벨마다의 실행가능 프로세스 큐를 유지
  - ✓ 각 큐에는 해당 우선순위 레벨의 실행 가능한 프로세스 목록 유지
  - ✓ 우선순위 비트맵 사용 → 상수 시간 내에 최고 우선순위 태스크 선택 가능
    - 특정 우선 순위의 태스크가 TASK_RUNNING상태가 되면 해당 우선순위 비트가 1로 set됨
    - sched_find_first_bit()이용 첫 번째 1로 set된 비트 찾음
    - 같은 우선 순위끼리는 라운드 로빈

Run Queue 동작

- 타임 슬라이스의 재계산
  - ✓ 기존의 타임슬라이스 재계산 위한 루프 형태의 루틴 탈피
  - ✓ 단지, 활성과 비 활성 배열을 스위칭 하는 것으로 끝남



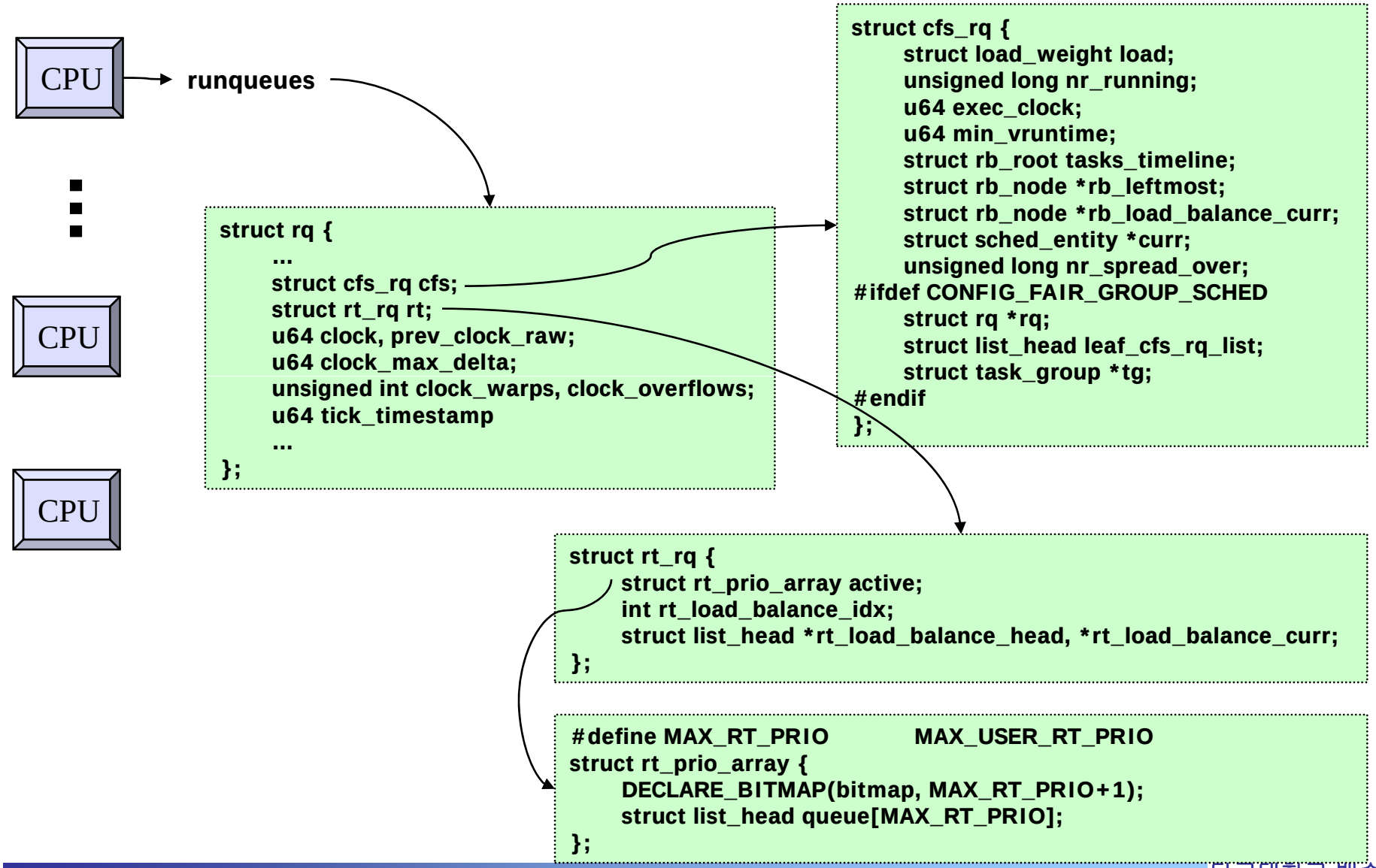
schedule() 함수의 호출

- schedule()함수
  - ✓ 직접적인 호출(direct invocation)
    - current process가 필요로 하는 자원을 사용할 수 없어, 이 process를 당장 블록 해야 하는 경우
    - Process를 block 하려는 kernel routine의 단계
      - current를 적당한 wait queue에 넣는다
      - current의 상태를 TASK_INTERRUPTIBLE 나 TASK_UNINTERRUPTIBLE로 만든다
      - schedule()을 호출
      - 자원이 사용가능한지 검사하여 불가하면 2단계로 돌아감
      - 사용 가능하면 current를 대기 큐에서 제거
  - ✓ 우회적인 호출(lazy invocation)
    - current의 need_resched 필드를 1로 설정해 우회적 호출함
    - 쓰이는 경우
      - current가 자신의 CPU시간 quantum을 모두 썼을 때, update_process_times()가 설정함
      - 임의의 process가 깨어 났는데, 그 process 의 우선순위가 현재 process 의 우선순위 보다 높을 때.
      - seched_setscheduler()이나, sched_yield() 시스템 콜 호출 시

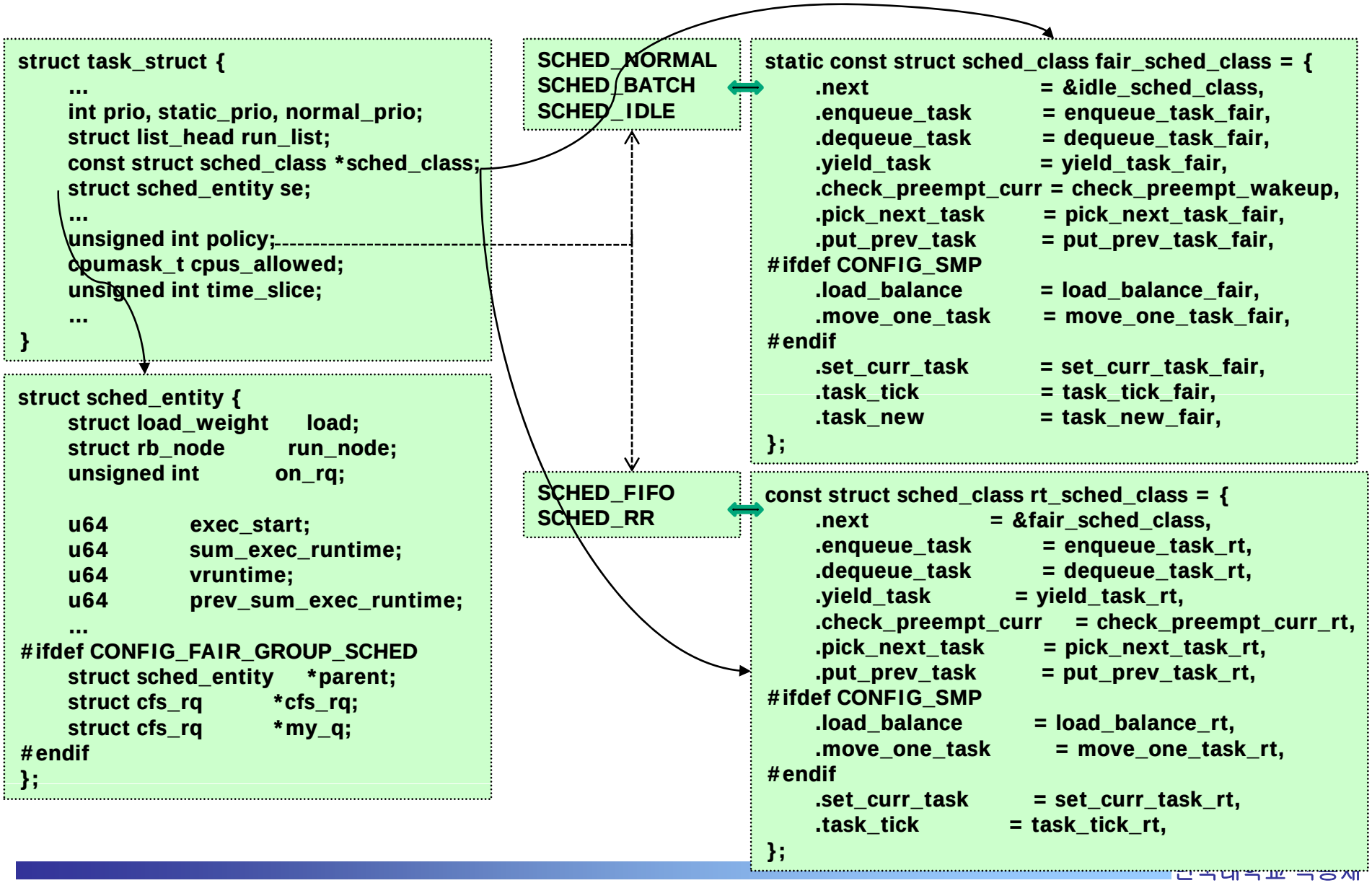
CFS basic concept

- If N tasks are on the Runqueue
  - ✓ fair_clock(virtual clock)
    - Real Clock * 1/N
  - ✓ wait_runtime
    - Waiting time of a task
- To sort tasks on the red-black tree
  - ✓ fair_clock – wait_runtime

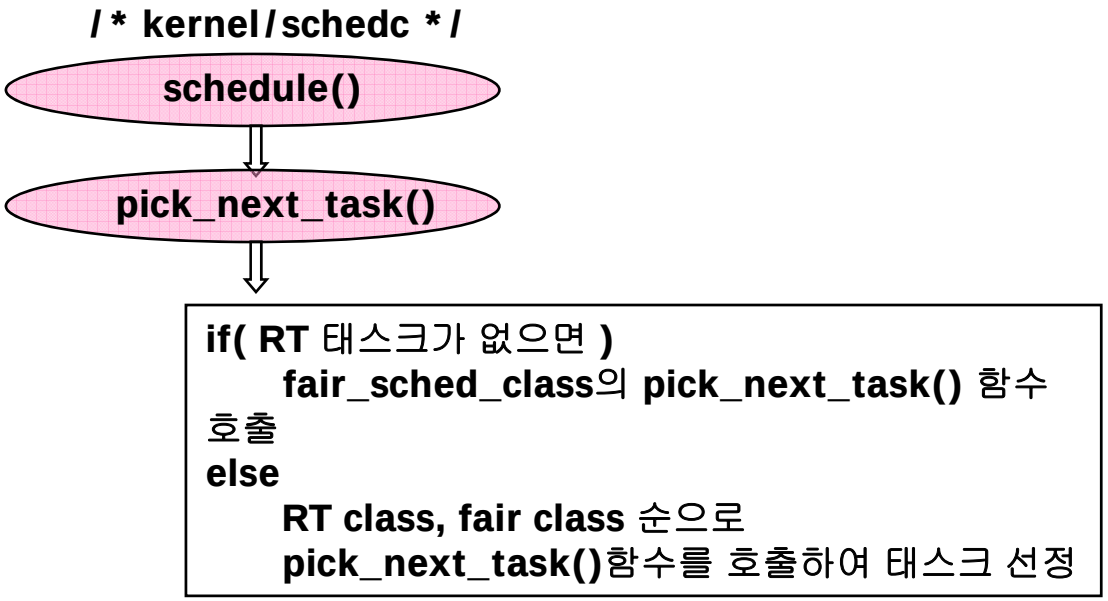
2.6.23부터 도입된 CFS 스케줄러의 자료구조



task와 스케줄러 관련 구조체



schedule()함수의 동작



Test Code

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

int main(void)
{
        struct sched_param param;
        int i, j;

        sched_getparam( 0, &param);

        printf(" ₩nBefore set₩n");
        printf(" Param.priority = %d₩n", param.sched_priority);
        printf(" Sched policy = %d₩n", sched_getscheduler(0));
                                ①
        param.sched_priority = 10;
        sched_setscheduler(0, SCHED_FIFO, &param);
        sched_getparam( 0, &param);

        printf(" ₩nFIFO set₩n");
        printf(" Param.priority = %d₩n", param.sched_priority);
        printf(" Sched policy = %d₩n", sched_getscheduler(0));
                                ②
        param.sched_priority = 20;
        sched_setscheduler(0, SCHED_RR, &param);
        sched_getparam( 0, &param);

        printf(" ₩nRR set₩n");
        printf(" Param.priority = %d₩n", param.sched_priority);
        printf(" Sched policy = %d₩n", sched_getscheduler(0));
                                ③
        return 0;
}
```

```
[root@embeddedDell root]# ./sched

Before set
 Param.priority = 0
 Sched policy = 0

FIFO set
 Param.priority = 10
 Sched policy = 1

RR set
 Param.priority = 20
 Sched policy = 2
[root@embeddedDell root]#
```

아래 코드를 화살표가 가리키고 있는 세 군데에 각각 넣고 실행해 보자.

실행 도중 키보드를 누르면 반응이 있는가?

언제 반응이 있고, 언제 없는가?

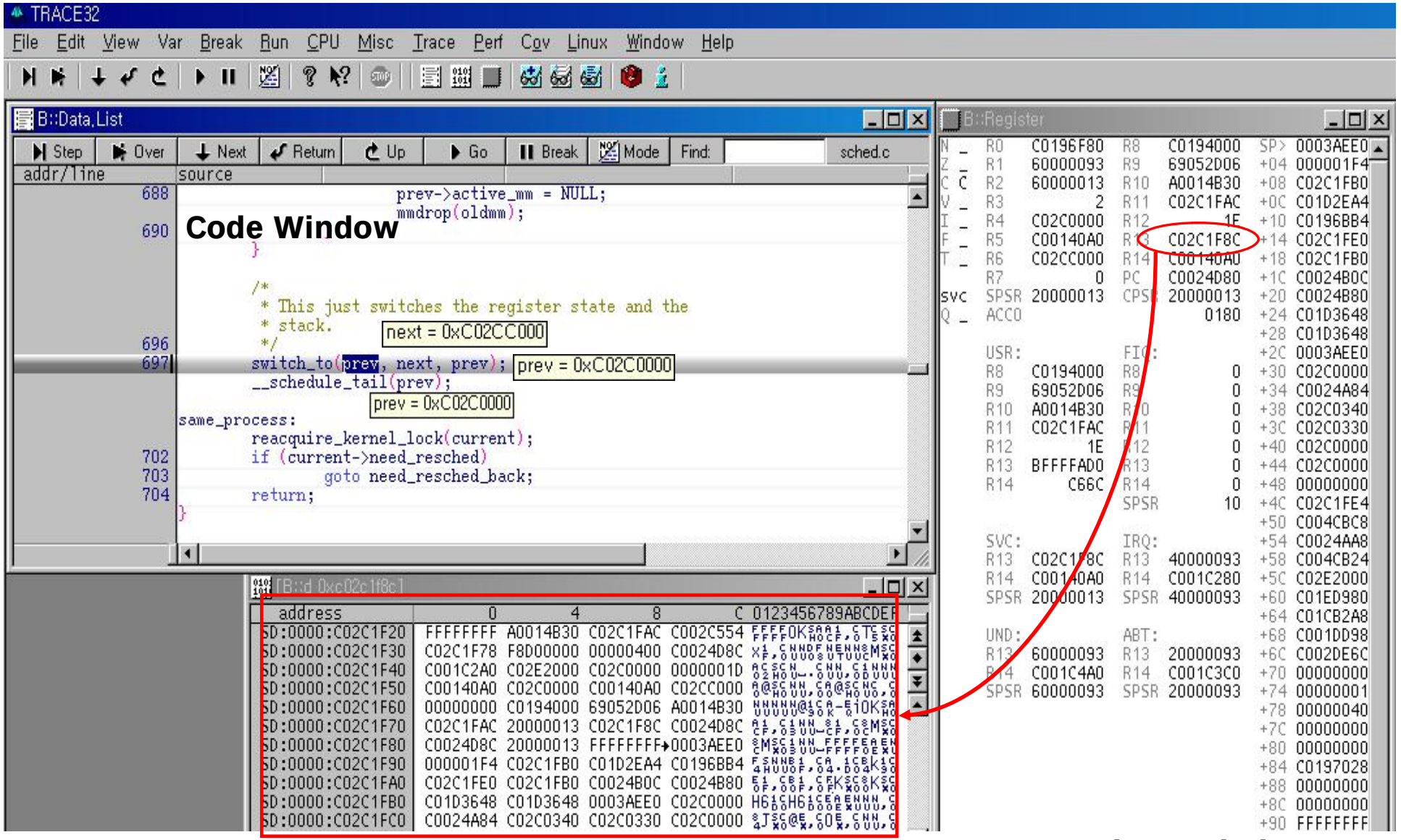
```
for(i=0; i<100000; i++)
    for(j=0; j<100000; j++);
```

Context Switching

- Context?
  - ✓ 사용자 주소 공간 : code, data, stack, 공유메모리
  - ✓ 제어정보 : task_struct
  - ✓ credentials : 보안 유지 위한 정보, uid.gid
  - ✓ 환경변수
  - ✓ H/W context : reg 값

- 진행 과정
  - ✓ 필요 시 schedule()가 context_switch()호출
  - ✓ switch_mm()을 호출
    - 이전 프로세스의 가상 메모리 매핑을 새 프로세스의 것으로 대체
  - ✓ switch_to()를 호출
    - 프로세서 상태를 이전 프로세스에서 현재 프로세스로 전환
    - 스택 정보와 프로세서 레지스터 값 저장 / 복원

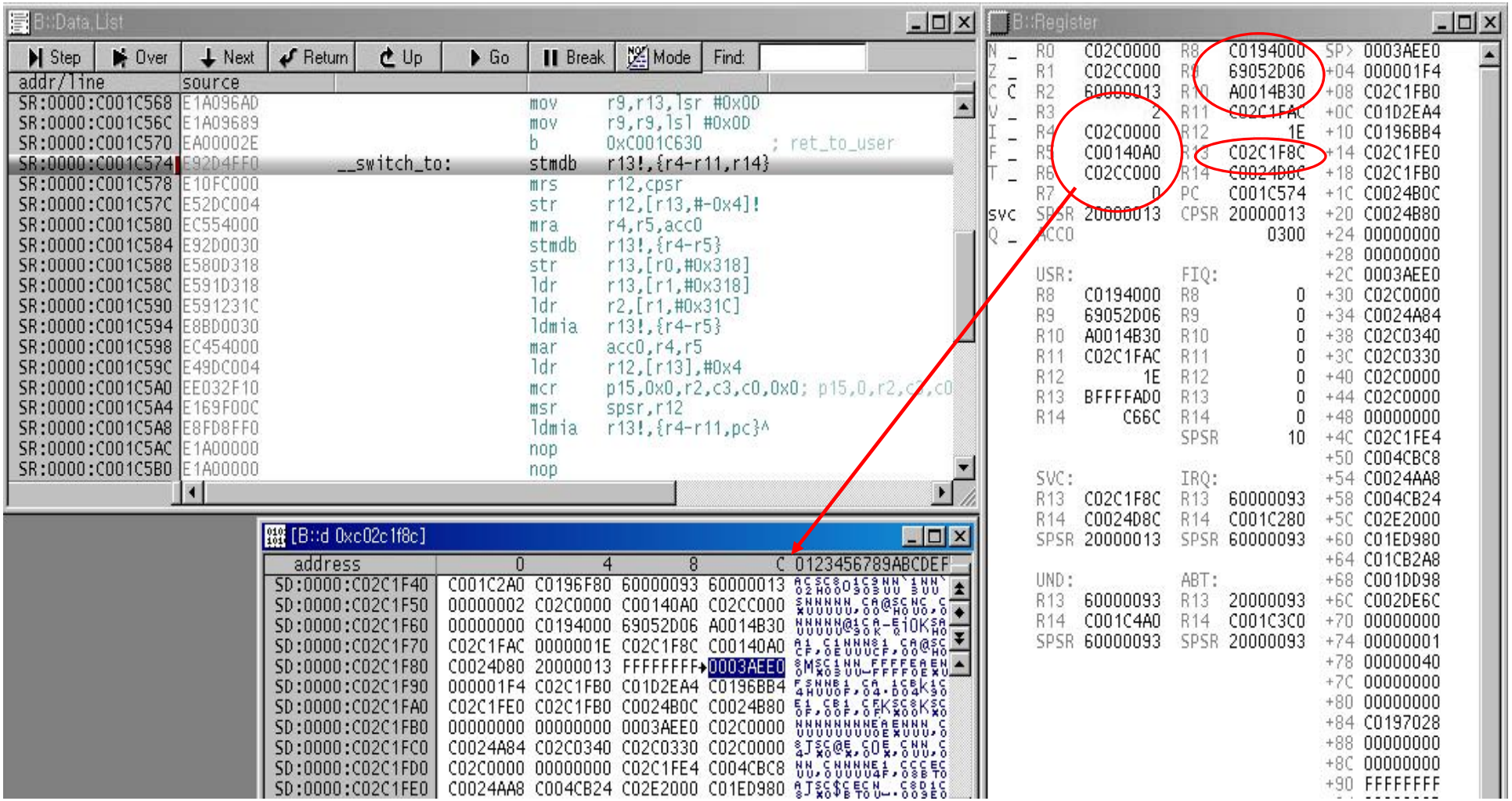
__switch_to() in ARM (1/16)



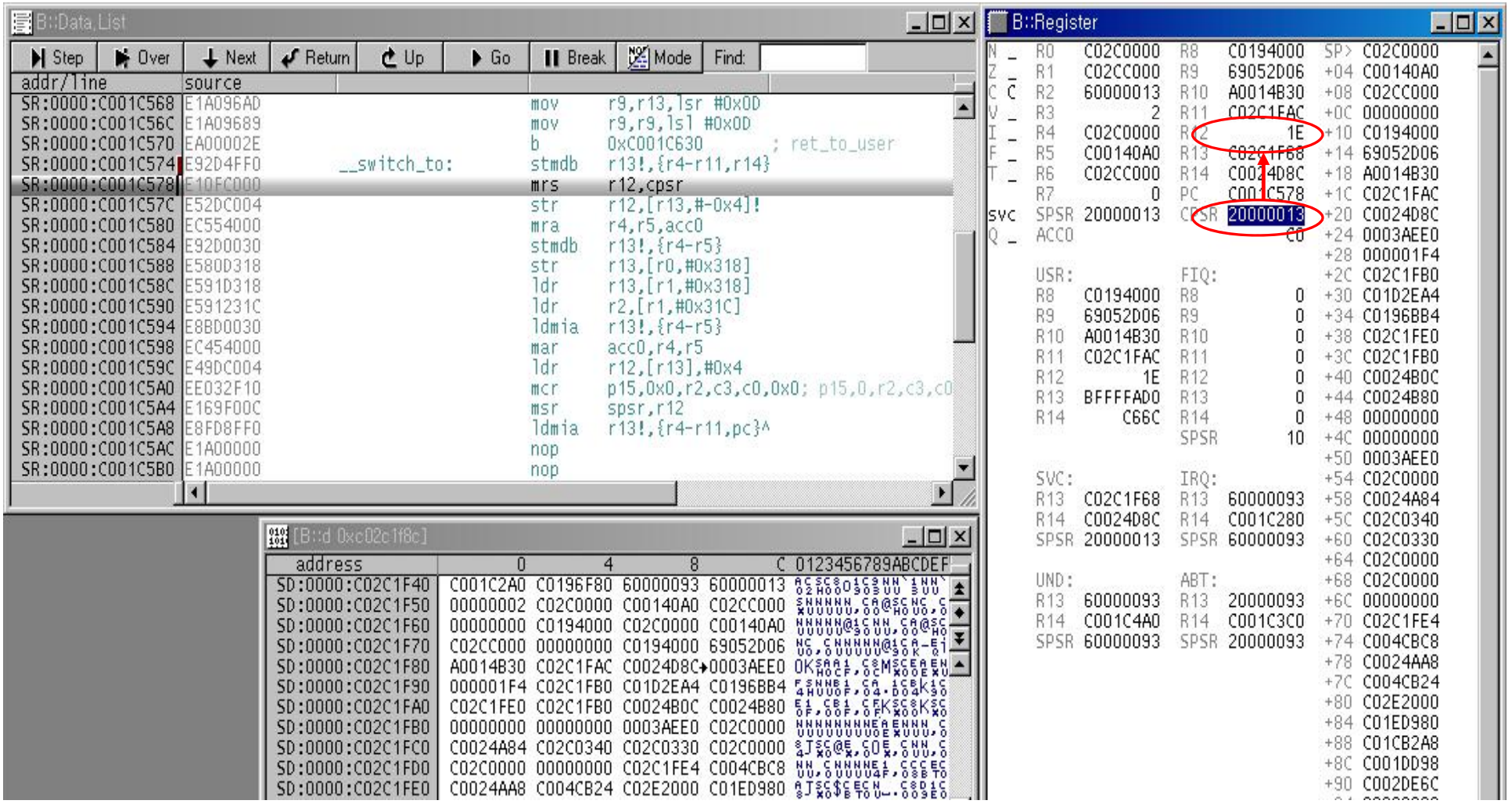
Memory Dump Window

CPU register window

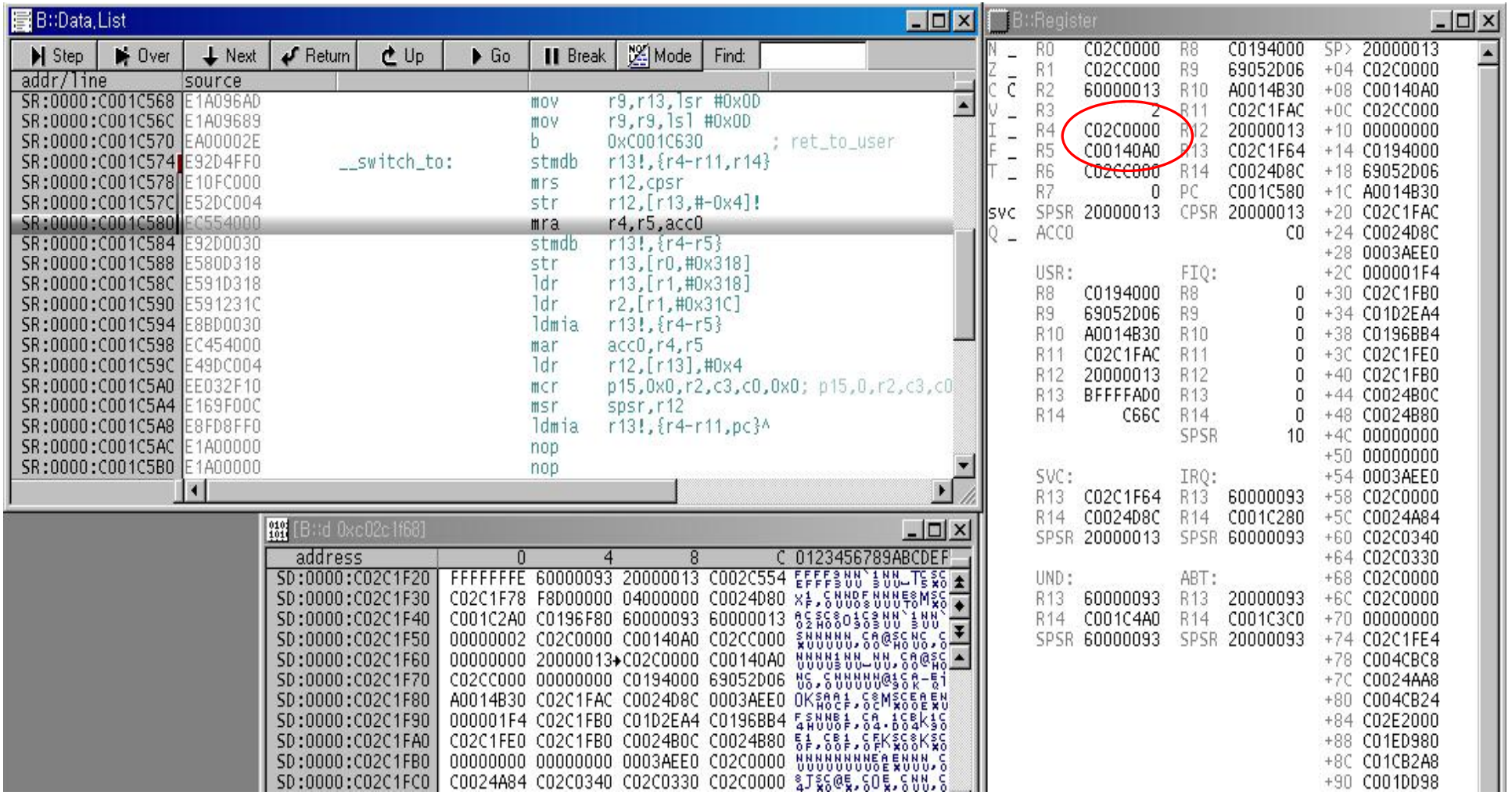
__switch_to() in ARM (2/16)



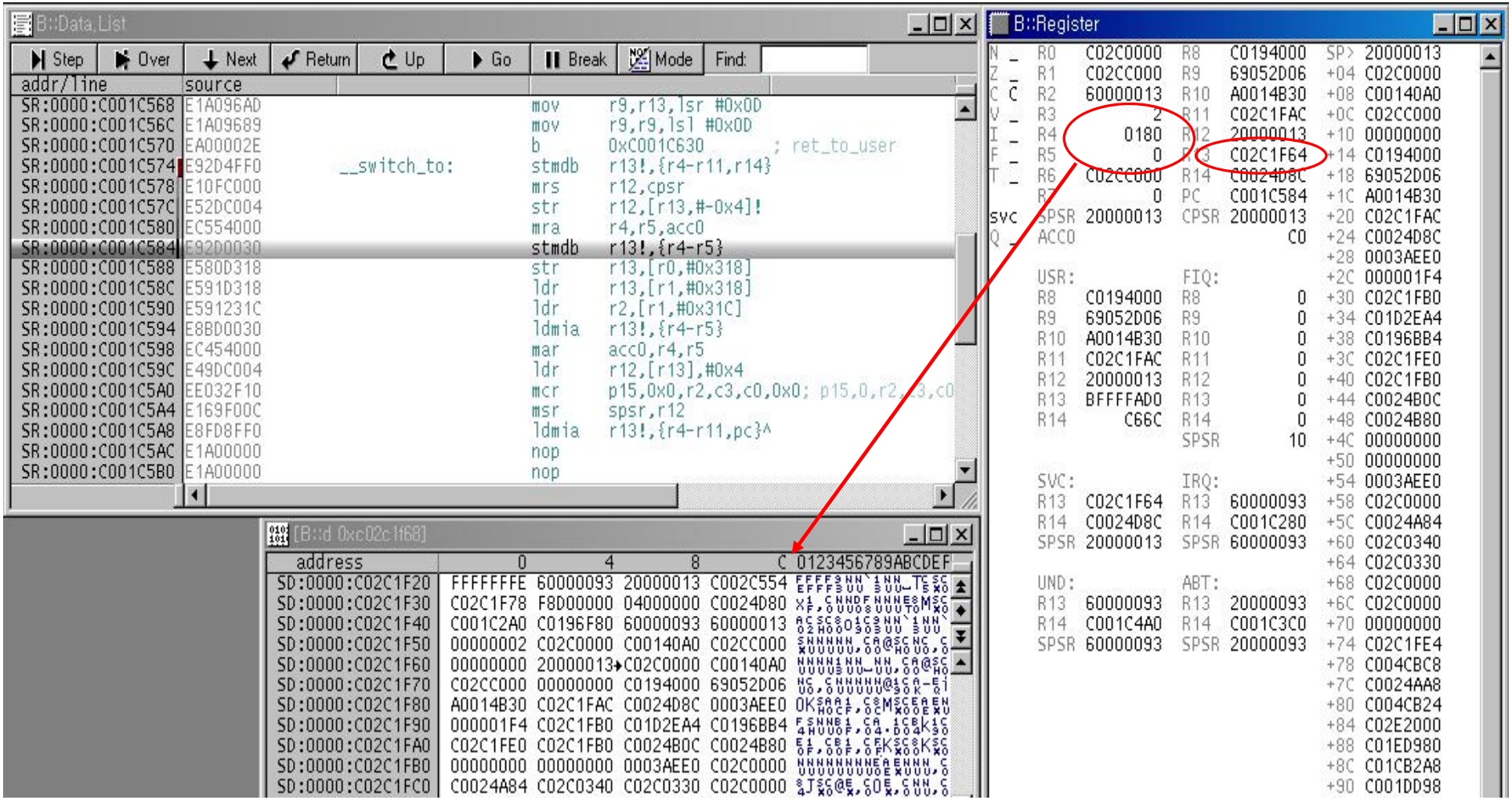
__switch_to() in ARM (3/16)



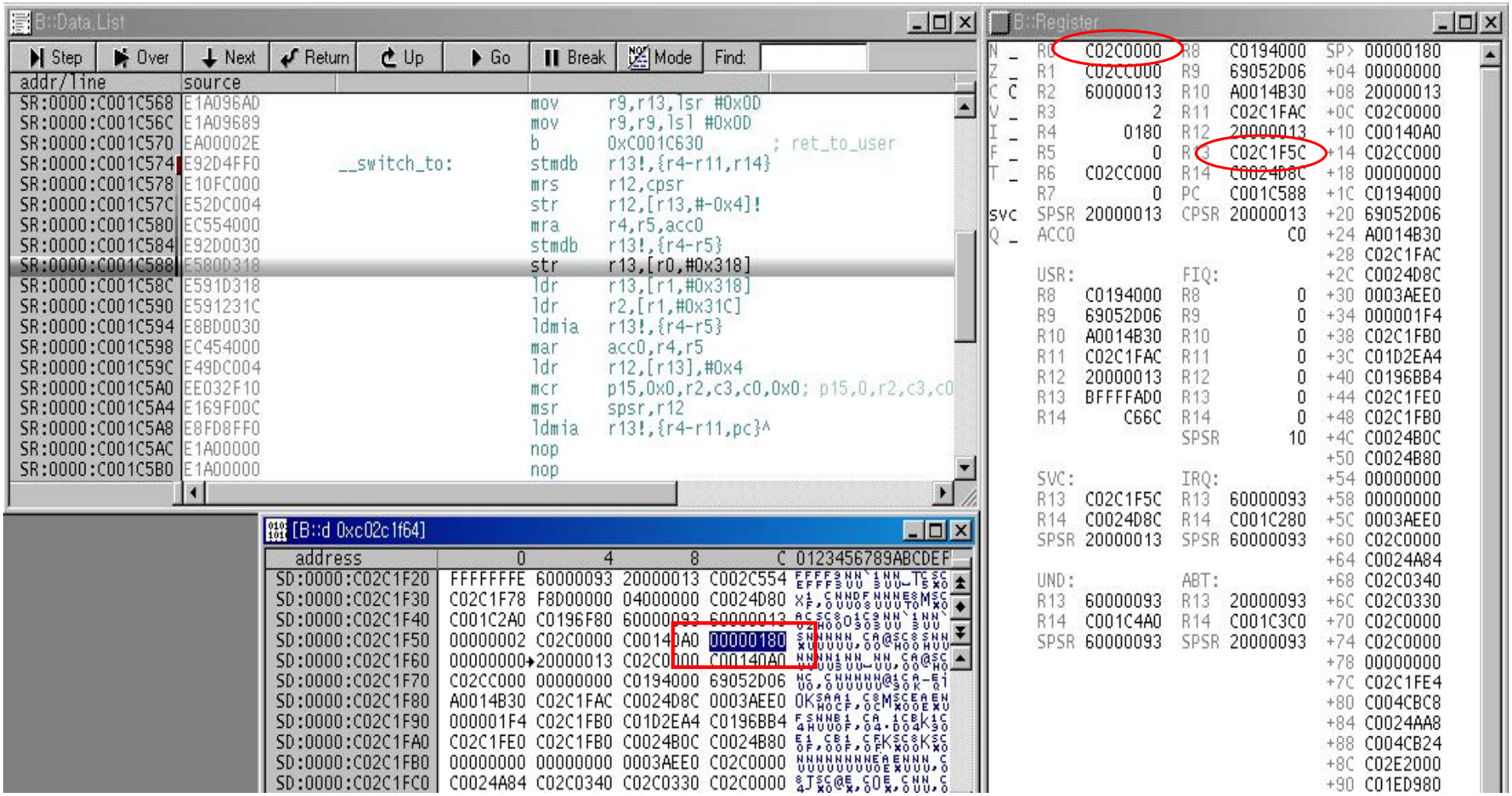
__switch_to() in ARM (4/16)



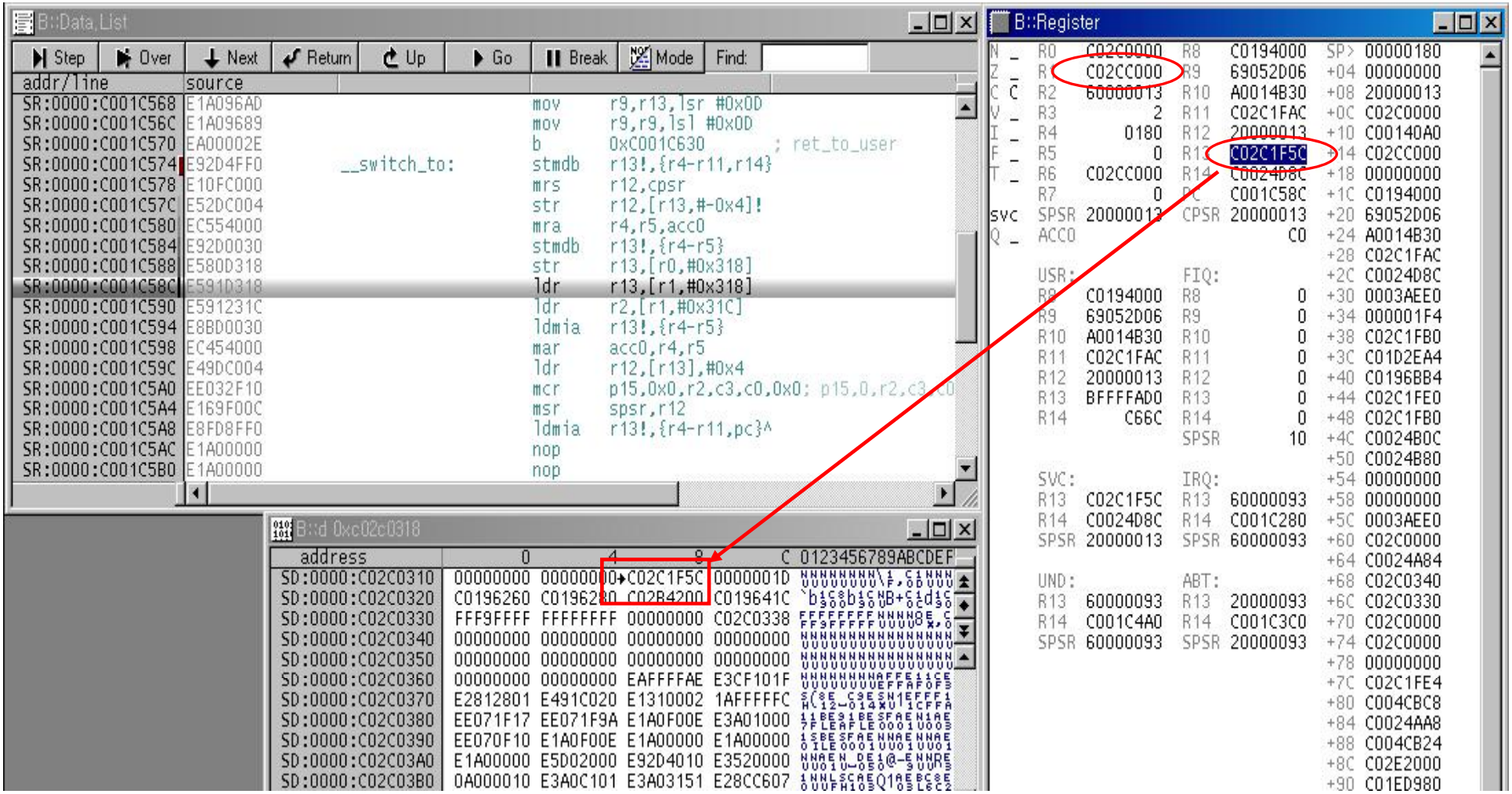
__switch_to() in ARM (5/16)



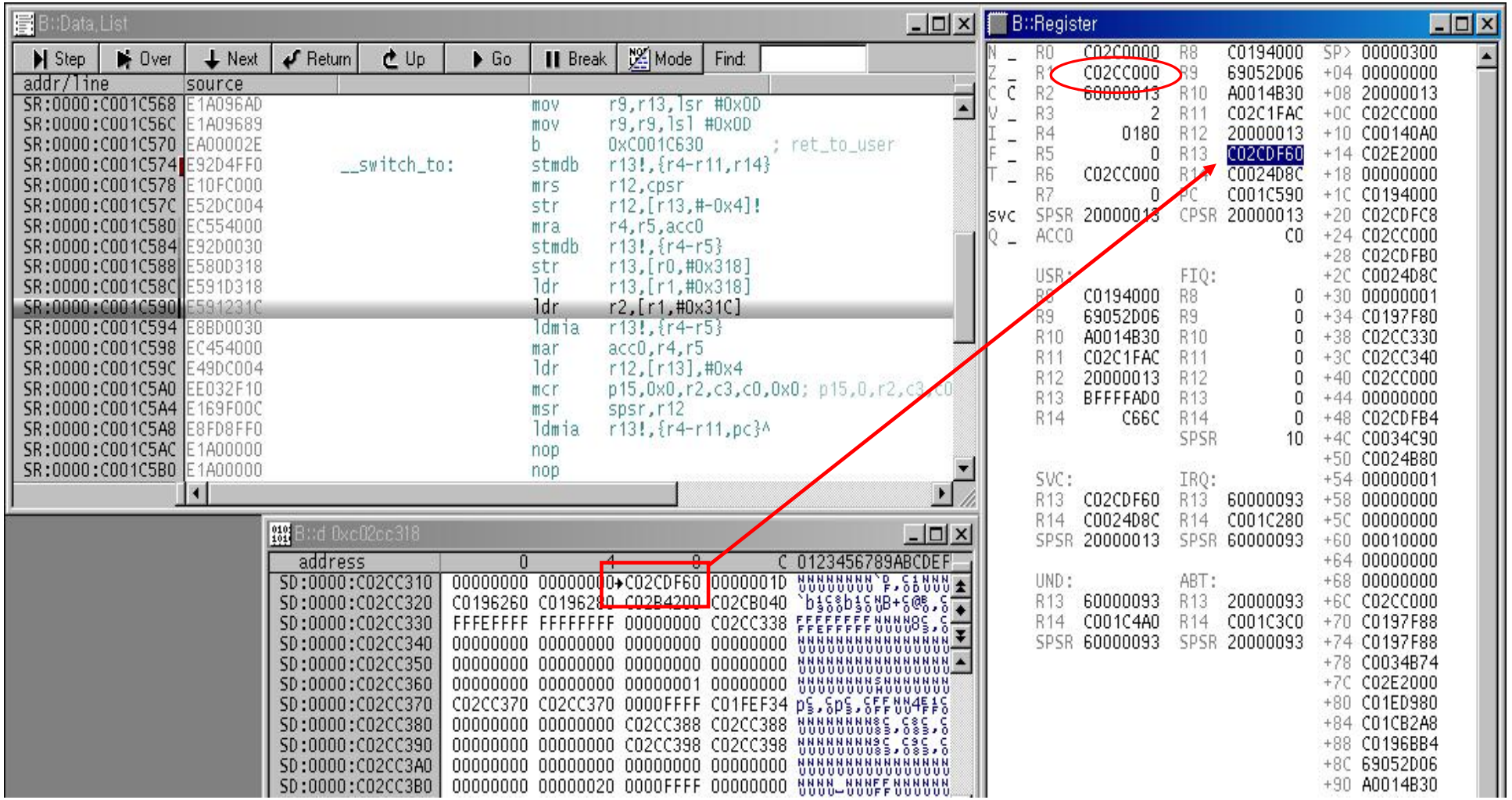
__switch_to() in ARM (6/16)



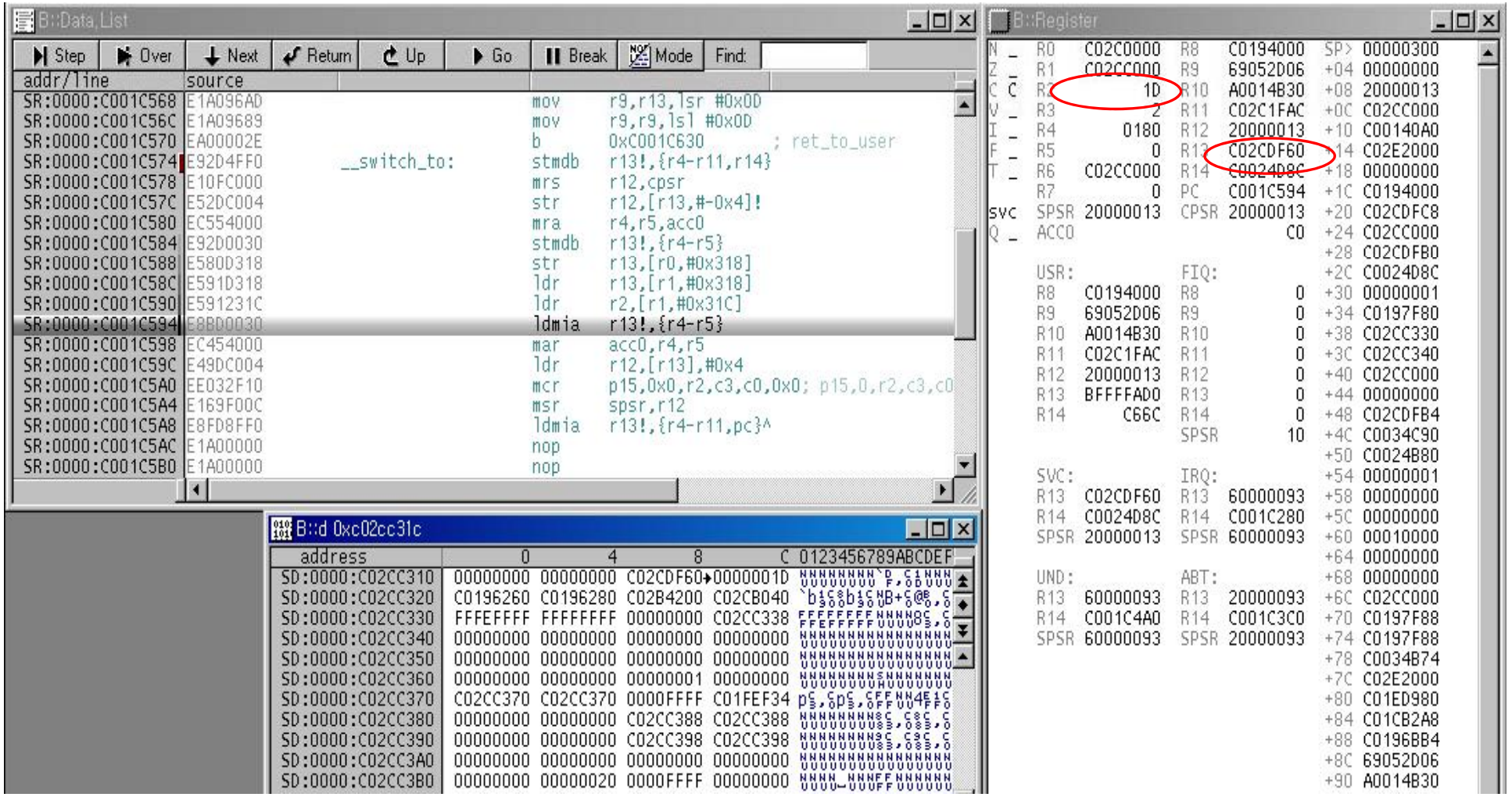
__switch_to() in ARM (7/16)



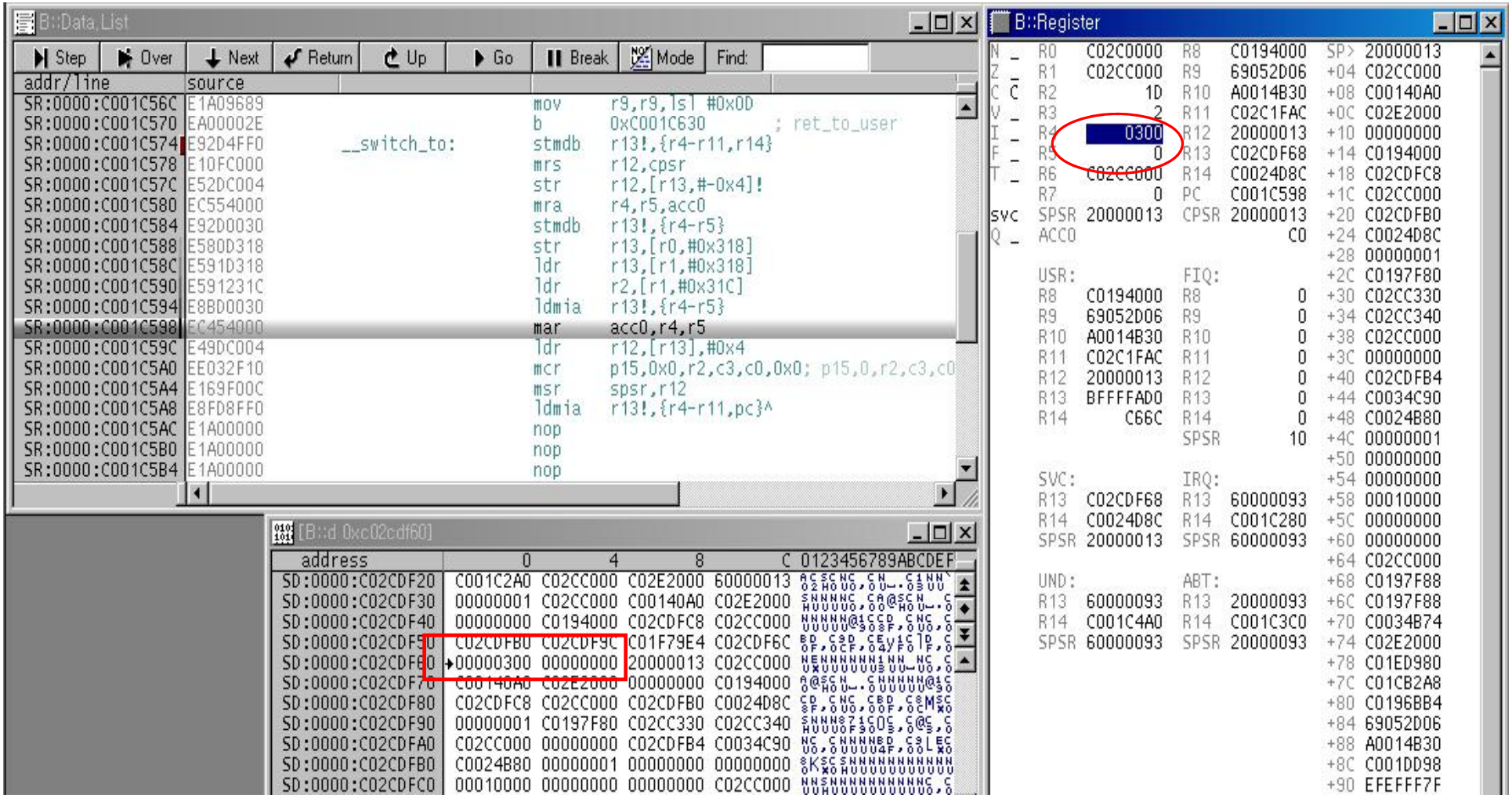
__switch_to() in ARM (8/16)



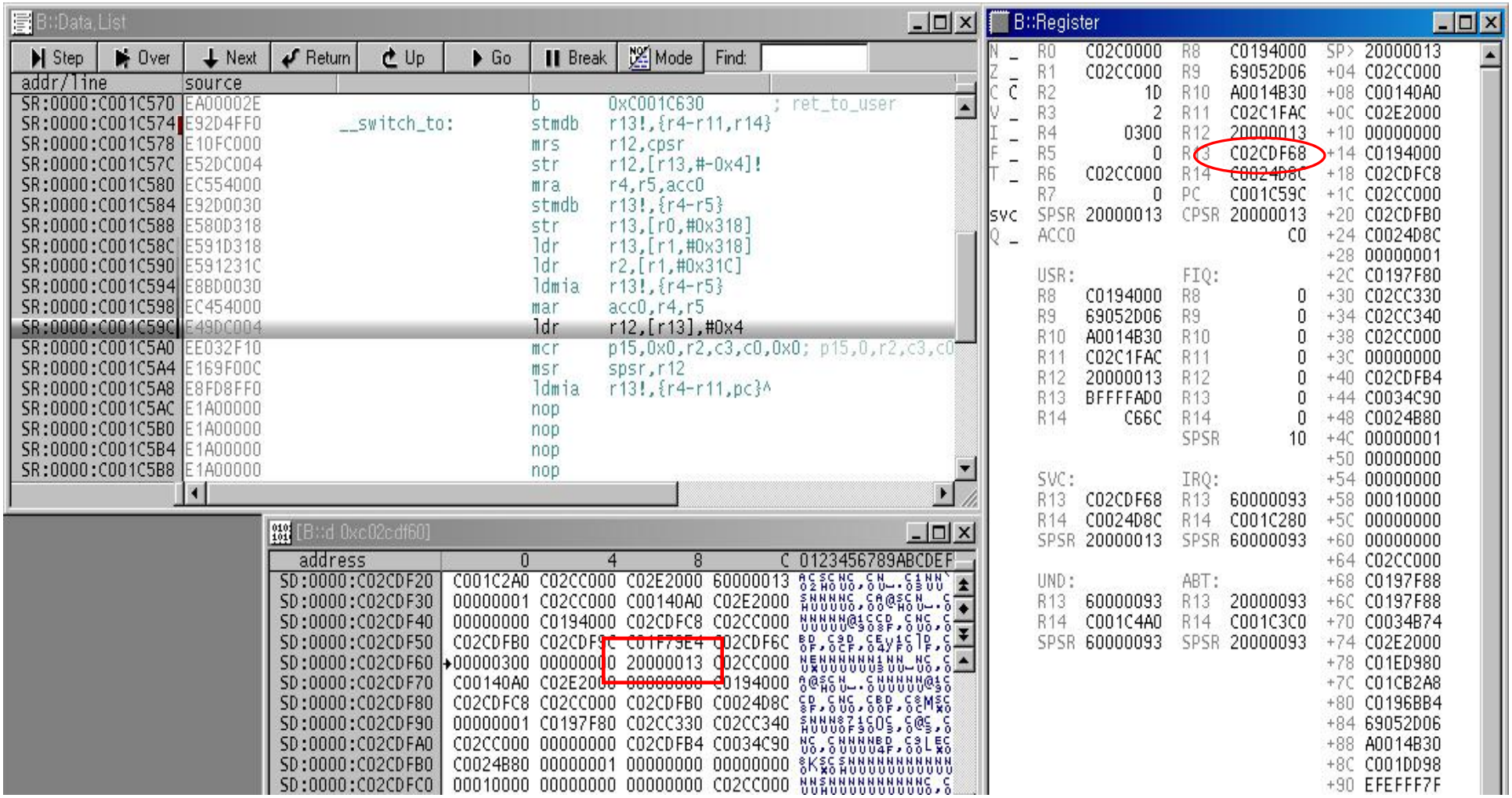
__switch_to() in ARM (9/16)



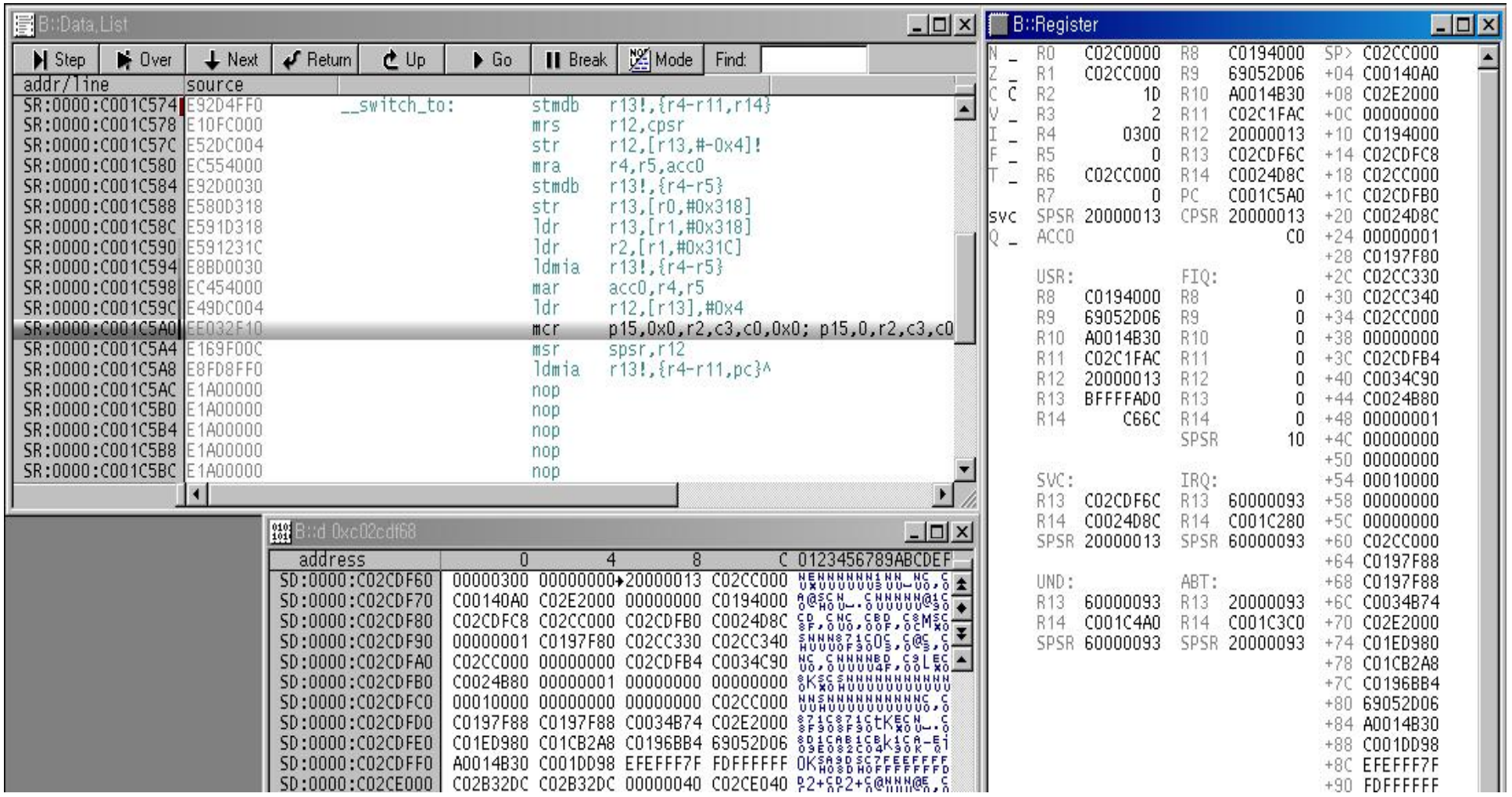
__switch_to() in ARM (10/16)



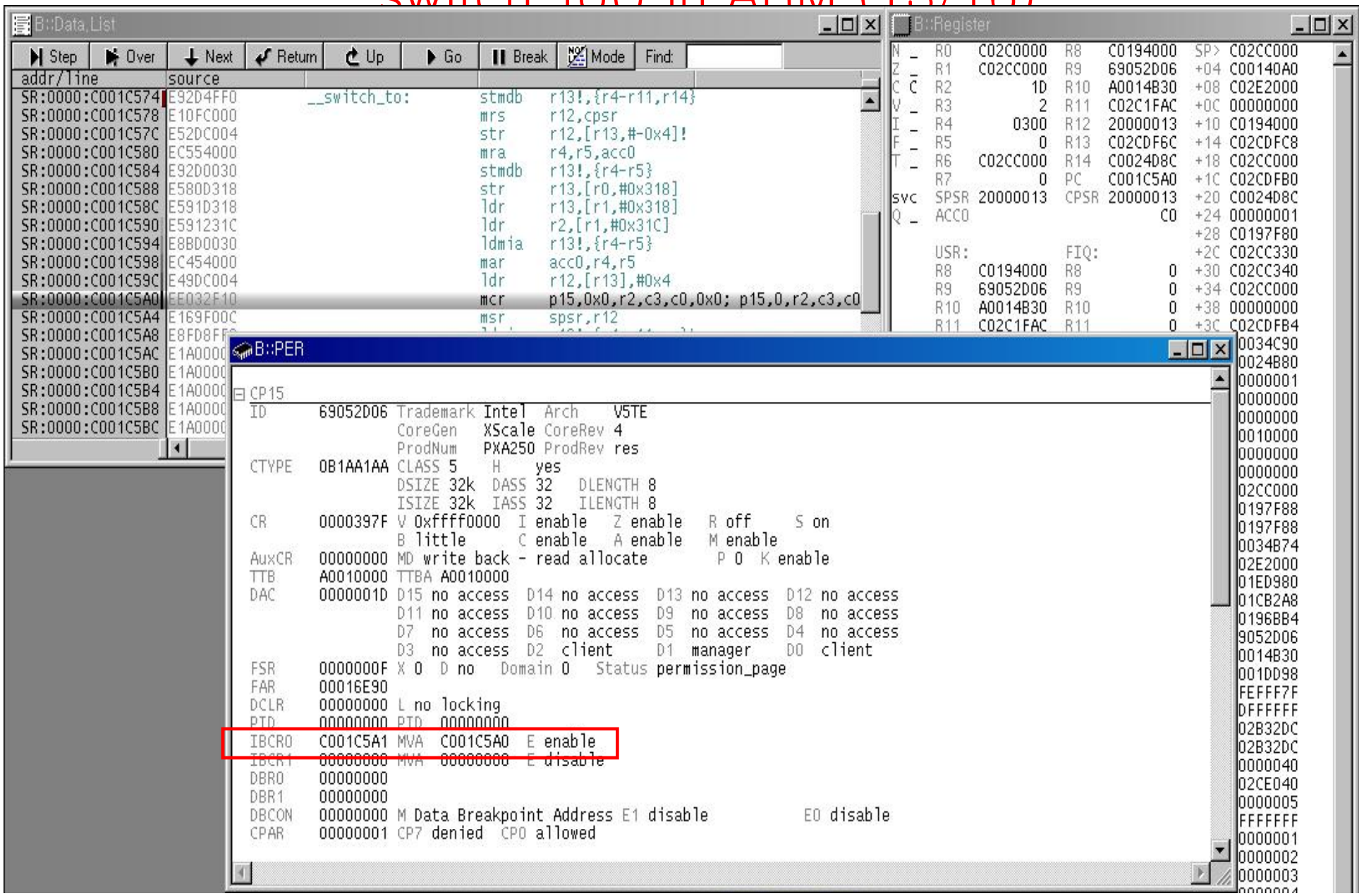
__switch_to() in ARM (11/16)



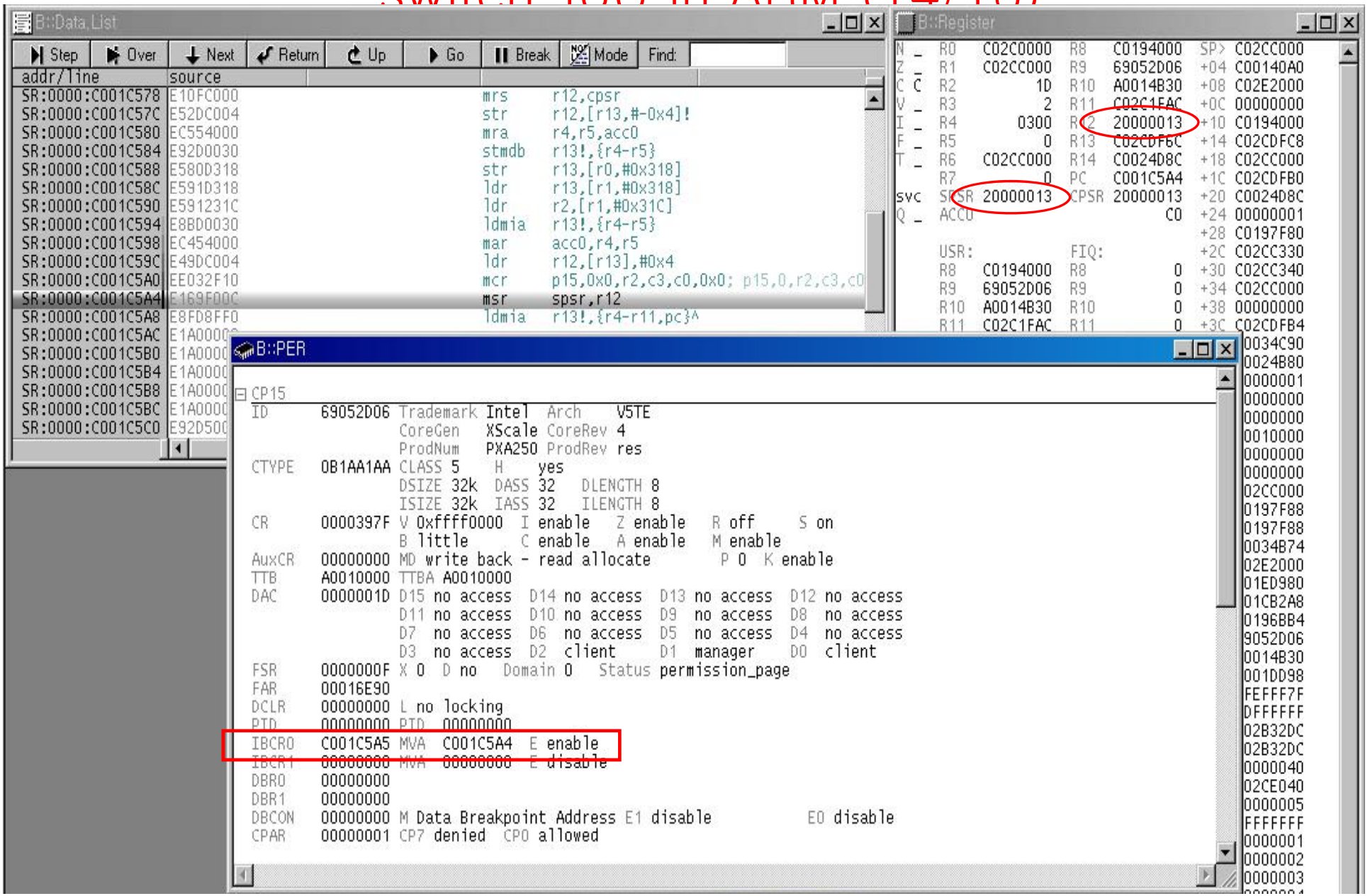
__switch_to() in ARM (12/16)



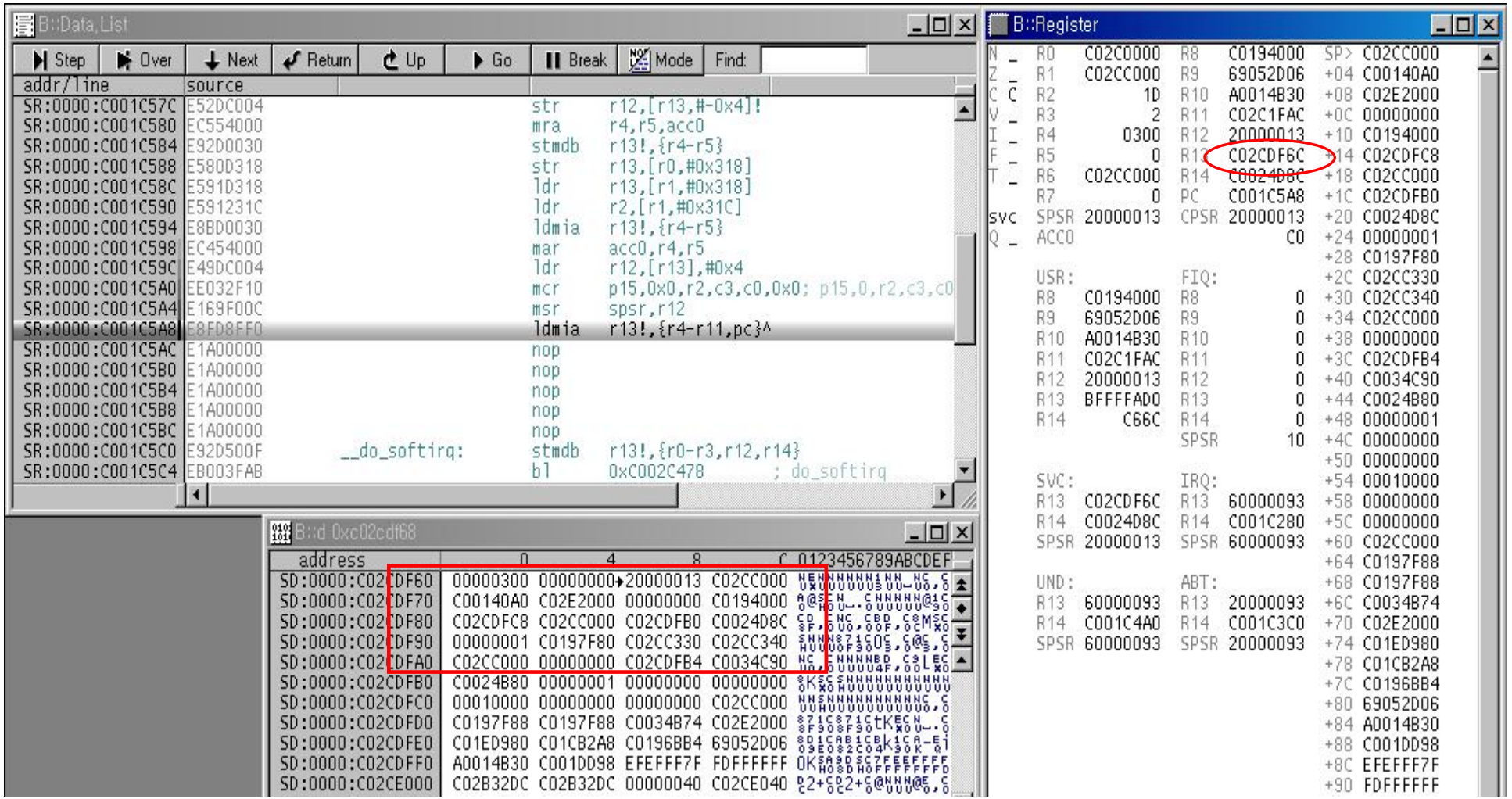
switch_to() in ARM (13/16)



switch_to() in ARM (14/16)



__switch_to() in ARM (15/16)



__switch_to() in ARM (16/16)