

# Linux Task Management

단국대학교

컴퓨터학과

2009

백승재

[ibanez1383@dankook.ac.kr](mailto:ibanez1383@dankook.ac.kr)

<http://embedded.dankook.ac.kr/~ibanez1383>

## 강의 목표

---

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

## ■ Process

- ✓ 실행 상태에 있는 프로그램의 instance ...
- ✓ 자원 소유권의 단위

## ■ Thread

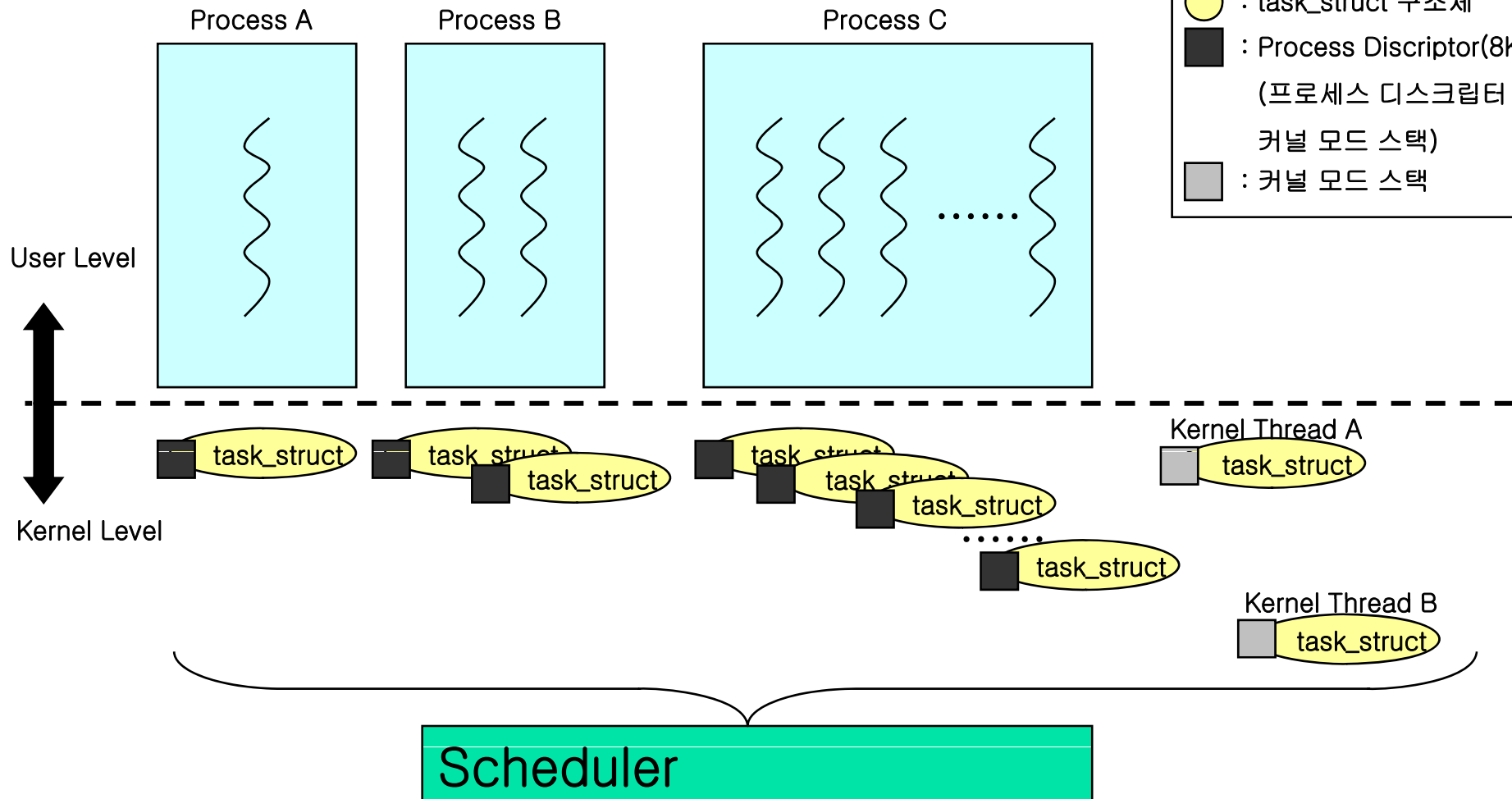
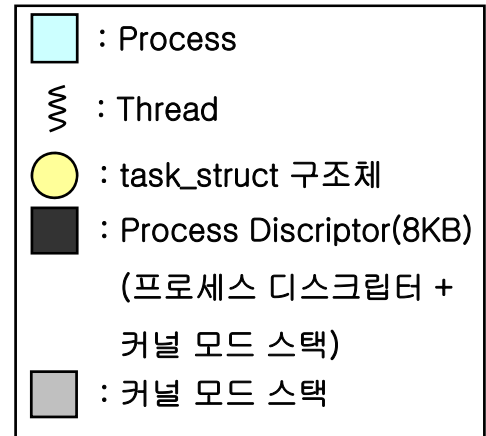
- ✓ 디스패칭의 단위, 실행 흐름 ...
- ✓ 수행의 단위

## ■ In Linux source code

- ✓ Process is Task and Thread is also Task

# Linux Task Model

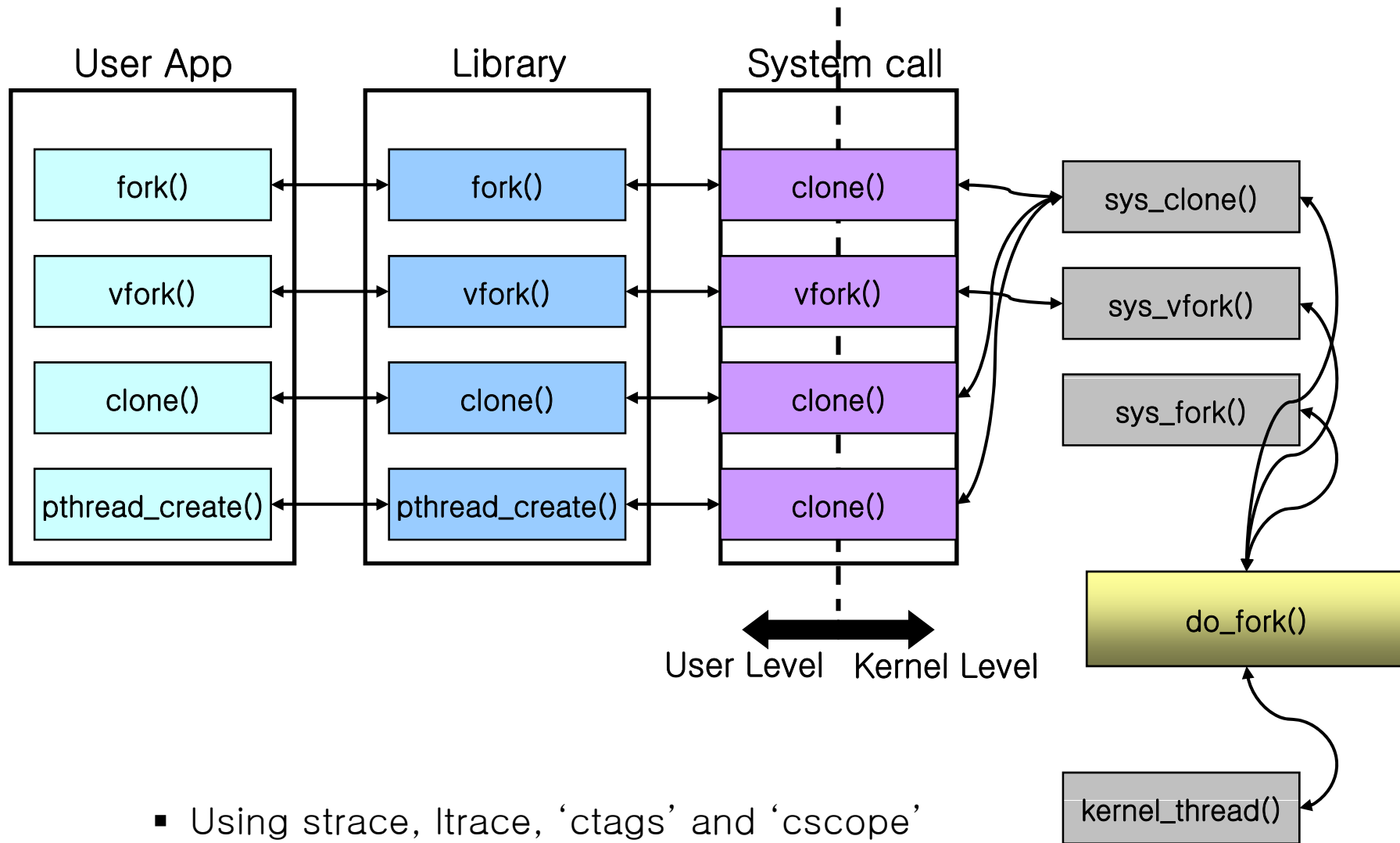
## Internal structure





# Task 관련 함수 흐름

## ■ Flow controls



# 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)

7

```
_syscall0(pid_t, getpid);

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(), getpid(), 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(), getpid());
    return 0;
}
```

## Thread example(3/5)

---

8

```
[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 설정할 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)

## ■ LinuxThreads vs NPTL

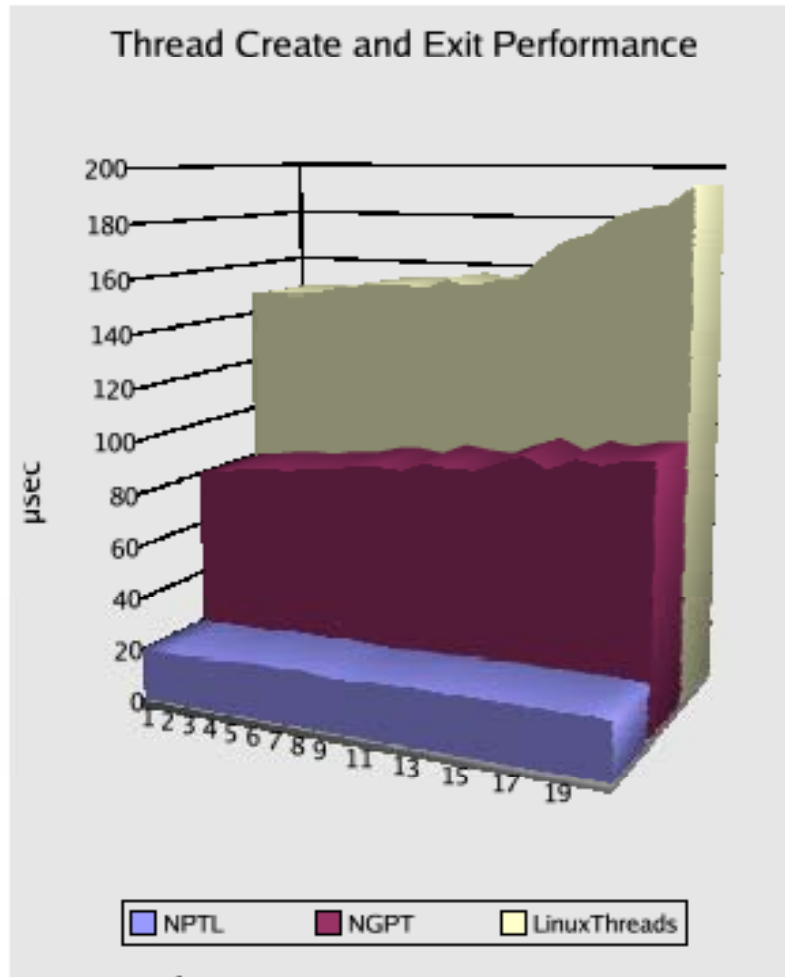


Figure 1: Varying number of Toplevel Threads

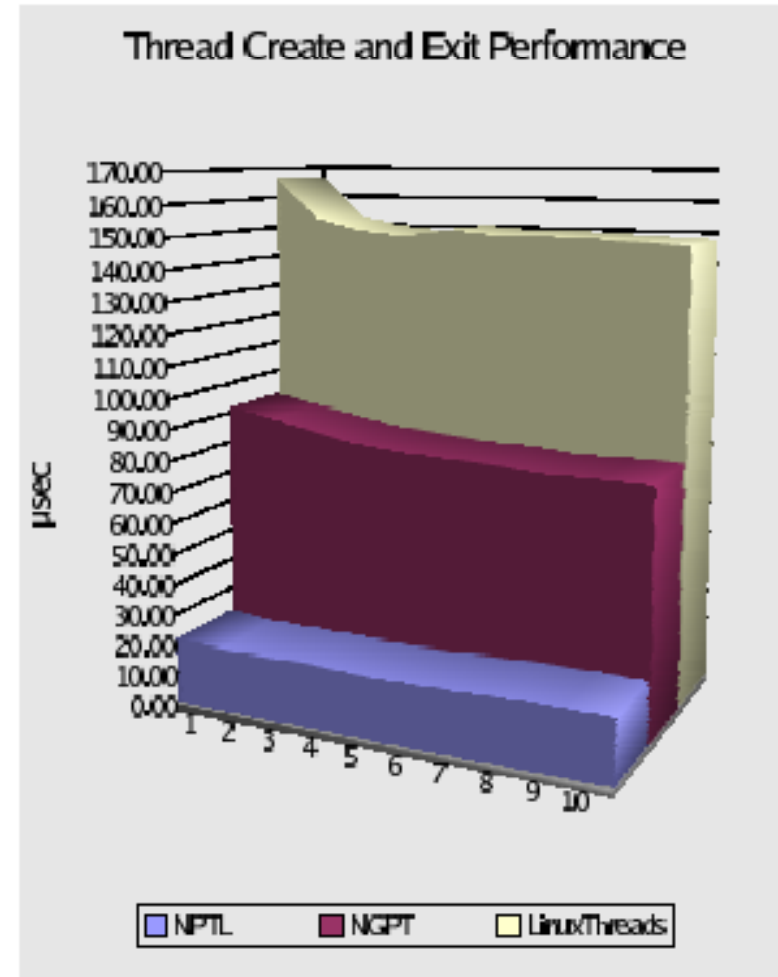


Figure 2: Varying number of Concurrent Children

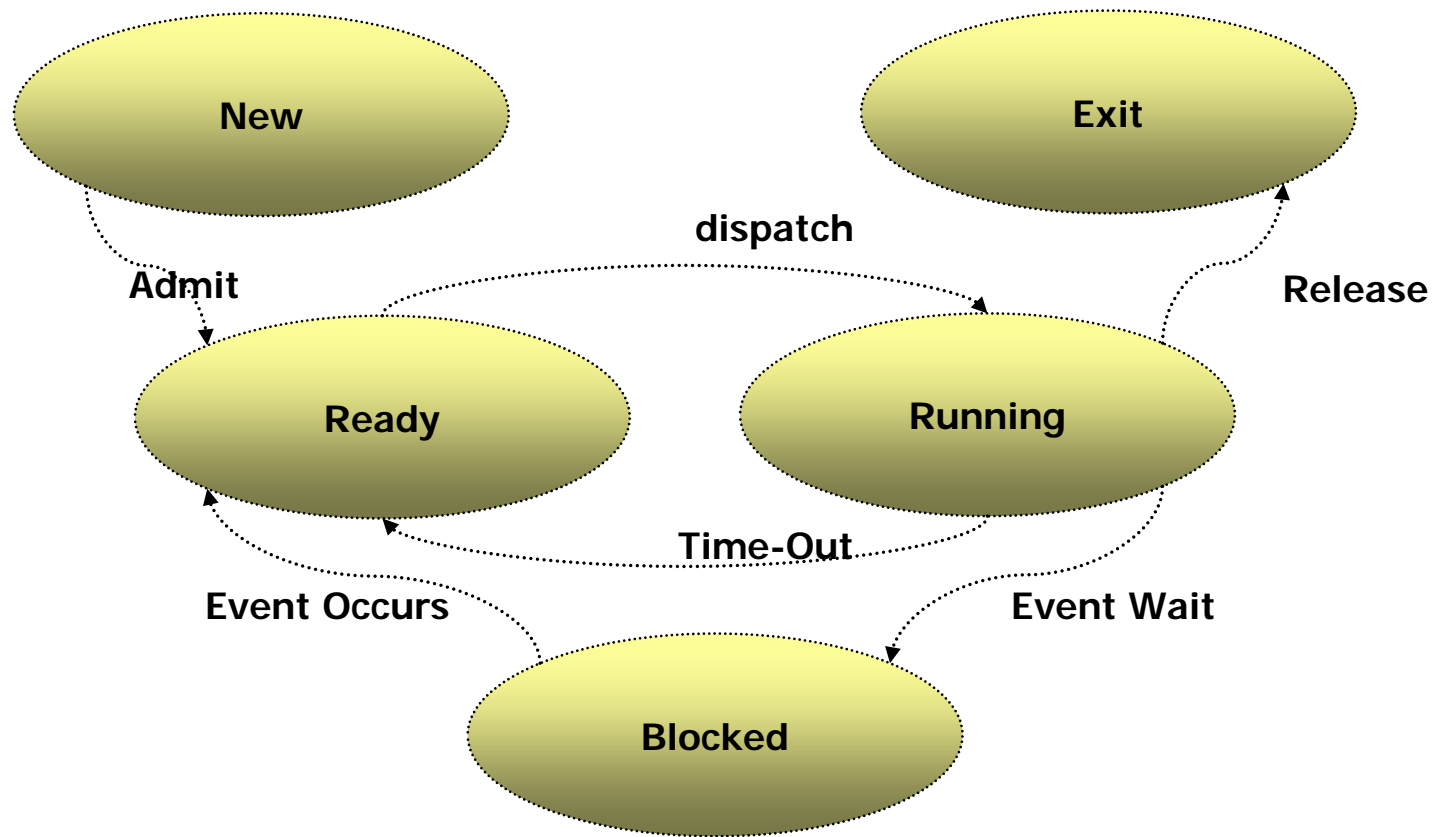


## ■ Task는

- ✓ 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)
```

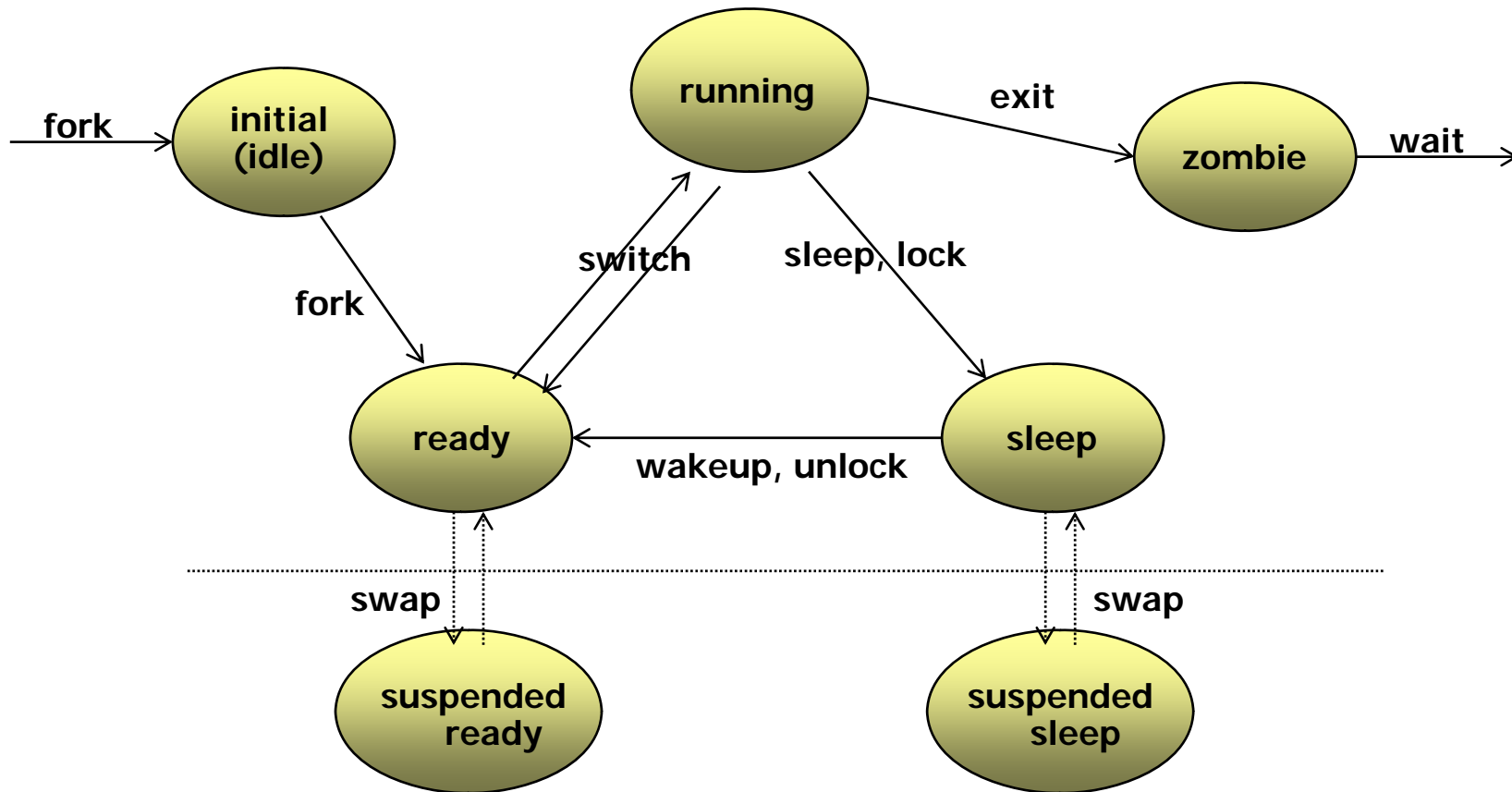
# Process State Diagram(1/2)



Five-State Process Model

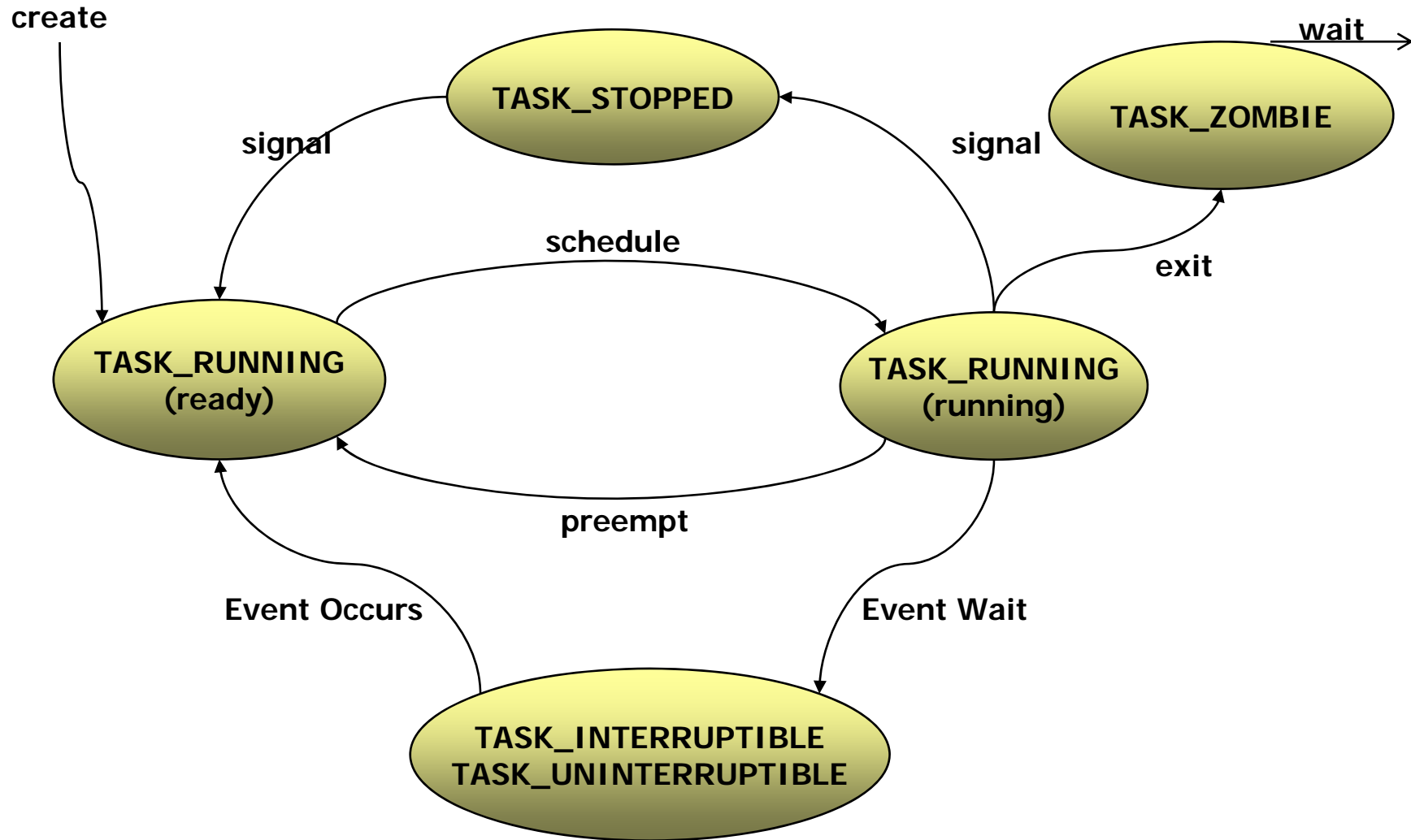
(Source : OS(stallings))

# Process State Diagram(2/2)

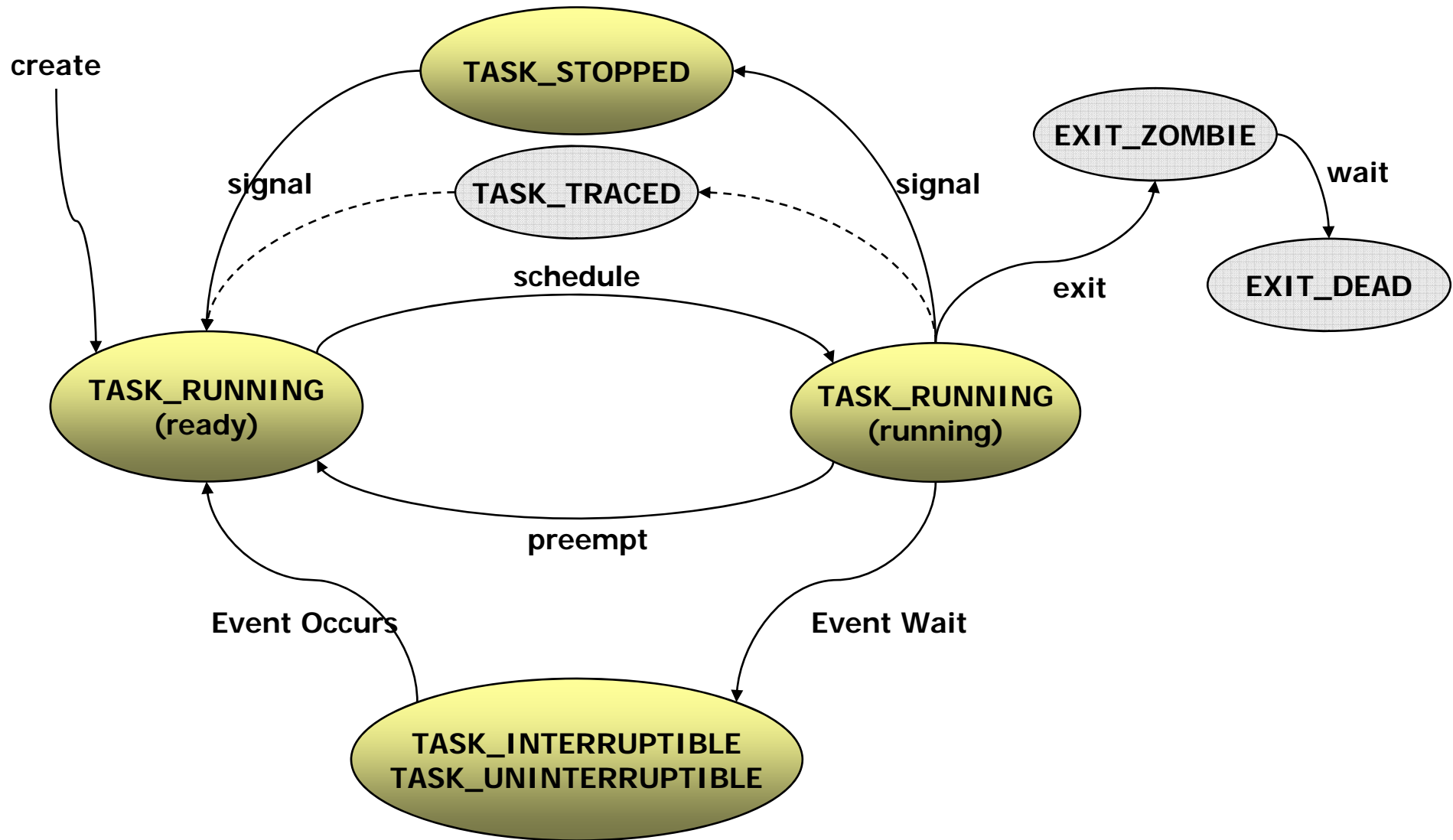


(Source : UNIX Internals)

# Task State Diagram of LINUX(2.4)



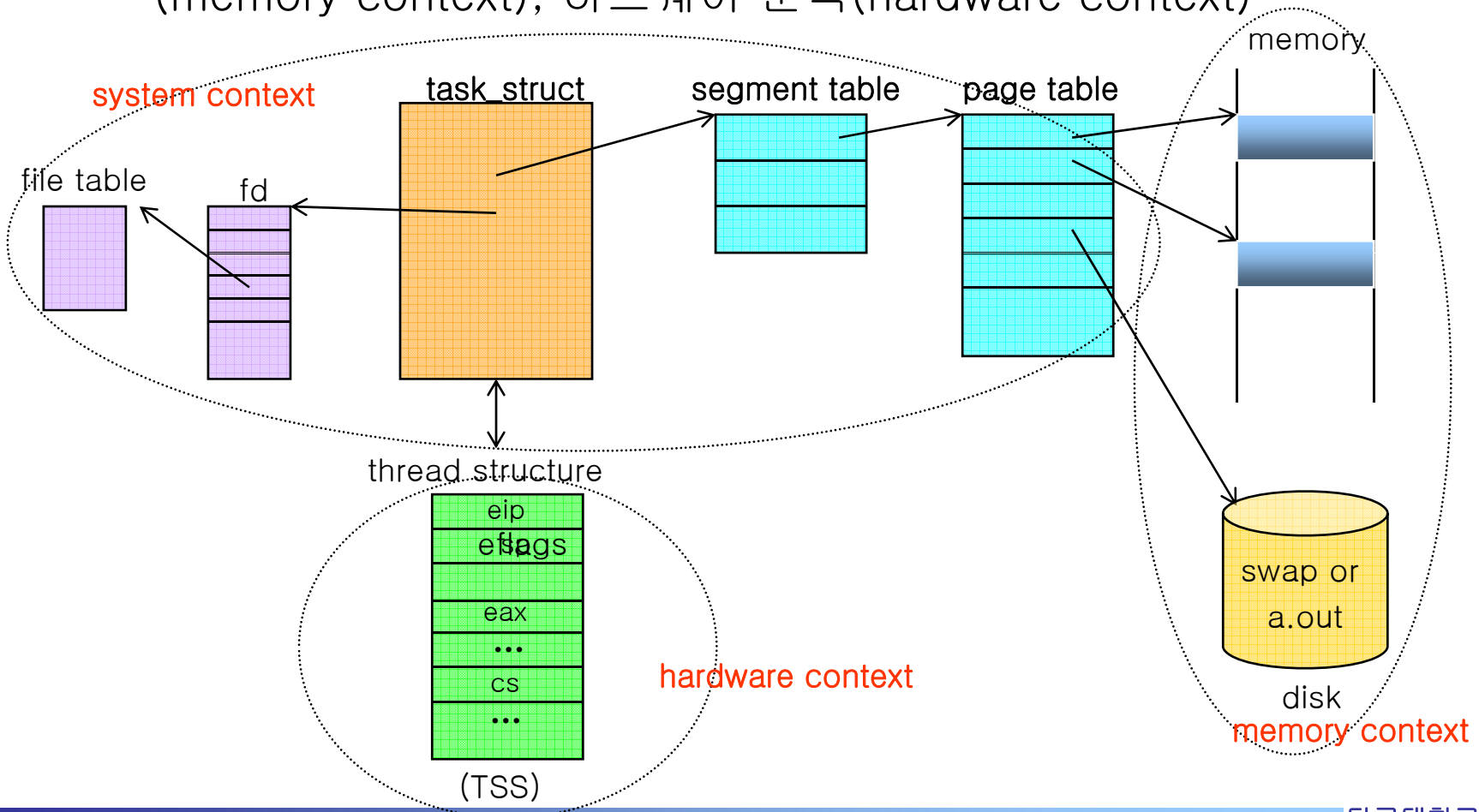
# Task State Diagram of LINUX(2.6)



# Context

## ■ 문맥 (Context) :

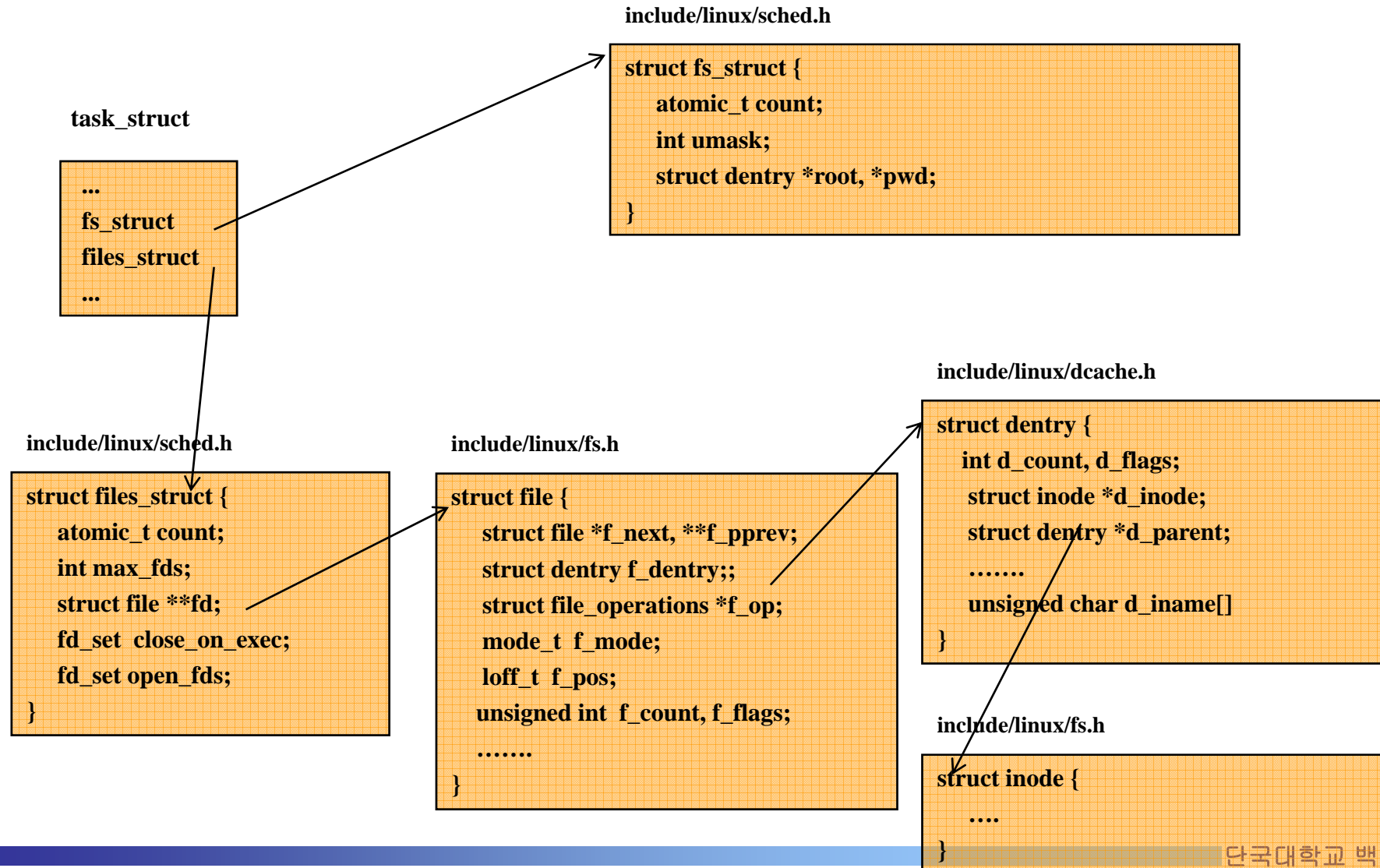
- ✓ 커널이 관리하는 태스크의 자원과 수행 환경 집합
- ✓ 3 부분으로 구성 : 시스템 문맥 (system context), 메모리 문맥 (memory context), 하드웨어 문맥 (hardware context)



- 태스크를 관리하는 정보 집합
  - ✓ 태스크 정보: pid, uid, euid, suid, ...
  - ✓ 태스크 상태: 실행 상태, READY 상태, 수면 상태, ...
  - ✓ 태스크의 가족 관계: p\_pptr, p\_cptra, 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)

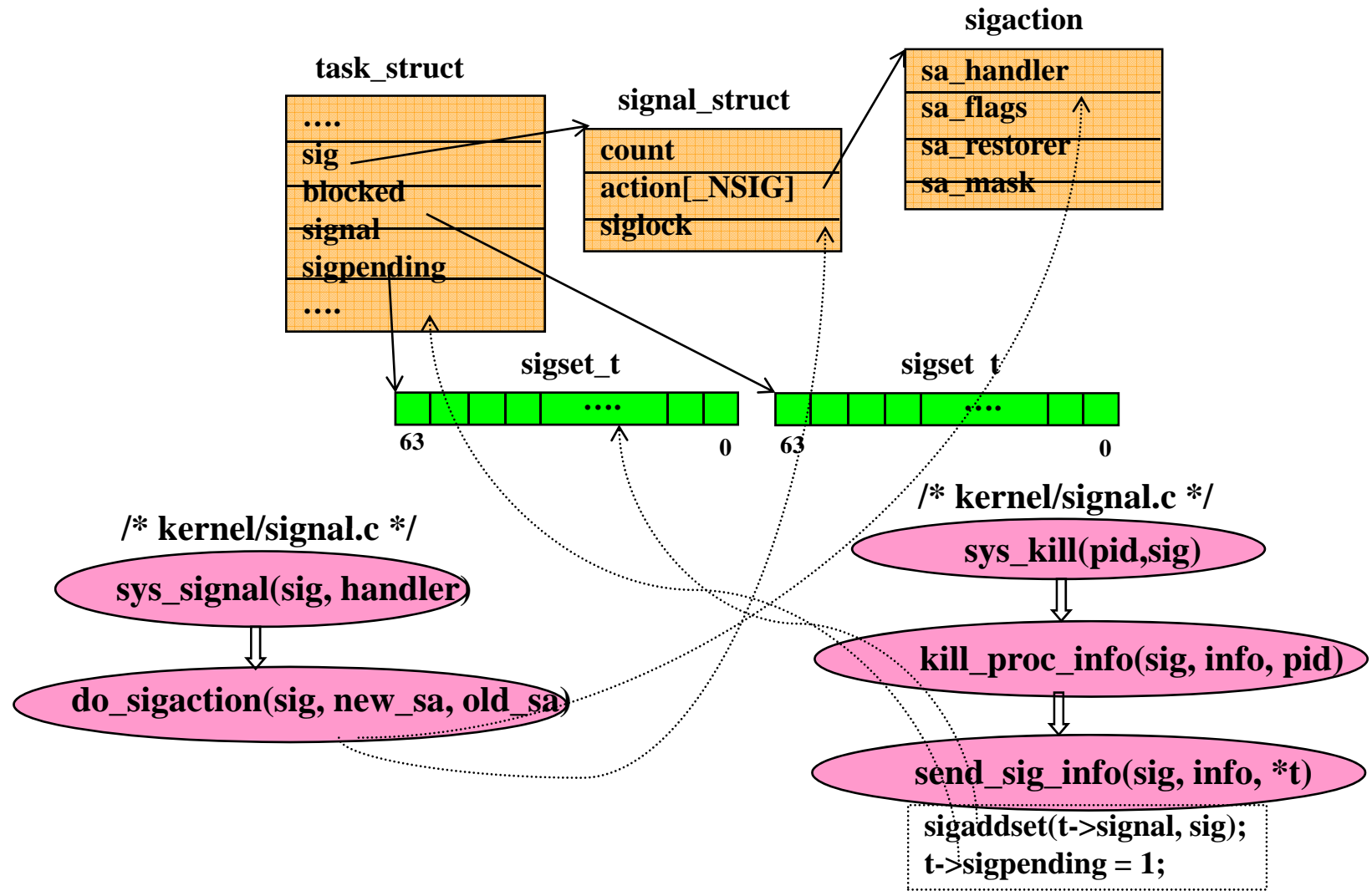
# Task와 file

## ■ 파일 관리 자료 구조 : fd, inode

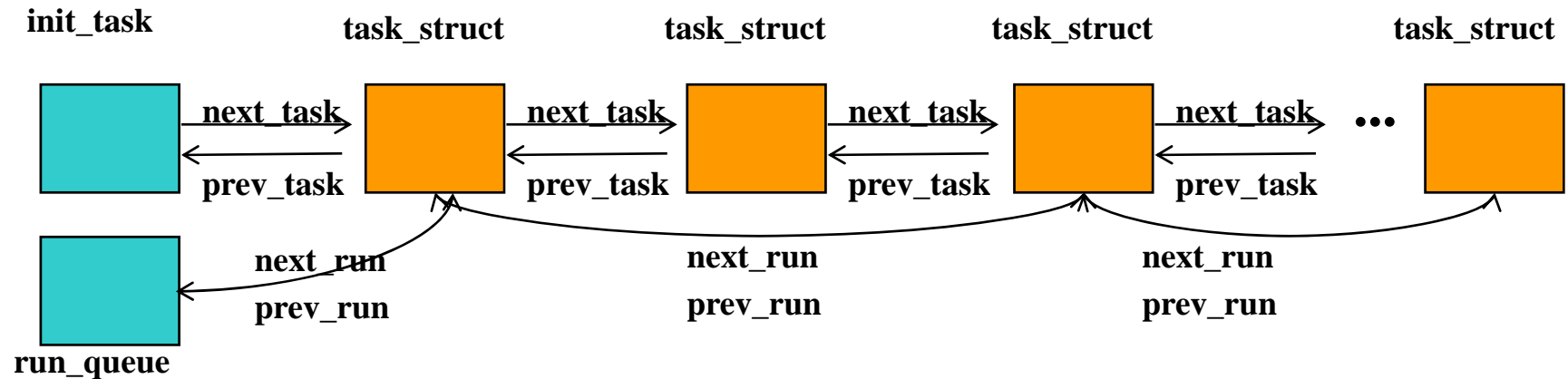




■ 시그널 처리 자료 구조

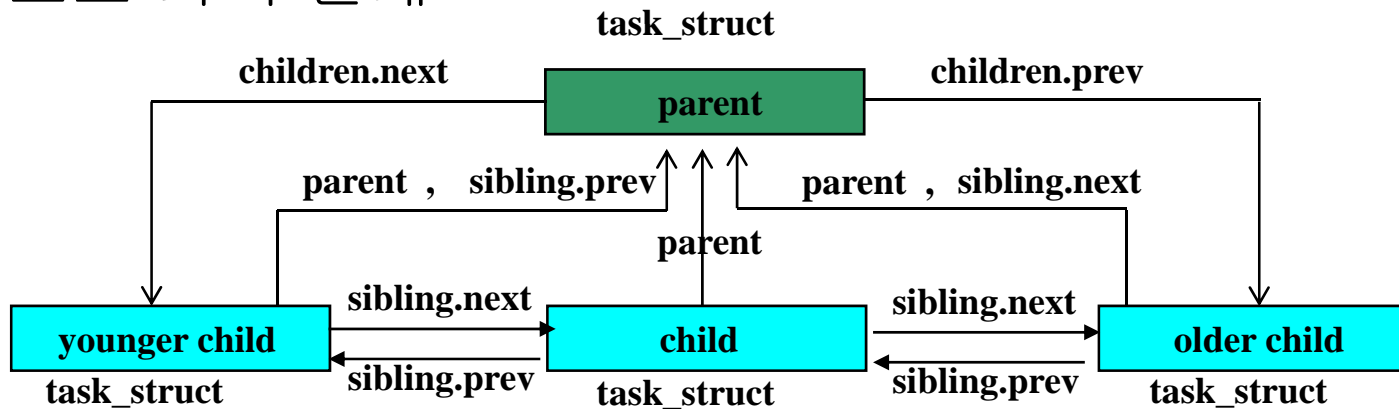


## ■ 태스크 연결 구조



where is run\_queue ? prev\_task? next\_task?

## ■ 태스크 가족 관계



# Task 계층 구조 확인(1/2)

```
embeddedDell:177 - Xmanager 2.0 [:0.0]
root@embeddedDell:~
[root@ez-x5 /root]$ ./nfs/sched

Before set
Param.priority = 0
Sched policy = 0

FIFO set
Param.priority = 10
Sched policy = 1
```

magic	command	state	uid	pid
C0194000	swapper	running	0.	0.
C02E2000	init	sleeping	0.	1.

Parent-Child relationships shown in red boxes:

- swapper (pid 0) is the parent of init (pid 1).
- init (pid 1) is the parent of rpciod.

# Task 계층 구조 확인(2/2)

The screenshot displays two windows from Windows Task Manager, showing the task hierarchy for 'bash' and 'sched' processes. Red boxes highlight specific fields in the task lists.

**Window 1: B::Task, DTask 0xC0C54000**

magic	command	state	uid	pid	spaceid	tty	flags	nic
C0C54000	bash	sleeping	0.	104.	0068	S-71	00000100	

gid	sigpending	vm size	ttb	tty name	path
0.	00000000	00000100	C0C4C000	ttyS2	/bin/bash

[-] flags  
[-] parent      youngest child    younger sibling    older sibling  
init                            sched                            rpciod                            inetd

**Window 2: B::Task, DTask 0xC0C36000**

magic	command	state	uid	pid	spaceid	tty	flags	nic
C0C36000	sched	current	0.	125.	0070	S-71	00000100	

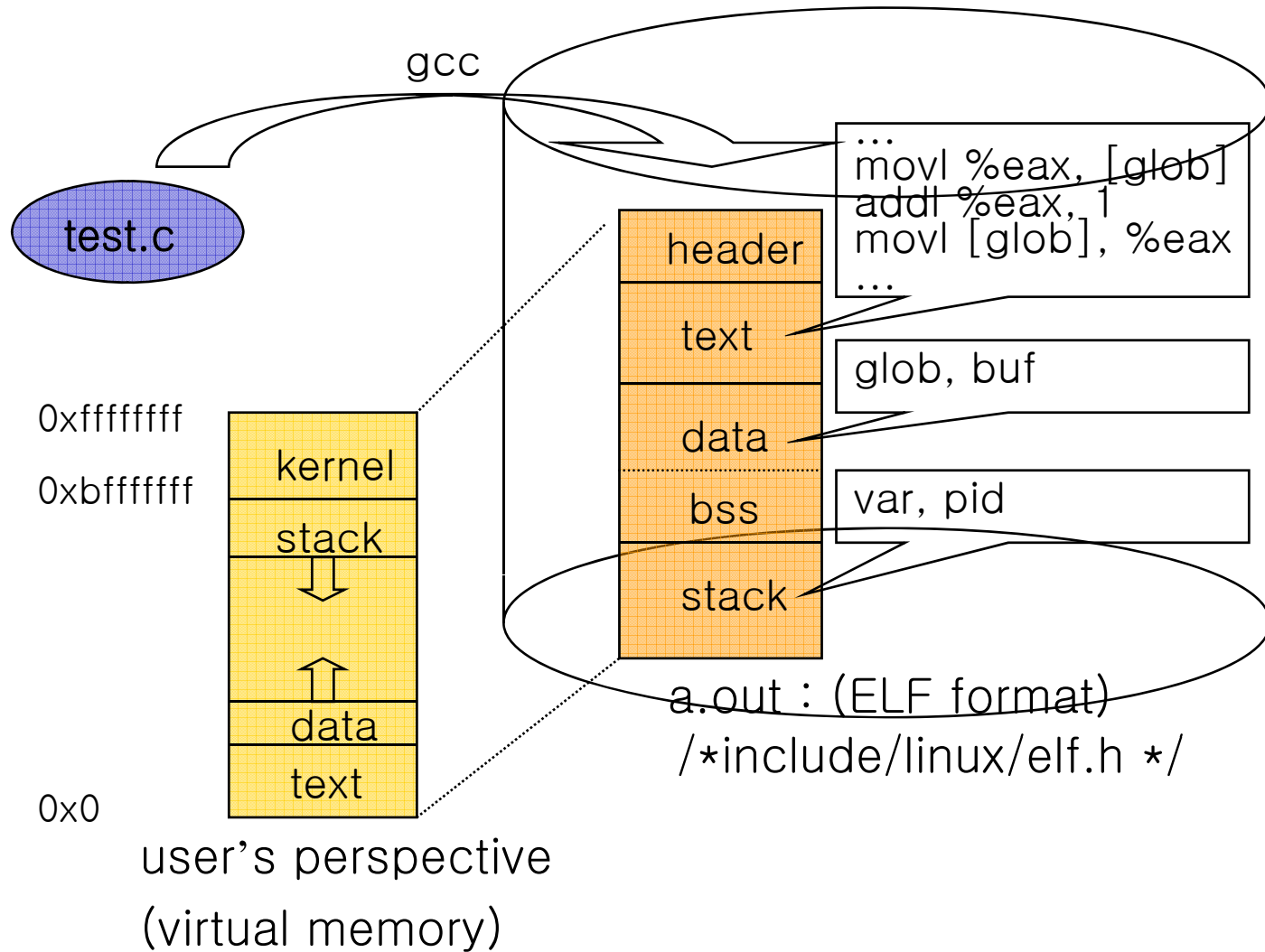
  

gid	sigpending	vm size	ttb	tty name	path
0.	00000000	00000141	C0C30000	ttyS2	./nfs/sched

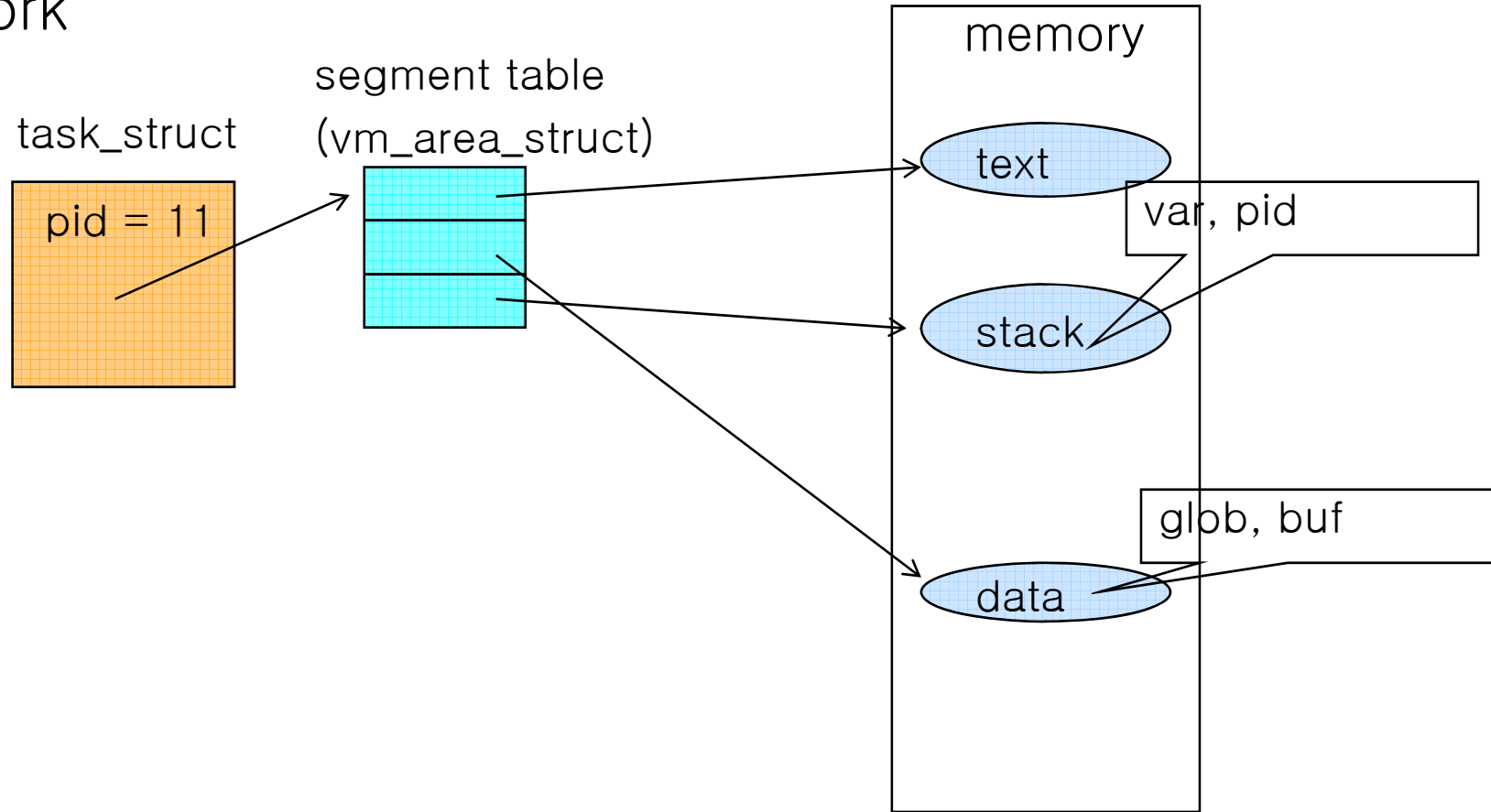
[+] flags  
[-] parent      youngest child    younger sibling    older sibling  
bash                            -                            -                            -

[+] arguments  
[+] environment  
[+] open files  
[+] addresses  
[+] code files  
[+] times

- fork internal : compile results

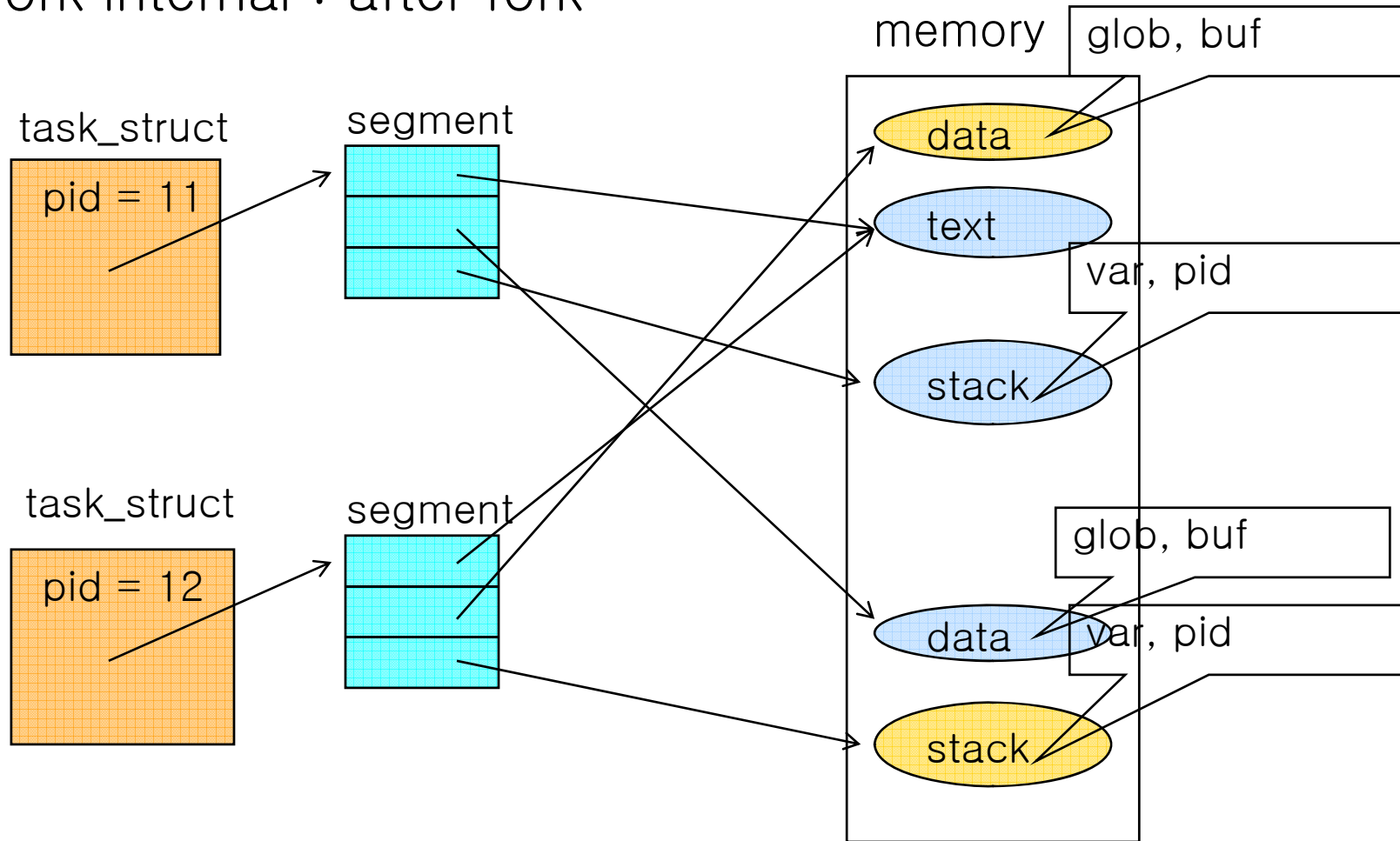


- fork internal : after loading (after run a.out) & before fork



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

- fork internal : after fork

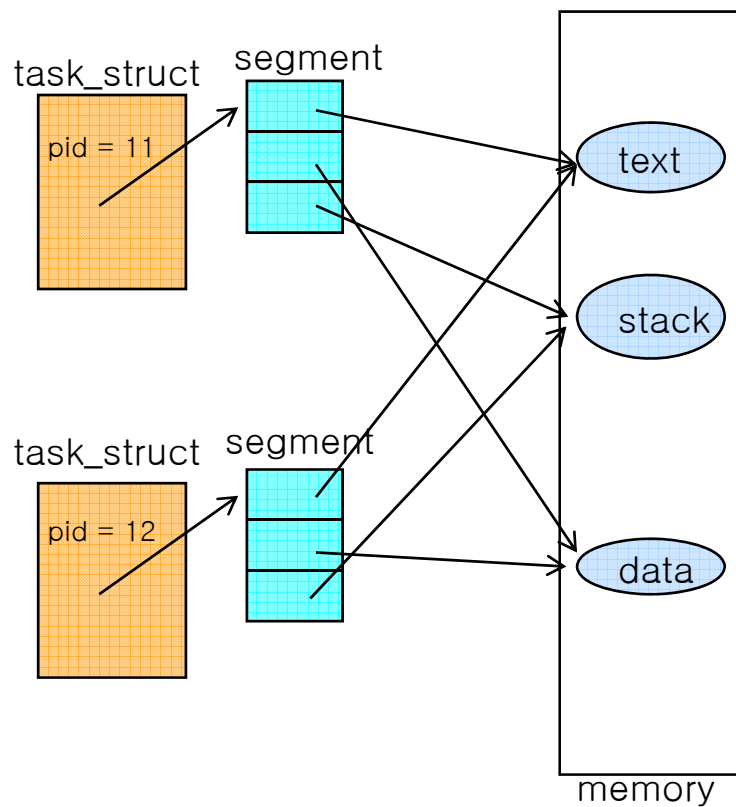


✓ address space : basic protection barrier

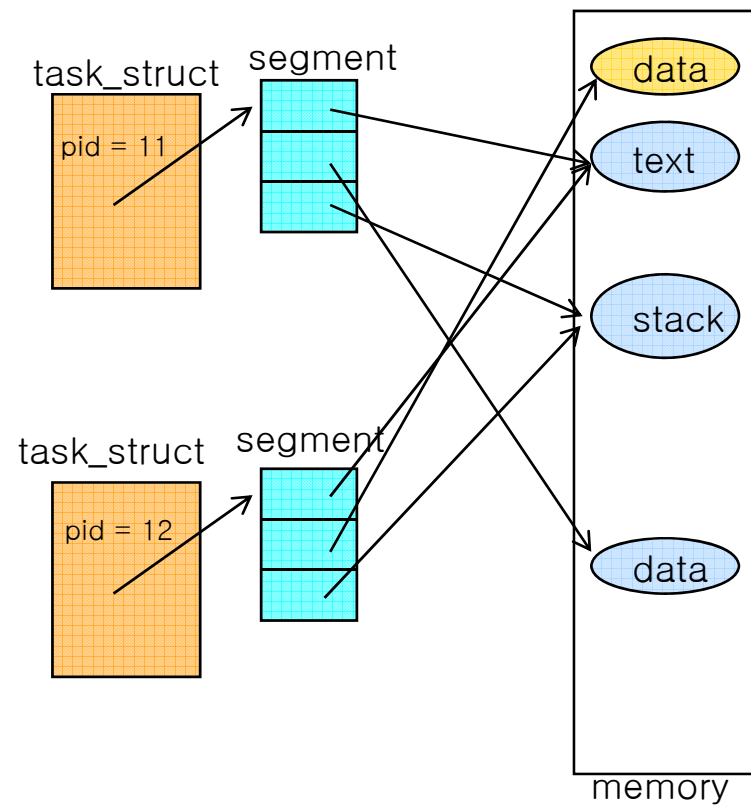
# Memory Context (4/5)

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

after fork with COW

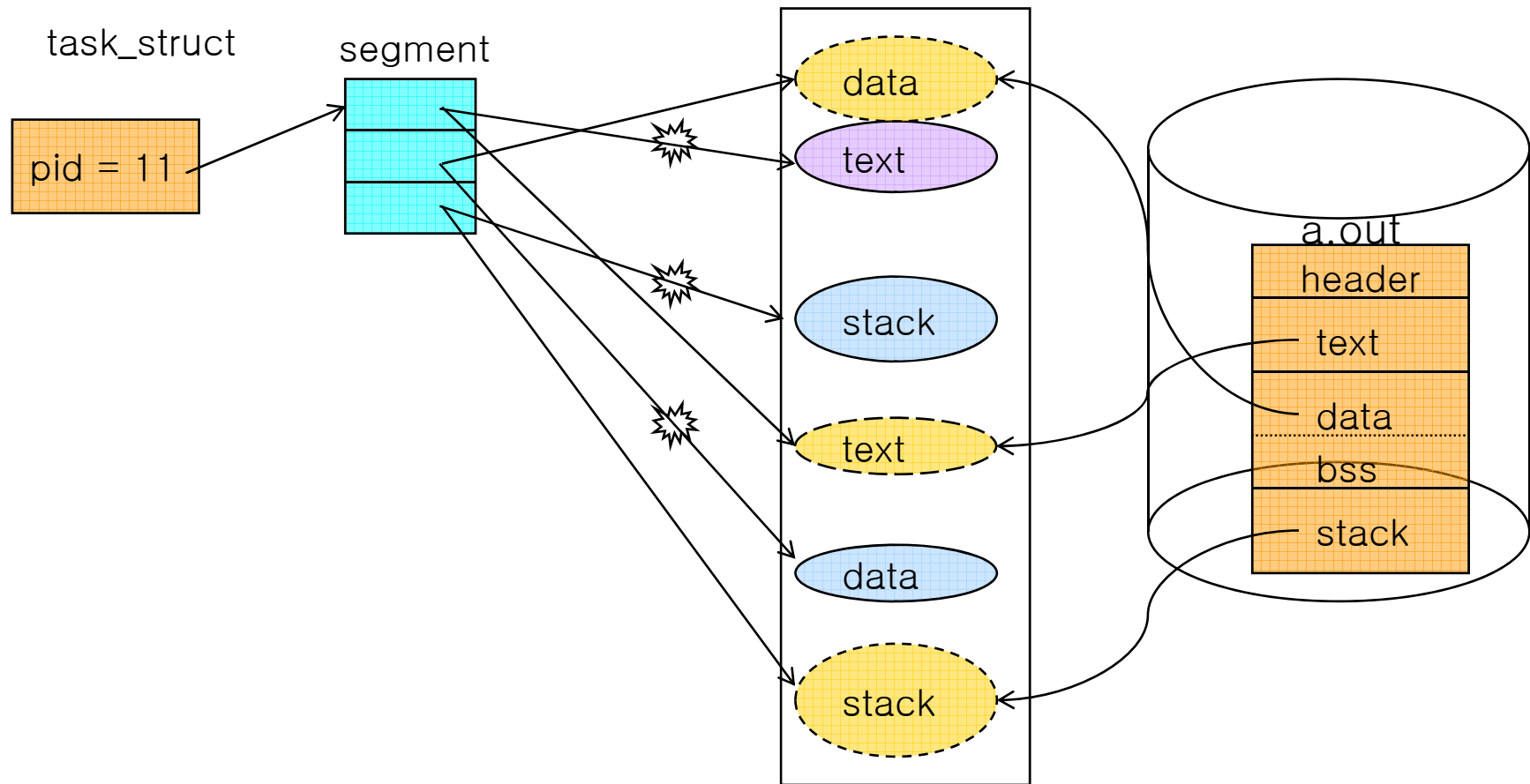


after "glob++" operation





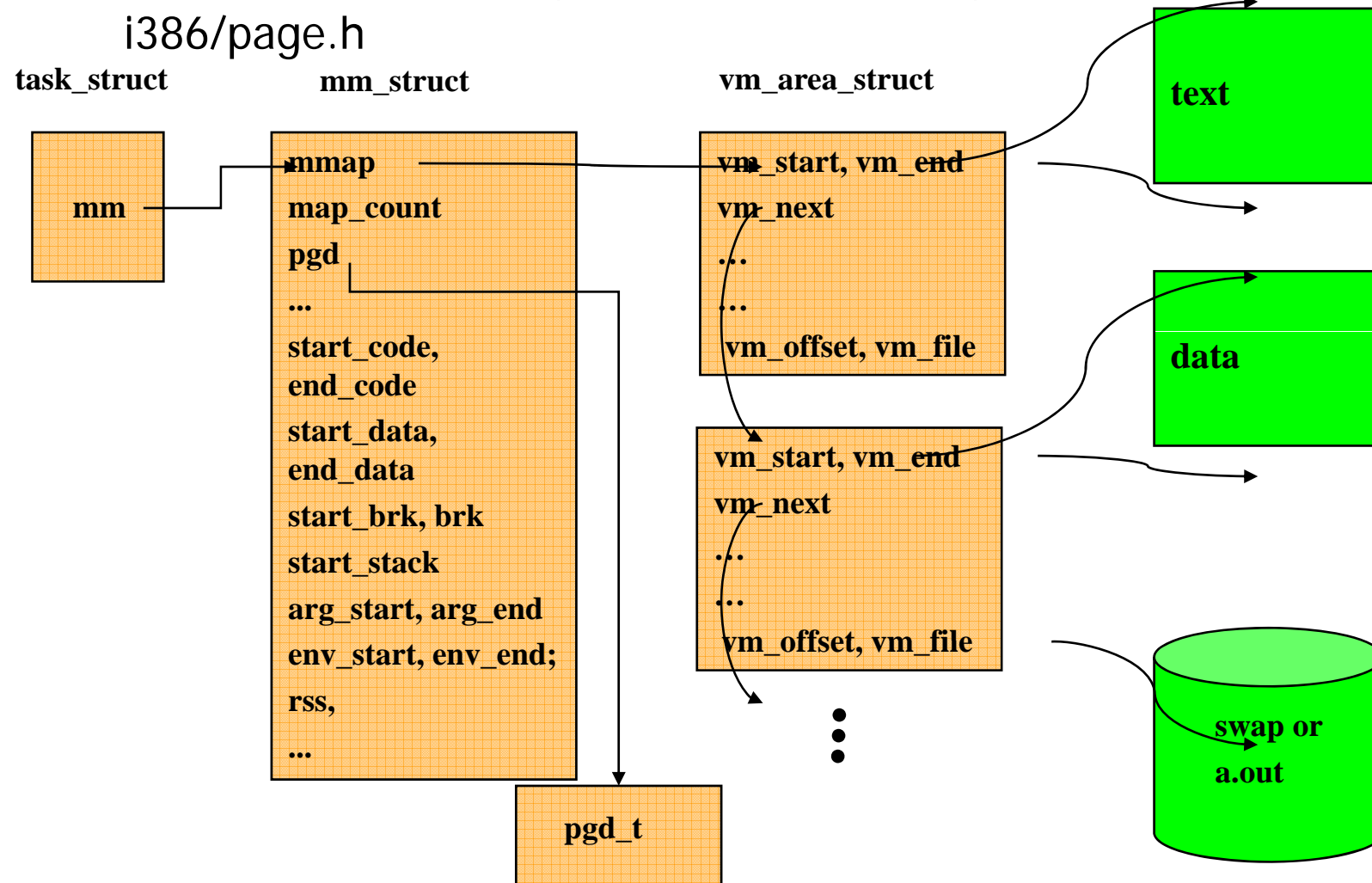
- 새로운 프로그램 수행: `execve()` internal memory



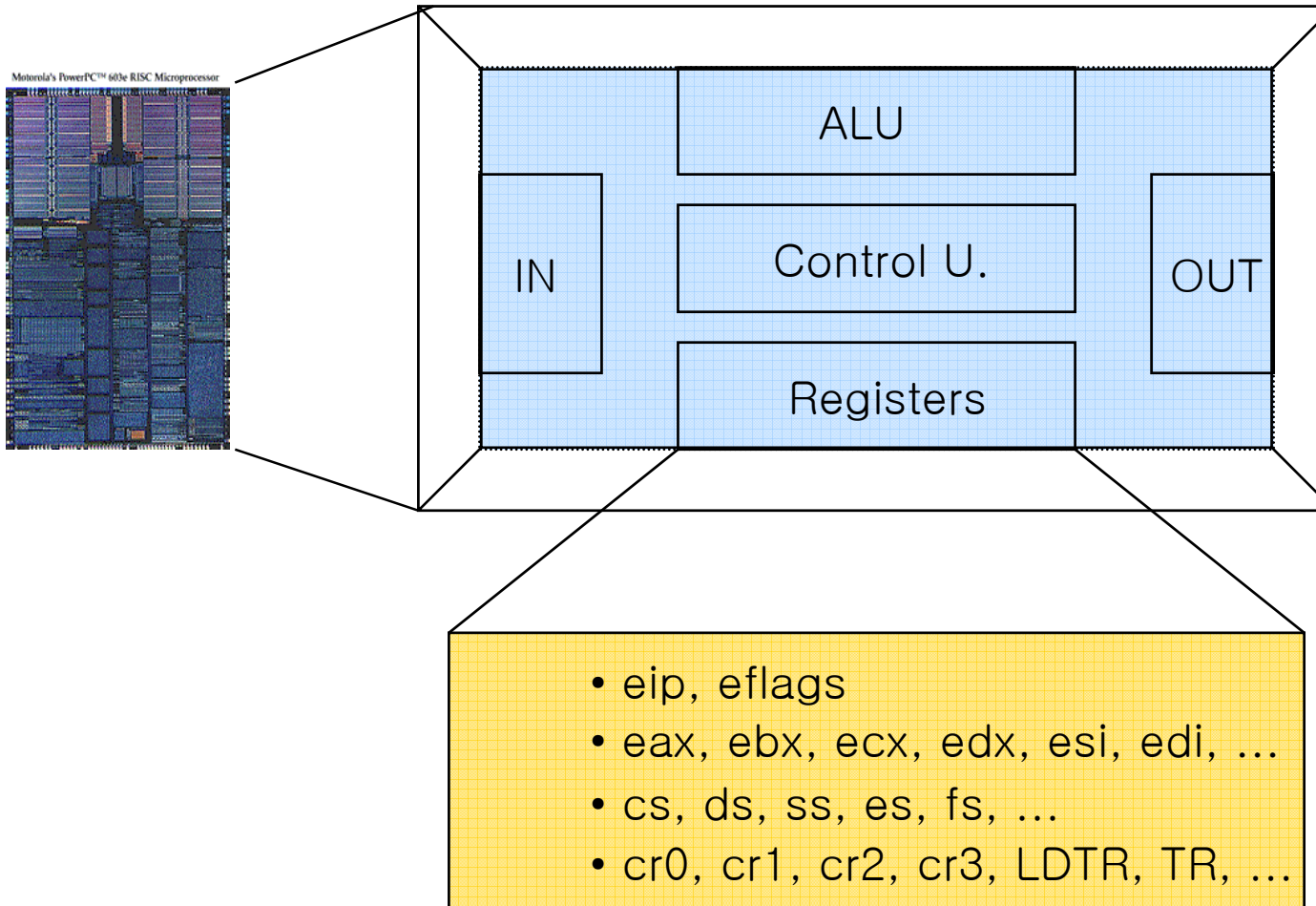
# Task의 memory context

## ■ 메모리 관리 자료 구조

- ✓ include/linux/sched.h, include/linux/mm.h, include/asm-i386/page.h

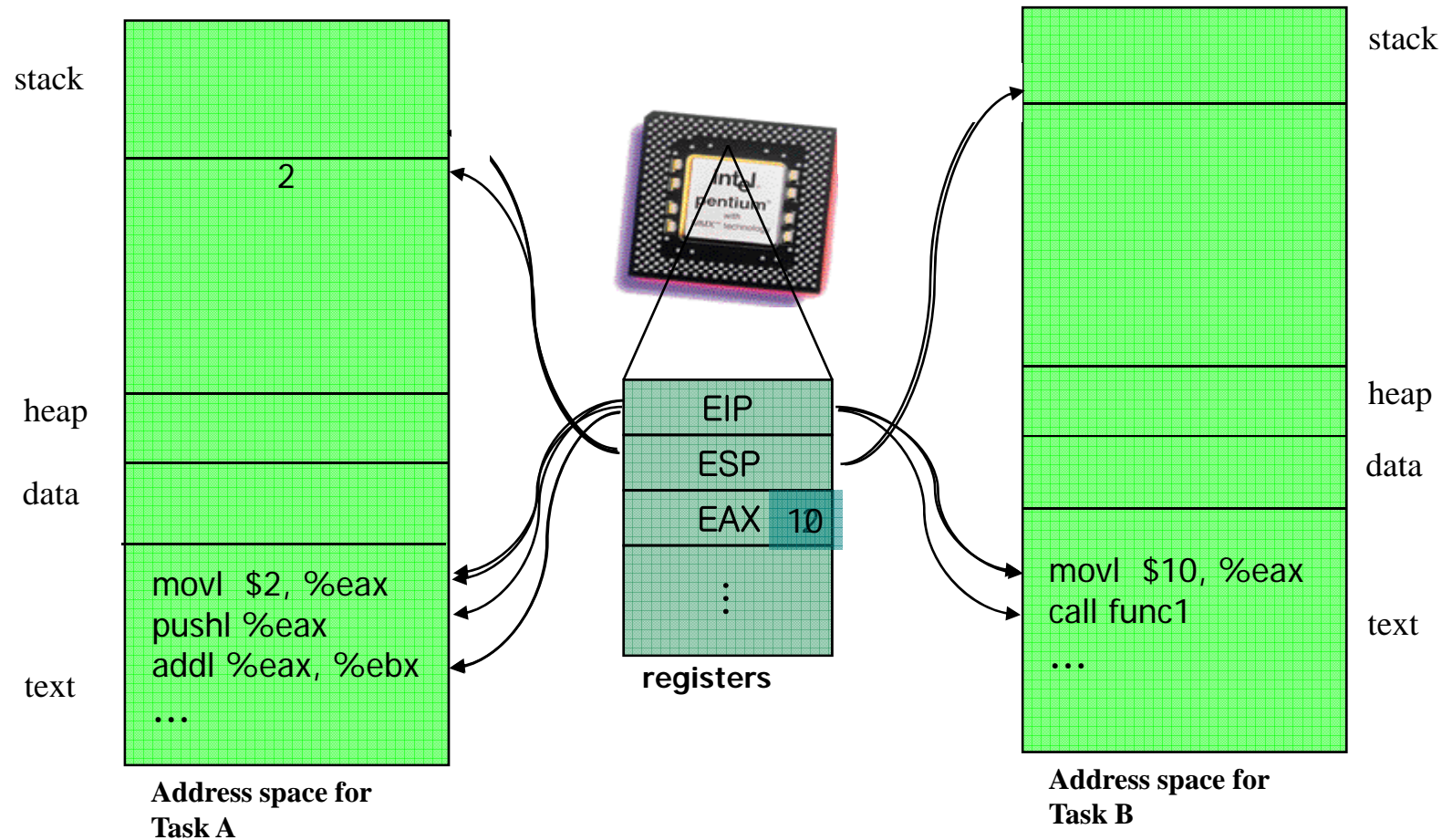


- brief reminds the 80x86 architecture



# Hardware Context(2/4)

- 스레드 자료 구조 : CPU 추상화

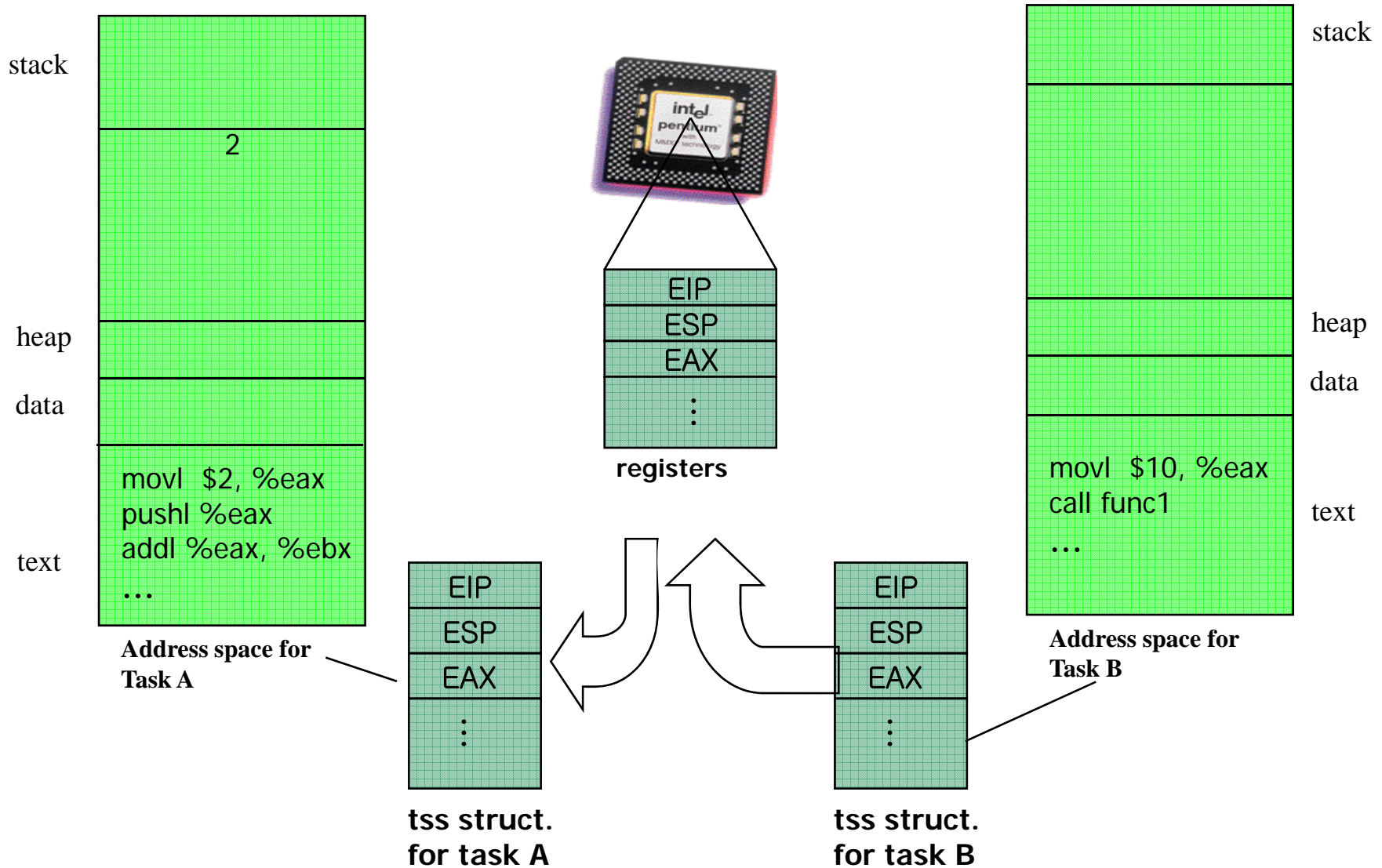


다시 Task A가 수행되려면?

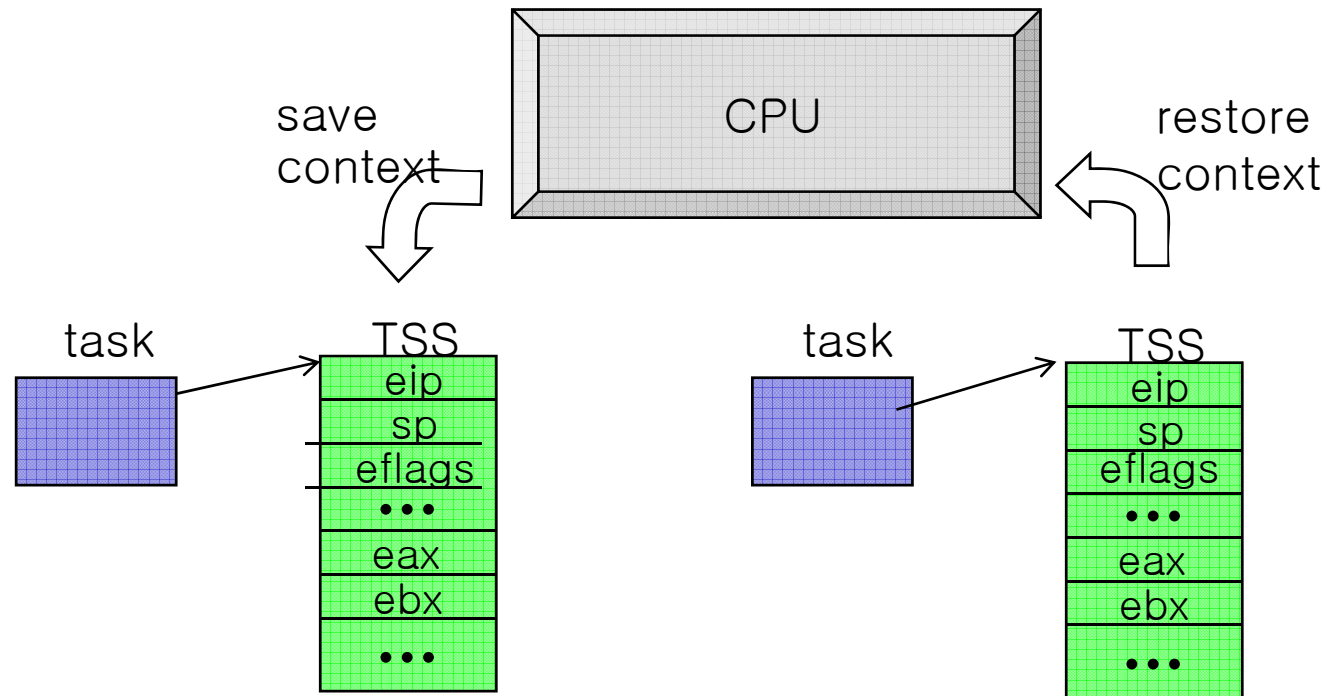
문맥 교환(Context Switch) → thread structure

# Hardware Context(3/4)

## ■ 쓰레드 자료 구조 : CPU 추상화

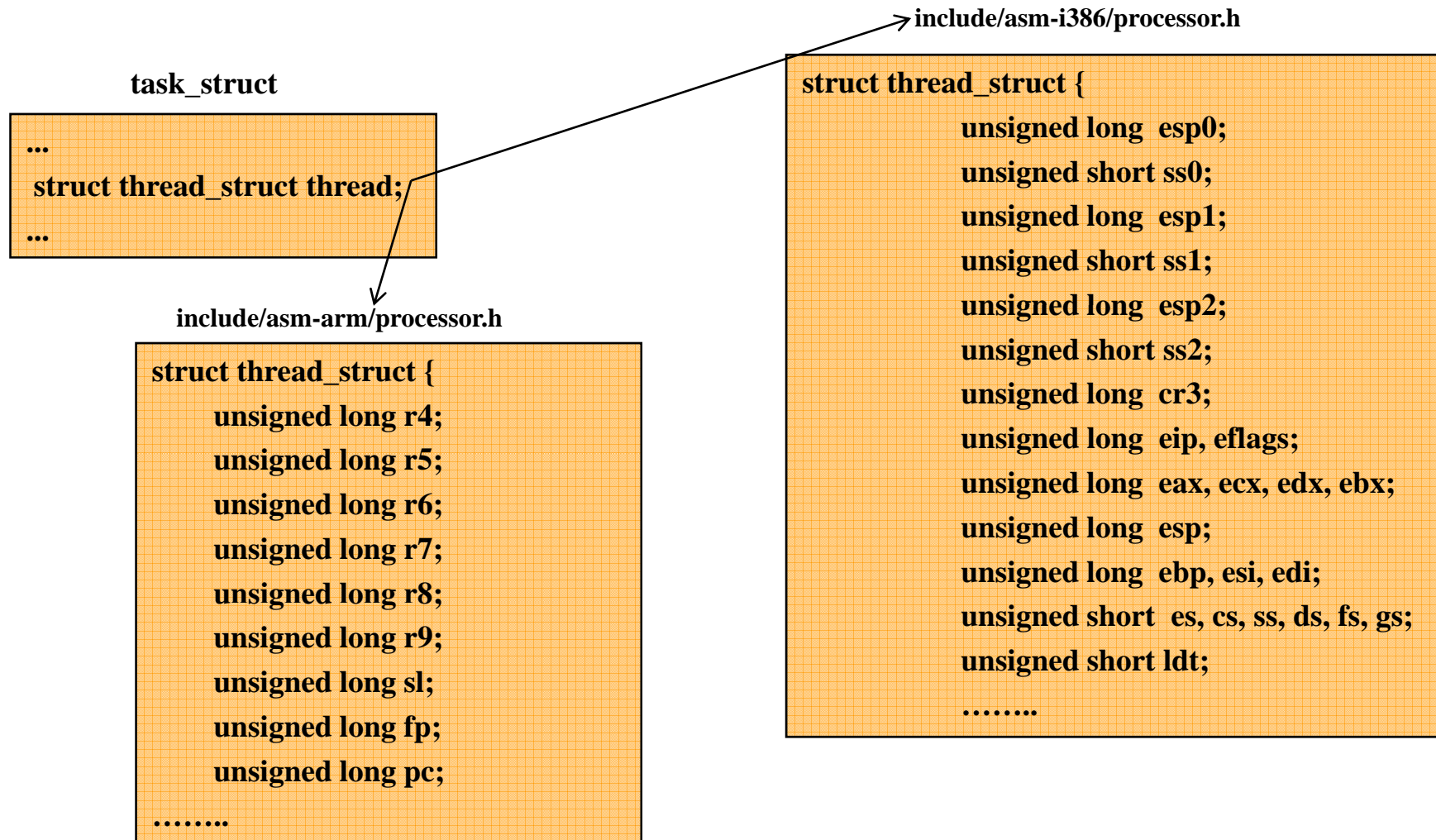


## ■ 문맥 교환 (Context Switch)



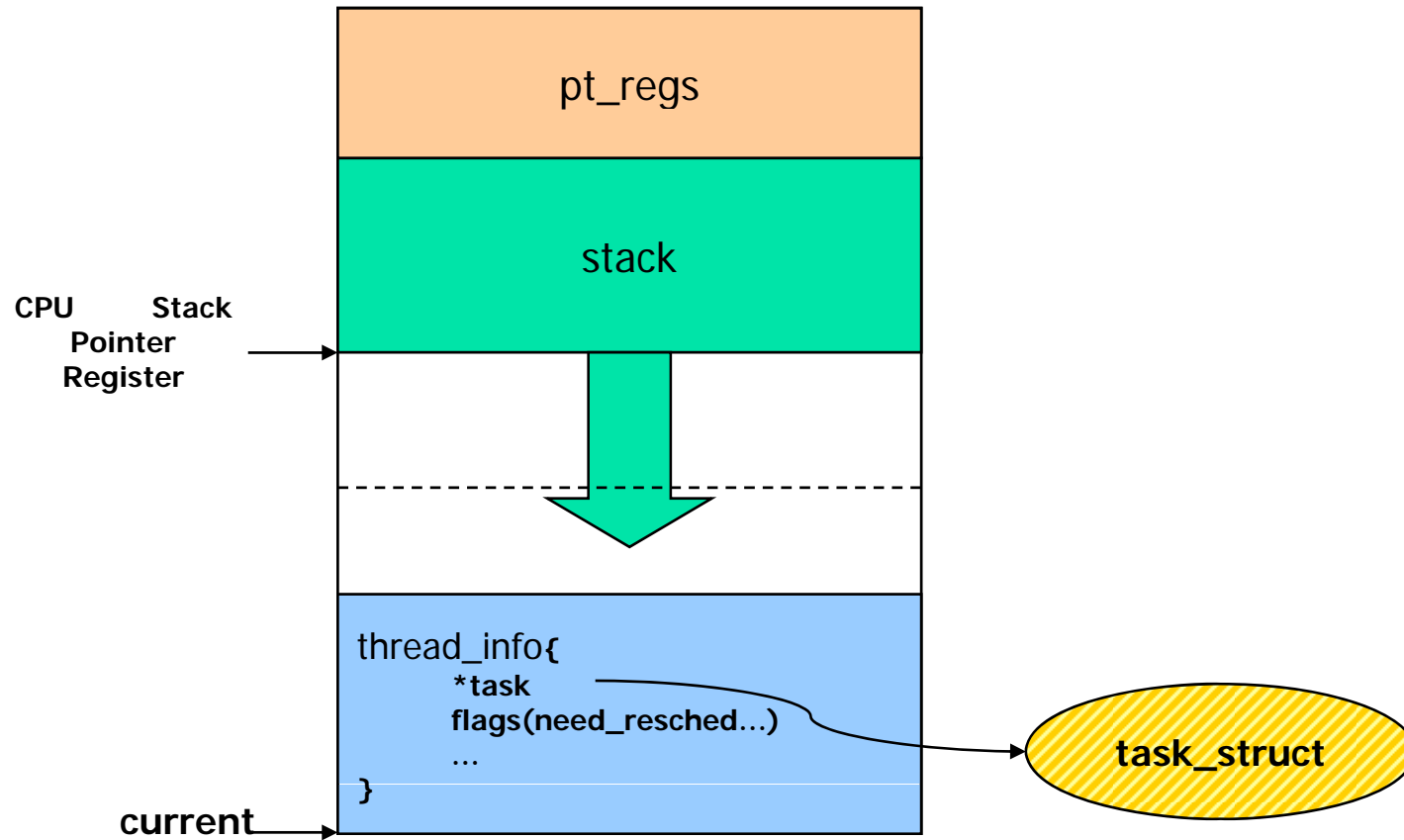
# Task의 H/W context

- 스레드 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

```
asm linkage 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;
};
```

- `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

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

변환한 IRQ번호

# Linux의 Task 생성

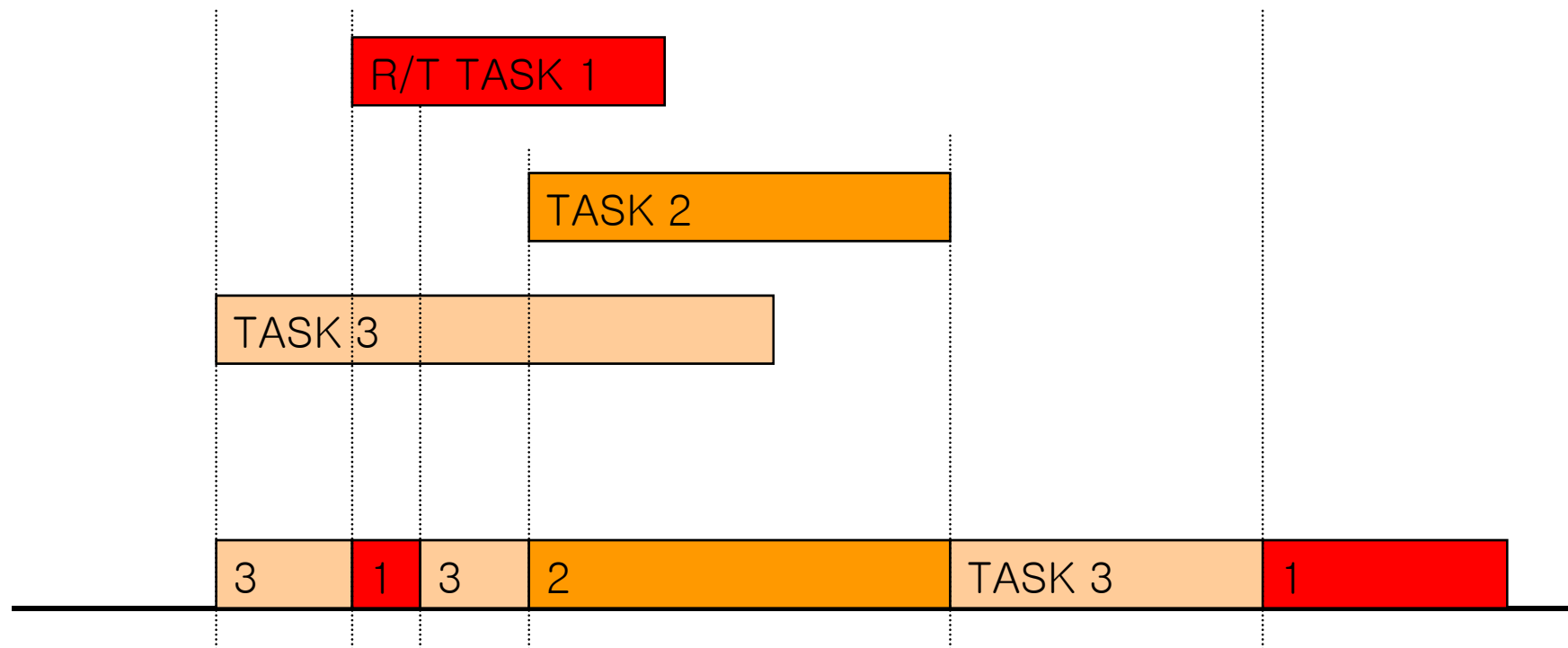
## ■ 프로세스 관련 Interface

- ✓ fork(), vfork(), clone() → do\_fork() → copy\_process()
  - dup\_task\_struct(): 새 커널 스택, 부모의 task\_struct 복사
  - get\_pid()를 호출해 새로운 PID 할당
  - 플래그에 따라 열린 파일, 파일시스템 정보, 시그널 핸들러, 프로세스 주소 영역, namespace 등을 복제 / 공유
    - 하위 바이트는 자식 프로세스 종료 시 부모 프로세스에 전달할 시그널 신호(보통 SIGCHLD), 나머지 3바이트는 아래와 같음
      - CLONE\_VM : 메모리 디스크립터와 모든 PT 공유
      - CLONE\_FS : 루트 dir과 CWD 나타내는 table과 'umask' 값 공유
      - CLONE\_FILES : 열린 file 나타내는 테이블 공유
      - CLONE\_PARENT : 새로 만들어지는 자식 P의 부모를 호출 P의 부모로 설정
      - CLONE\_PID : PID 공유
      - CLONE\_PTRACE : ptrace()로 부모 P 추적중 이라면, 자식 P도 추적 가능
      - CLONE\_SIGHAND : 시그널 핸들러 나타내는 table 공유
      - CLONE\_THREAD : 자식 P를 부모 P와 같은 스레드 그룹에 넣고, 자식 P의 tgid 필드도 이에 알맞게 설정, CLONE\_PARENT 자동 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

- `exit()` → `do_exit()`
  - ✓ `task_struct`구조체의 `flags`멤버에 `PF_EXITING`플래그 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 `init`프로세스로 지정
    - 고아라면?
      - `forget_original_parent()`를 사용함
      - `Reaper`를 프로세스의 스레드그룹에 있는 다른 태스크로 설정
      - 다른 태스크가 없으면 `reaper`를 `child_reaper`로 설정 → 이것이 바로 `init`프로세스이다
  - ✓ 태스크의 상태를 `TASK_ZOMBIE`로 설정
    - 이후에는 부모에게 정보를 전달하기 위한 커널 스택과 슬랩 객체 즉, `thread_info`와 `task_struct`구조체가 남아있다.
  - ✓ `schedule()`함수를 호출해 새 프로세스로 스위칭

- 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



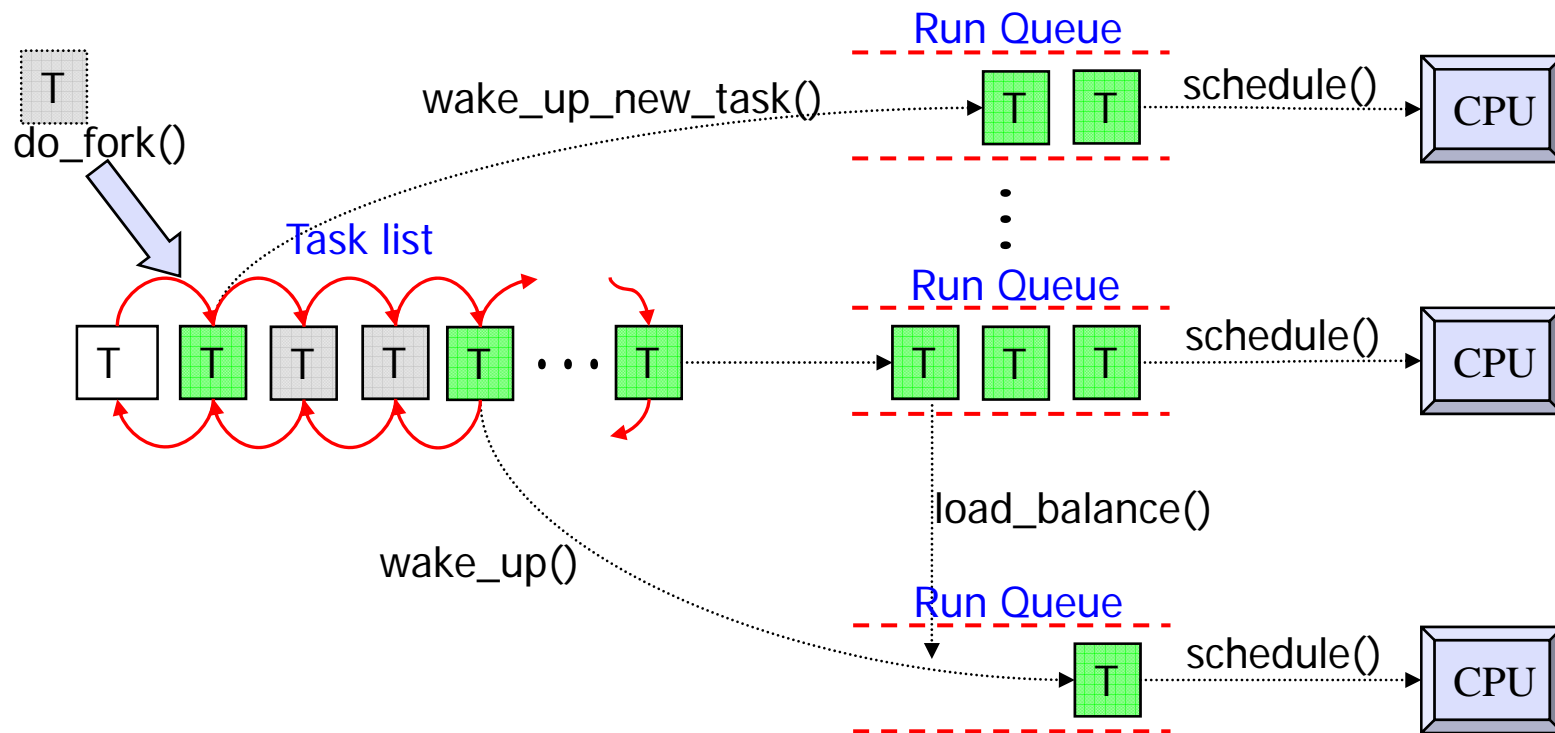
## ■ 해결책은?

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

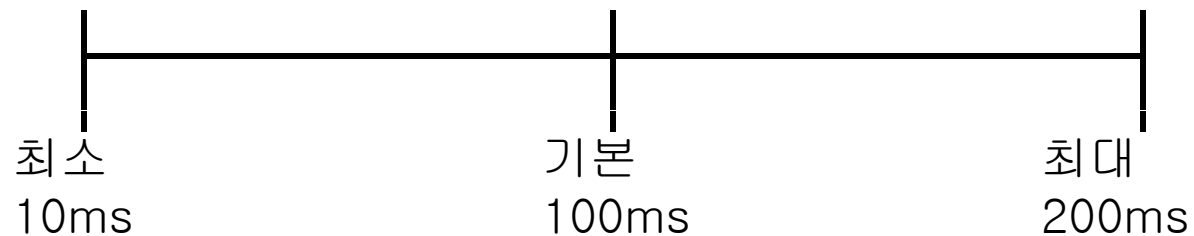
➔ 동적 우선 순위 변경이 가능해야 한다

## ■ 스케줄링

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



- 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



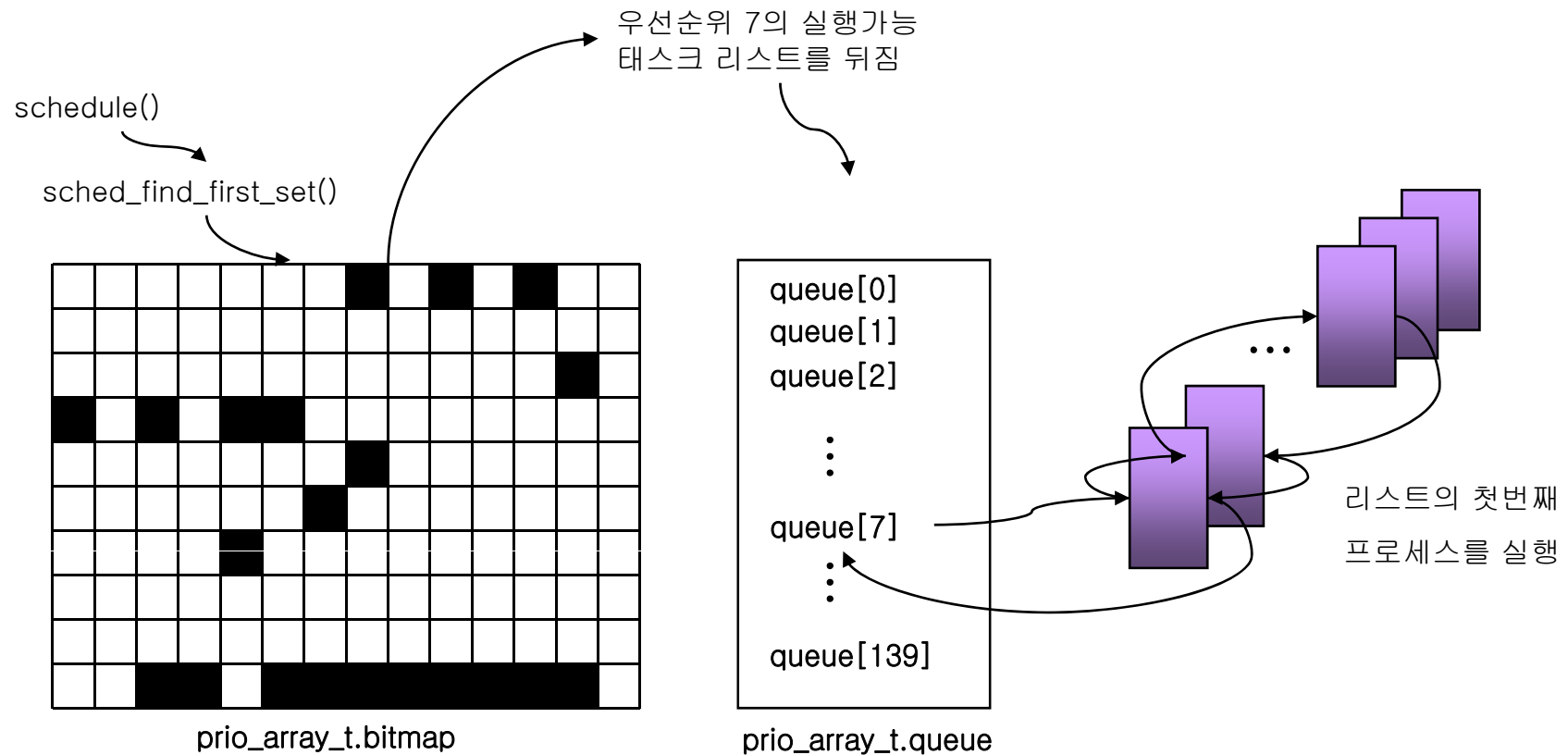
- 실행 큐
  - ✓ 프로세서에 있는 실행 가능한 프로세스의 목록
  
- 우선순위 배열 →  $O(1)$  scheduling
  - ✓ 각 실행 큐는 2개의 우선순위 배열을 가짐
  - ✓ 활성(active), 비활성(expired)
  - ✓ 각 우선순위 레벨마다의 실행가능 프로세스 큐를 유지
  - ✓ 각 큐에는 해당 우선순위 레벨의 실행 가능한 프로세스 목록 유지
  
- ✓ 우선순위 비트맵 사용 → 상수 시간 내에 최고 우선순위 태스크 선택 가능
  - 특정 우선 순위의 태스크가 TASK\_RUNNING상태가 되면 해당 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)

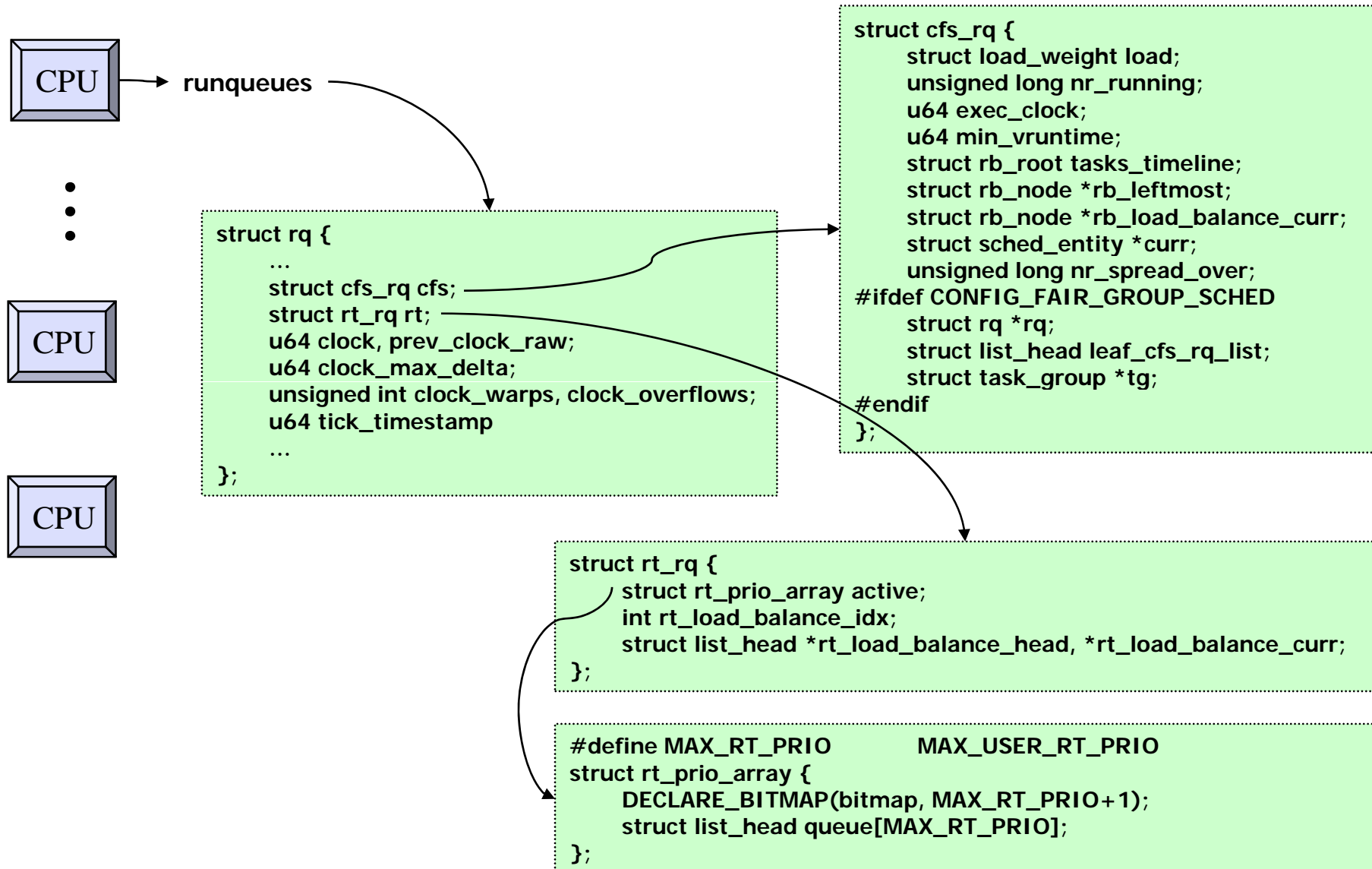
- current의 need\_resched 필드를 1로 설정해 우회적 호출함
- 쓰이는 경우
  - current가 자신의 CPU시간 quantum을 모두 썼을 때, update\_process\_times()가 설정함
  - 임의의 process가 깨어 났는데, 그 process의 우선순위가 현재 process의 우선순위보다 높을 때.
  - seched\_setscheduler()이나, sched\_yield() 시스템 콜 호출 시

# 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

```

struct task_struct {
    ...
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
    unsigned int policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice;
    ...
}
    
```

```

struct sched_entity {
    struct load_weight    load;
    struct rb_node        run_node;
    unsigned int          on_rq;

    u64    exec_start;
    u64    sum_exec_runtime;
    u64    vruntime;
    u64    prev_sum_exec_runtime;
    ...
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct sched_entity *parent;
    struct cfs_rq *cfs_rq;
    struct cfs_rq *my_q;
#endif
};
    
```

```

SCHEID_NORMAL
SCHEID_BATCH
SCHEID_IDLE
    
```

```

static const struct sched_class fair_sched_class = {
    .next           = &idle_sched_class,
    .enqueue_task   = enqueue_task_fair,
    .dequeue_task   = dequeue_task_fair,
    .yield_task     = yield_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task = pick_next_task_fair,
    .put_prev_task  = put_prev_task_fair,
#ifdef CONFIG_SMP
    .load_balance   = load_balance_fair,
    .move_one_task  = move_one_task_fair,
#endif
    .set_curr_task  = set_curr_task_fair,
    .task_tick      = task_tick_fair,
    .task_new       = task_new_fair,
};
    
```

```

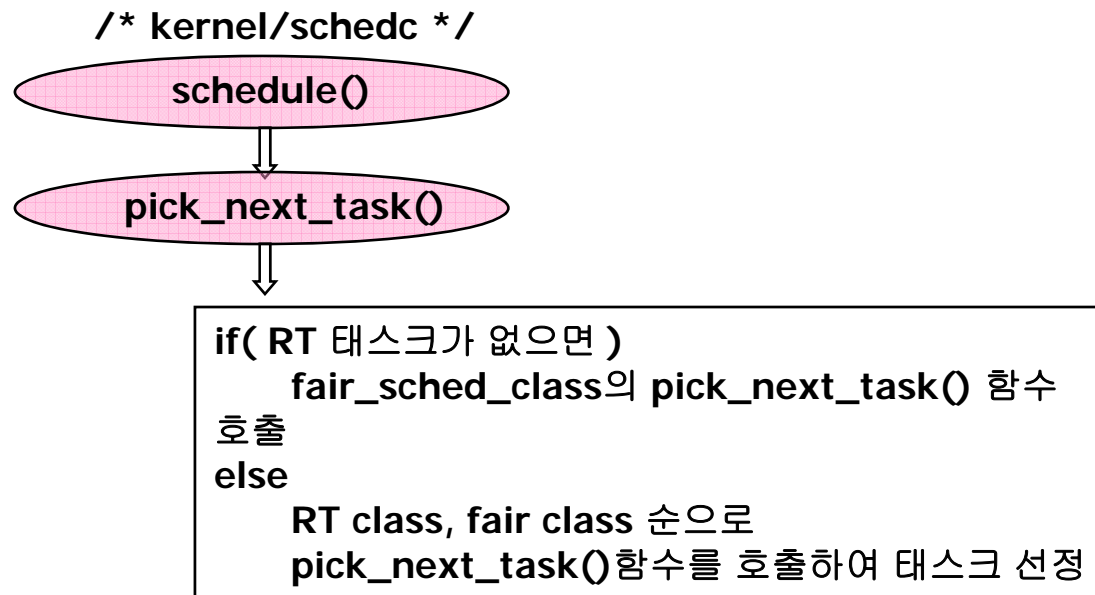
SCHEID_FIFO
SCHEID_RR
    
```

```

const struct sched_class rt_sched_class = {
    .next           = &fair_sched_class,
    .enqueue_task   = enqueue_task_rt,
    .dequeue_task   = dequeue_task_rt,
    .yield_task     = yield_task_rt,
    .check_preempt_curr = check_preempt_curr_rt,
    .pick_next_task = pick_next_task_rt,
    .put_prev_task  = put_prev_task_rt,
#ifdef CONFIG_SMP
    .load_balance   = load_balance_rt,
    .move_one_task  = move_one_task_rt,
#endif
    .set_curr_task  = set_curr_task_rt,
    .task_tick      = task_tick_rt,
};
    
```

# 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));

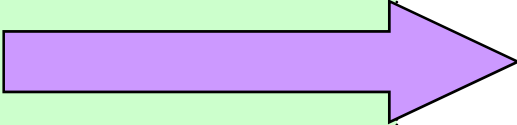
    ① param.sched_priority = 10;
    sched_setscheduler(0, SCHED_FIFO, &param);
    sched_getparam( 0, &param);

    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)

The screenshot shows the TRACE32 debugger interface. The main window is titled 'B::Data.List' and contains a 'Code Window' with the following C code:

```
688         prev->active_mm = NULL;
689         mmdrop(oldmm);
690     }
691
692     /*
693     * This just switches the register state and the
694     * stack.
695     */
696     next = 0xC02CC000;
697     switch_to(prev, next, prev);
698     __schedule_tail(prev);
699     prev = 0xC02C0000;
700
701     same_process:
702     reacquire_kernel_lock(current);
703     if (current->need_resched)
704         goto need_resched_back;
705     return;
706 }
```

The 'Memory Dump Window' at the bottom left shows a dump of memory starting at address 0xC02C1F80. A red box highlights the address 0xC02C1F80, which contains the value 0003AEED. This value is also shown in the 'CPU register window' at the bottom right, where it is assigned to register R8. A red circle highlights the value 0003AEED in the register window, and a red arrow points from this circle to the highlighted memory address in the dump window.

The 'CPU register window' at the bottom right shows the state of the CPU registers. The registers are listed in two columns, with their names, values, and offsets. The register R8 is highlighted with a red circle, showing the value 0003AEED. The register R13 is highlighted with a red circle, showing the value 40000093. The register R14 is highlighted with a red circle, showing the value 001C280. The register SPSR is highlighted with a red circle, showing the value 40000093.

Memory Dump Window

CPU register window

# \_\_switch\_to() in ARM (2/16)

The screenshot displays a debugger interface with three main windows:

- Assembly View (Top Left):** Shows assembly code for the `__switch_to` function. The instruction `ldmia r13!, {r4-r11, pc}` is highlighted, and a red arrow points from it to the register window.
- Register Window (Top Right):** Lists registers R0 through R14 and SPSR. Register R14 is highlighted with a red circle and contains the value `C66C`. Other registers like R8, R9, R10, and R11 are also circled in red.
- Memory Window (Bottom):** Shows memory addresses and their contents. The address `0003AEE0` is highlighted, which corresponds to the PC value in the register window.



# \_\_switch\_to() in ARM (3/16)

The image shows a debugger interface with three main windows:

- Assembly Window (B::Data\_List):** Displays assembly code for the `__switch_to:` function. The current instruction is `mrs r12, cpsr` at address `SR:0000:C001C578`. Other instructions include `mov r9, r13, lsr #0x0D`, `mov r9, r9, lsr #0x0D`, `b 0xC001C630 ; ret_to_user`, `stmdb r13!, {r4-r11, r14}`, `str r12, [r13, #-0x4]!`, `mra r4, r5, acc0`, `stmdb r13!, {r4-r5}`, `str r13, [r0, #0x318]`, `ldr r13, [r1, #0x318]`, `ldr r2, [r1, #0x31C]`, `ldmia r13!, {r4-r5}`, `mar acc0, r4, r5`, `ldr r12, [r13], #0x4`, `mcr p15, 0x0, r2, c3, c0, 0x0; p15, 0, r2, c3, c0`, `msr spsr, r12`, `ldmia r13!, {r4-r11, pc}A`, `nop`, and `nop`.
- Register Window (B::Register):** Shows the state of registers. The PC register is highlighted with a red circle and contains the value `20000013`. Other registers include R0-R14, SPSR, and ACC0. The SPSR register also contains `20000013`.
- Memory Window (B::d 0xc02c1f8c):** Shows memory contents at address `0xc02c1f8c`. The memory contains a sequence of instructions, including `SD:0000:C02C1F40` through `SD:0000:C02C1FE0`.

# \_\_switch\_to() in ARM (4/16)

The image shows a debugger interface with three main windows:

- B::Data\_List**: Shows assembly code for the `__switch_to` function. The code includes instructions like `mov r9, r13, lsr #0x0D`, `stmdb r13!, {r4-r11, r14}`, `mrs r12, cpsr`, `str r12, [r13, #-0x4]!`, `mra r4, r5, acc0`, `stmdb r13!, {r4-r5}`, `str r13, [r0, #0x318]`, `ldr r13, [r1, #0x318]`, `ldr r2, [r1, #0x31C]`, `ldmia r13!, {r4-r5}`, `mar acc0, r4, r5`, `ldr r12, [r13], #0x4`, `mcr p15, 0x0, r2, c3, c0, 0x0; p15, 0, r2, c3, c0`, `msr spsr, r12`, `ldmia r13!, {r4-r11, pc}^`, `nop`, and `nop`.
- B::Register**: Shows the current state of registers. R11 is highlighted with a red circle and contains the value `C02C0000`. Other registers shown include R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R12, R13, R14, SPSR, and ACC0.
- 010101 [B::d 0xc02c1f68]**: Shows a memory dump at address `0xc02c1f68`. The dump displays hexadecimal values and their corresponding ASCII representations for addresses from `SD:0000:C02C1F20` to `SD:0000:C02C1FC0`.



# \_\_switch\_to() in ARM (5/16)

The screenshot shows a debugger window with three panes:

- Source:** Assembly code for `__switch_to()`. The instruction `b 0xC001C630 ; ret_to_user` is highlighted.
- Register:** Register values for R0-R14, CPSR, and SPSR. Register R4 contains the value 0180, which is circled in red.
- Memory:** A memory dump at address `0xC02C1F64`. A red arrow points from the circled value in R4 to the first byte of the memory dump, which is `01`.

Register values (relevant to the red circle):

Register	Value
R0	C02C0000
R1	C02CC000
R2	60000013
R3	2
R4	0180
R5	0
R6	C02CC000
R7	0
R8	C0194000
R9	69052D06
R10	A0014B30
R11	C02C1FAC
R12	20000013
R13	BFFFFFF0
R14	C66C

Memory dump (relevant to the red arrow):

Address	0	4	8	C
SD:0000:C02C1F20	FFFFFFFE	60000093	20000013	C002C554
SD:0000:C02C1F30	C02C1F78	F8D00000	04000000	C0024D80
SD:0000:C02C1F40	C001C2A0	C0196F80	60000093	60000013
SD:0000:C02C1F50	00000002	C02C0000	C00140A0	C02CC000
SD:0000:C02C1F60	00000000	20000013	C02C0000	C00140A0
SD:0000:C02C1F70	C02CC000	00000000	C0194000	69052D06
SD:0000:C02C1F80	A0014B30	C02C1FAC	C0024D8C	0003AEE0
SD:0000:C02C1F90	000001F4	C02C1FB0	C01D2EA4	C0196BB4
SD:0000:C02C1FA0	C02C1FE0	C02C1FB0	C0024B0C	C0024B80
SD:0000:C02C1FB0	00000000	00000000	0003AEE0	C02C0000
SD:0000:C02C1FC0	C0024A84	C02C0340	C02C0330	C02C0000

# \_\_switch\_to() in ARM (6/16)

The screenshot displays a debugger interface with three main windows:

- B::Data\_List**: Shows assembly code for the `__switch_to` function. The current instruction is `str r13,[r0,#0x318]` at address `SR:0000:C001C588`. Other instructions include `ldr r13,[r1,#0x318]`, `ldr r2,[r1,#0x31C]`, `ldmia r13!,{r4-r5}`, `mar acc0,r4,r5`, `ldr r12,[r13],#0x4`, `mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0`, `msr spsr,r12`, and `ldmia r13!,{r4-r11,pc}^`.
- B::Register**: Shows the state of registers. `R8` is highlighted with a red circle and contains the value `C02C0000`. `R13` is also highlighted with a red circle and contains `20000013`. Other registers like `R9` (69052D06), `R10` (A0014B30), and `R11` (C02C1FAC) are also visible.
- Memory View**: Shows a memory dump at address `0xc02c1f64`. The value `00000180` is highlighted with a red box, corresponding to the value in register `R8`.



# \_\_switch\_to() in ARM (7/16)

The screenshot displays a debugger interface with three main windows:

- Assembly Window (B::Data\_List):** Shows assembly instructions for the `__switch_to:` function. The instruction `ldr r13,[r1,#0x318]` is highlighted.
- Registers Window (B::Register):** Lists the state of various registers. R8 is `C02C0000`, R9 is `C02CC000`, and R14 is `C02C1F5C`. A red circle highlights R14, and a red arrow points from it to the memory window.
- Memory Window (B::d 0xc02c0318):** Shows memory contents. The address `C02C1F5C` is highlighted with a red box, and its value is `00000010`.

The assembly code in the assembly window includes:

```
SR:0000:C001C568 E1A096AD mov r9,r13,lsr #0x0D
SR:0000:C001C56C E1A09689 mov r9,r9,ls1 #0x0D
SR:0000:C001C570 EA00002E b 0xC001C630 ; ret_to_user
SR:0000:C001C574 E92D4FF0 __switch_to: stmdb r13!,{r4-r11,r14}
SR:0000:C001C578 E10FC000 mrs r12,cpsr
SR:0000:C001C57C E52DC004 str r12,[r13,#-0x4]!
SR:0000:C001C580 EC554000 mra r4,r5,acc0
SR:0000:C001C584 E92D0030 stmdb r13!,{r4-r5}
SR:0000:C001C588 E580D318 str r13,[r0,#0x318]
SR:0000:C001C58C E591D318 ldr r13,[r1,#0x318]
SR:0000:C001C590 E591231C ldr r2,[r1,#0x31C]
SR:0000:C001C594 E8BD0030 ldmia r13!,{r4-r5}
SR:0000:C001C598 EC454000 mar acc0,r4,r5
SR:0000:C001C59C E49DC004 ldr r12,[r13],#0x4
SR:0000:C001C5A0 EE032F10 mcr p15,0x0,r2,c3,c0,0x0: p15,0,r2,c3,c0
SR:0000:C001C5A4 E169F00C msr spsr,r12
SR:0000:C001C5A8 E8FD8FF0 ldmia r13!,{r4-r11,pc}^
SR:0000:C001C5AC E1A00000 nop
SR:0000:C001C5B0 E1A00000 nop
```

# \_\_switch\_to() in ARM (8/16)

The image shows a debugger interface with three windows:

- Assembly View (B::Data\_List):** Shows assembly code for the `__switch_to` function. The instruction at address `E591231C` is `ldr r2,[r1,#0x31C]`, which is highlighted in grey. The instruction at `E591D318` is `ldr r13,[r1,#0x318]`.
- Register Window (B::Register):** Shows the state of registers. The PC register is highlighted with a red circle and contains the value `C02CDF60`. A red arrow points from this value to the assembly view.
- Memory View (B::d 0xc02cc318):** Shows memory contents. The address `C02CDF60` is highlighted with a red box, and its value is `0000001D`.



# \_\_switch\_to() in ARM (9/16)

The image shows a debugger window with three panes. The top-left pane displays assembly code for the function `__switch_to()`. The top-right pane shows the register window with values for R0 through R14 and SPSR. The bottom pane shows a memory dump for address `0xc02cc31c`.

**Assembly Code (Left Pane):**

```
addr/line source
SR:0000:C001C568 E1A096AD mov r9,r13,lsr #0x0D
SR:0000:C001C56C E1A09689 mov r9,r9,lsr #0x0D
SR:0000:C001C570 EA00002E b 0xC001C630 ; ret_to_user
SR:0000:C001C574 E92D4FF0 __switch_to: stmdb r13!,{r4-r11,r14}
SR:0000:C001C578 E10FC000 mrs r12,cpsr
SR:0000:C001C57C E52DC004 str r12,[r13,#-0x4]!
SR:0000:C001C580 EC554000 mra r4,r5,acc0
SR:0000:C001C584 E92D0030 stmdb r13!,{r4-r5}
SR:0000:C001C588 E580D318 str r13,[r0,#0x318]
SR:0000:C001C58C E591D318 ldr r13,[r1,#0x318]
SR:0000:C001C590 E591231C ldr r2,[r1,#0x31C]
SR:0000:C001C594 E8BD0030 ldmia r13!,{r4-r5}
SR:0000:C001C598 EC454000 mar acc0,r4,r5
SR:0000:C001C59C E49DC004 ldr r12,[r13],#0x4
SR:0000:C001C5A0 EE032F10 mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0
SR:0000:C001C5A4 E169F00C msr spsr,r12
SR:0000:C001C5A8 E8FD8FF0 ldmia r13!,{r4-r11,pc}A
SR:0000:C001C5AC E1A00000 nop
SR:0000:C001C5B0 E1A00000 nop
```

**Register Window (Top-Right Pane):**

Register	Value
R0	C02C0000
R1	C02CC000
R2	1D
R3	2
R4	0180
R5	0
R6	C02CC000
R7	0
SPSR	20000013
PC	C001C594
CPSR	20000013

**Memory Dump (Bottom Pane):**

Address	0	4	8	C
SD:0000:C02CC310	00000000	00000000	C02CDF60	0000001D
SD:0000:C02CC320	C0196260	C0196280	C02B4200	C02CB040
SD:0000:C02CC330	FFFFFFFF	FFFFFFFF	00000000	C02CC338
SD:0000:C02CC340	00000000	00000000	00000000	00000000
SD:0000:C02CC350	00000000	00000000	00000000	00000000
SD:0000:C02CC360	00000000	00000000	00000001	00000000
SD:0000:C02CC370	C02CC370	C02CC370	0000FFFF	C01FEF34
SD:0000:C02CC380	00000000	00000000	C02CC388	C02CC388
SD:0000:C02CC390	00000000	00000000	C02CC398	C02CC398
SD:0000:C02CC3A0	00000000	00000000	00000000	00000000
SD:0000:C02CC3B0	00000000	00000020	0000FFFF	00000000

# \_\_switch\_to() in ARM (10/16)

The image shows a debugger window with three panes:

- Top Left (Data List):** Shows assembly code for the `__switch_to` function. The current instruction is `stmdb r13!, {r4-r11, r14}` at address `00000300`. The label `__switch_to:` is visible.
- Top Right (Register):** Shows the state of registers. Register `R12` is highlighted with a blue box and contains the value `0300`. Other registers like `R8`, `R9`, `R10`, `R11`, `R13`, `R14`, `SPSR`, and `PC` are also listed.
- Bottom (Memory Dump):** Shows a memory dump at address `00000300`. The value `00000300` is highlighted with a red box, corresponding to the value in register `R12`.



# \_\_switch\_to() in ARM (11/16)

The image shows a debugger window with three panes. The top-left pane displays assembly code for the function `__switch_to`. The top-right pane shows the current register state. The bottom pane shows a memory dump.

**Assembly Code (B::Data\_List):**

```
addr/line  source
SR:0000:C001C570 EA00002E b 0xC001C630 ; ret_to_user
SR:0000:C001C574 E92D4FF0 __switch_to: stmdb r13!,{r4-r11,r14}
SR:0000:C001C578 E10FC000 mrs r12,cpsr
SR:0000:C001C57C E52DC004 str r12,[r13,#-0x4]!
SR:0000:C001C580 EC554000 mra r4,r5,acc0
SR:0000:C001C584 E92D0030 stmdb r13!,{r4-r5}
SR:0000:C001C588 E580D318 str r13,[r0,#0x318]
SR:0000:C001C58C E591D318 ldr r13,[r1,#0x318]
SR:0000:C001C590 E591231C ldr r2,[r1,#0x31C]
SR:0000:C001C594 E88D0030 ldmia r13!,{r4-r5}
SR:0000:C001C598 EC454000 mar acc0,r4,r5
SR:0000:C001C59C E49DC004 ldr r12,[r13],#0x4
SR:0000:C001C5A0 EE032F10 mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0
SR:0000:C001C5A4 E169F00C msr spsr,r12
SR:0000:C001C5A8 E8FD8FF0 ldmia r13!,{r4-r11,pc}^
SR:0000:C001C5AC E1A00000 nop
SR:0000:C001C5B0 E1A00000 nop
SR:0000:C001C5B4 E1A00000 nop
SR:0000:C001C5B8 E1A00000 nop
```

**Register State (B::Register):**

Register	Value	Register	Value	Register	Value
R0	C02C0000	R8	C0194000	SP	20000013
R1	C02CC000	R9	69052D06		+04 C02CC000
R2	1D	R10	A0014B30		+08 C00140A0
R3	2	R11	C02C1FAC		+0C C02E2000
R4	0300	R12	20000013		+10 00000000
R5	0	R13	C02CDF68		+14 C0194000
R6	C02CC000	R14	C0024D8C		+18 C02CDFC8
R7	0	PC	C001C59C		+1C C02CC000
SVC	SPSR 20000013	CPSR	20000013		+20 C02CDFB0
Q	ACCO				+24 C0024D8C
					+28 00000001
USR:		FIQ:			+2C C0197F80
R8	C0194000	R8	0		+30 C02CC330
R9	69052D06	R9	0		+34 C02CC340
R10	A0014B30	R10	0		+38 C02CC000
R11	C02C1FAC	R11	0		+3C 00000000
R12	20000013	R12	0		+40 C02CDFB4
R13	BFFFFFF0	R13	0		+44 C0034C90
R14	C66C	R14	0		+48 C0024880
		SPSR	10		+4C 00000001
					+50 00000000
SVC:		IRQ:			+54 00000000
R13	C02CDF68	R13	60000093		+58 00010000
R14	C0024D8C	R14	C001C280		+5C 00000000
SPSR	20000013	SPSR	60000093		+60 00000000
					+64 C02CC000
UND:		ABT:			+68 C0197F88
R13	60000093	R13	20000093		+6C C0197F88
R14	C001C4A0	R14	C001C3C0		+70 C0034B74
SPSR	60000093	SPSR	20000093		+74 C02E2000
					+78 C01ED980
					+7C C01CB2A8
					+80 C0196BB4
					+84 69052D06
					+88 A0014B30
					+8C C001DD98
					+90 EFFFFFFF

**Memory Dump (B::d 0xc02cdf60):**

address	0	4	8	C	0123456789ABCDEF
SD:0000:C02CDF20	C001C2A0	C02CC000	C02E2000	60000013	AC SCNC, CN, C1NN
SD:0000:C02CDF30	00000001	C02CC000	C00140A0	C02E2000	02 H0 U0, 0 U, 0 3 U0
SD:0000:C02CDF40	00000000	C0194000	C02CDFC8	C02CC000	SNHNC, CR@SCH, C
SD:0000:C02CDF50	C02CDFB0	C02CDF9C	C01F73E4	C02CDF6C	NNNNN@16CD, CNC, C
SD:0000:C02CDF60	00000300	00000000	20000013	C02CC000	UUUUU@36SF, 0 U0, C
SD:0000:C02CDF70	C00140A0	C02E2000	00000000	C0194000	0@SCH, C NNNNN@16
SD:0000:C02CDF80	C02CDFC8	C02CC000	C02CDFB0	C0024D8C	CD, CNC, C8D, CEMSC
SD:0000:C02CDF90	00000001	C0197F80	C02CC330	C02CC340	SNNH87160C, 6@C, C
SD:0000:C02CFFA0	C02CC000	00000000	C02CDFB4	C0034C90	NC, C NNNHBD, C@L EC
SD:0000:C02CDFB0	C0024880	00000001	00000000	00000000	U0, 0 UUUU3F, C@L X0
SD:0000:C02CDFC0	00010000	00000000	00000000	C02CC000	0 KSC SNNNNNNNNNN
					NNNNNNNNNNNNNC, C
					UUUUUUUUUUUUU0, C

# \_\_switch\_to() in ARM (12/16)

The screenshot displays a debugger interface with three main windows:

- B::Data\_List**: Shows assembly code for the `__switch_to:` function. The current instruction is `mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0` at address `EE032F10`.
- B::Register**: Shows the state of registers. R13 contains `20000013`, R14 contains `C001C4A0`, and the PC is `C001C5A0`.
- B::d 0xc02cdf68**: Shows a memory dump at address `0xc02cdf68`. The current instruction `mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0` is highlighted, with its value `00000000` shown in the dump.



# switch to() in ARM (13/16)

The screenshot shows a debugger interface with three main windows:

- B::Data List:** Shows assembly code for the `__switch_to:` function. The code includes instructions like `stmdb r13!,{r4-r11,r14}`, `mrs r12,cpsr`, `str r12,[r13,#-0x4]!`, `mra r4,r5,acc0`, `stmdb r13!,{r4-r5}`, `str r13,[r0,#0x318]`, `ldr r13,[r1,#0x318]`, `ldr r2,[r1,#0x31C]`, `ldmia r13!,{r4-r5}`, `mar acc0,r4,r5`, `ldr r12,[r13],#0x4`, `mcr p15,0x0,r2,c3,c0,0x0; p15,0,r2,c3,c0`, and `msr spsr,r12`.
- B::Register:** Shows the current state of registers. R0 is `C02C0000`, R1 is `C02CC000`, R2 is `10`, R3 is `2`, R4 is `0300`, R5 is `0`, R6 is `C02CC000`, R7 is `0`, R8 is `C0194000`, R9 is `69052D06`, R10 is `A0014B30`, R11 is `C02C1FAC`. Special registers include `SPSR 20000013` and `CPSR 20000013`.
- B::PER:** Shows the CPUID register dump for an Intel XScale CoreGen processor. The `IBCRO` field is highlighted with a red box, showing `IBCRO C001C5A1 MVA C001C5A0 E enable`.

# switch to() in ARM (14/16)

The screenshot displays a debugger interface with three main windows:

- B::Data List:** Shows assembly instructions with addresses and sources. The instruction `msr spsr,r12` at address `E169F00C` is highlighted.
- B::Register:** Shows the state of registers. The `SPSR` register is highlighted with a red circle, showing a value of `20000013`.
- B::PER:** Shows system properties for the CPU. The `IBCRO` property is highlighted with a red box, showing `MVA C001C5A4 E enable`.



# \_\_switch\_to() in ARM (15/16)

The screenshot displays a debugger interface with three main windows:

- B::Data\_List**: Shows assembly code with addresses and sources. The current instruction is `ldmia r13!, {r4-r11, pc}^` at address `0x00000000`.
- B::Register**: Shows the state of registers. R14 is highlighted with a red circle, containing the value `C02CDF6C`.
- B::d 0xc02cdf68**: Shows a memory dump starting at address `0xc02cdf68`. The first few bytes are `00000300 00000000 20000013 C02CC000`.

# \_\_switch\_to() in ARM (16/16)

The screenshot displays a debugger interface with three main panes:

- Source Code (B::Data.List):** Shows the implementation of `__switch_to()`. Key lines include:
  - 696: `* This just switches the register state and the`
  - 697: `* stack.`
  - 697: `*/`
  - 697: `switch_to(prev, next, prev);`
  - 697: `__schedule_tail(prev);`
  - 702: `same_process:`
  - 702: `reactivate_kernel_lock(current);`
  - 703: `if (current->need_resched)`
  - 703: `goto need_resched_back;`
  - 704: `return;`
  - 704: `}`
- Register (B::Register):** Shows the state of registers R0-R14, SPSR, and ACC0. The PC is 002408C. The SPSR is 20000013. The ACC0 register contains 0180.
- Memory Dump (B::d 0xc02cdf6c):** Shows a dump of memory starting at address 0xc02cdf20. The dump includes addresses and hex values, such as `SD:0000:C02CDF20 20000013 FFFFFFFF C02CC000 C02CDFB0`.