

# Task Management

March, 2016  
Seungjae Baek

Dept. of software  
Dankook University

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

# Program & Process

2

```
root@localhost:/home/Lecture/LKI/test
File Edit View Search Terminal Help
[root@localhost test]# vi test.c
[root@localhost test]# gcc -o test test.c
[root@localhost test]# ls
test test.c
[root@localhost test]#
[root@localhost test]# file test
test: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (u
ses shared libs), for GNU/Linux 2.6.18, not stripped
[root@localhost test]#
[root@localhost test]# ./test
start
end
[root@localhost test]#
[root@localhost test]# ./test &
[1] 5360
[root@localhost test]# start

[root@localhost test]# ps
  PID TTY          TIME CMD
 5360 pts/0    00:00:00 test
 5361 pts/0    00:00:00 ps
22252 pts/0    00:00:00 bash
[root@localhost test]# end

[1]+  Done                  ./test
[root@localhost test]#
```

```
root@lo
File Edit View Search Term
#include <stdio.h>

int main(void)
{
    printf("start\n");
    sleep(5);
    printf("end\n");
    return 0;
}
```

# Task, Process and Thread

---

**Process**

**Thread**

# Task, Process and Thread

---

## Process

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

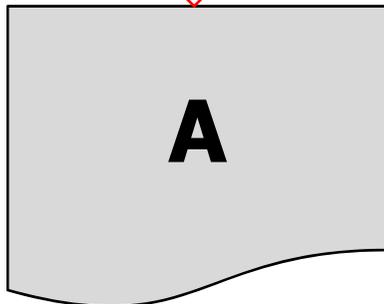
## Thread

- 디스패칭의 단위
- 실행 흐름
- 수행의 단위
- ...

# Task, Process and Thread

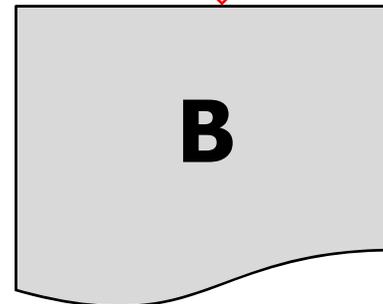
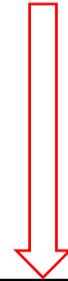
## Process

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



## Thread

- 디스패칭의 단위
- 실행 흐름
- 수행의 단위
- ...



# Task, Process and Thread

## Process

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

## Thread

- 디스패칭의 단위
- 실행 흐름
- 수행의 단위
- ...

fork or vfork

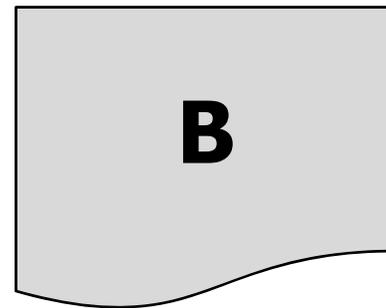
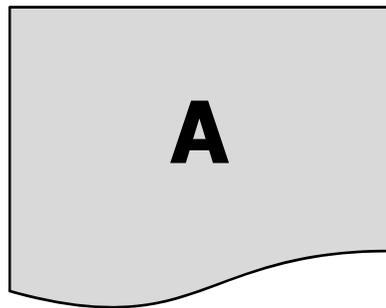
clone or  
pthread\_create

**A**

**B**

# Task, Process and Thread

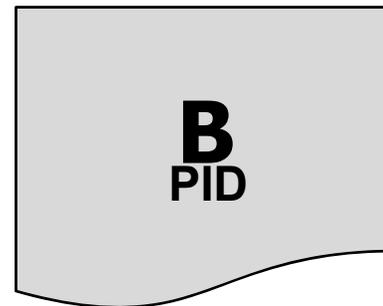
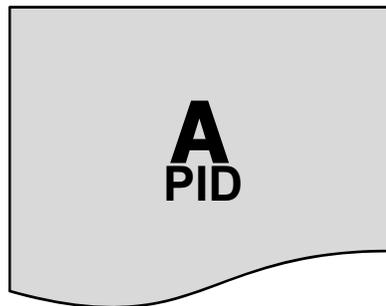
---



# Task, Process and Thread

---

## 1. PID

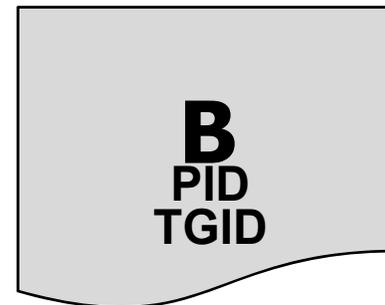
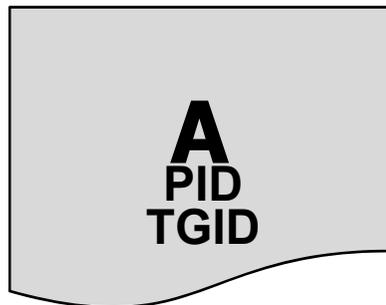


# Task, Process and Thread

---

## 1. PID

2. POSIX: 한 process내의 thread는 동일한 PID를 공유해야 한다

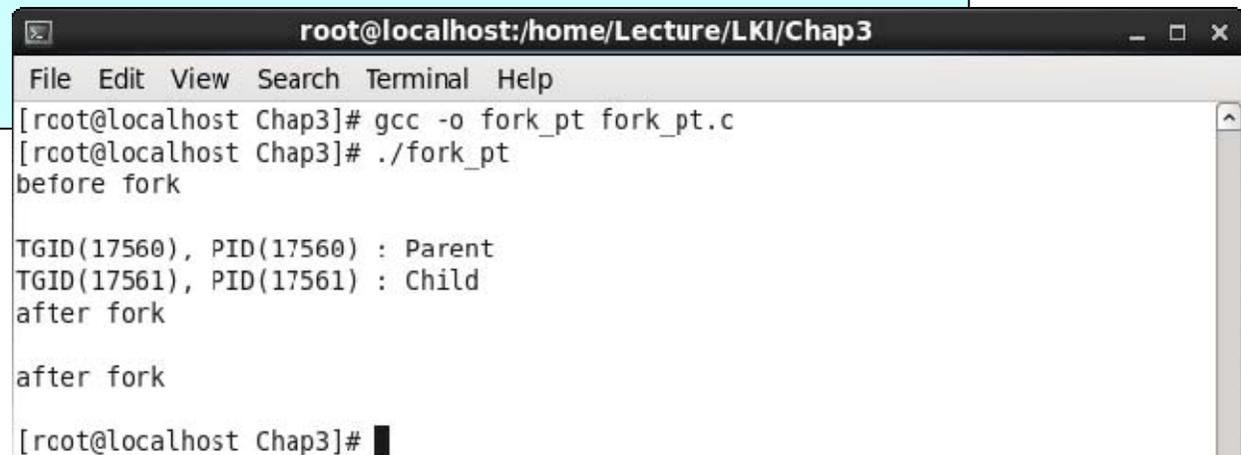


# Task example - fork

```
#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;
}
```



```
root@localhost:/home/Lecture/LKI/Chap3
File Edit View Search Terminal Help
[rcot@localhost Chap3]# gcc -o fork_pt fork_pt.c
[rcot@localhost Chap3]# ./fork_pt
before fork

TGID(17560), PID(17560) : Parent
TGID(17561), PID(17561) : Child
after fork

after fork

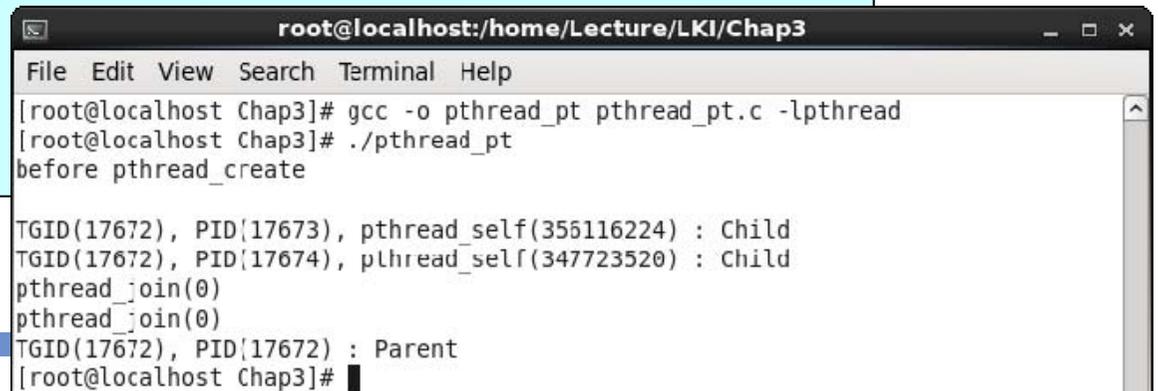
[rcot@localhost Chap3]#
```

# Task example - pthread

11

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
void *t_function(void *data)
{
    int id;      int i=0;      pthread_t t_id;
    id = *((int *)data);
    printf("TGID(%d), PID(%d), pthread_self(%d): Child %d\n", getpid(), syscall(__NR_gettid), pthread_self());
    sleep(2);
}
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("TGID(%d), PID(%d): Parent %d\n",
           getpid(), syscall(__NR_gettid));
    return 0;
}
```



```
root@localhost:/home/Lecture/LKI/Chap3
File Edit View Search Terminal Help
[root@localhost Chap3]# gcc -o pthread_pt pthread_pt.c -lpthread
[root@localhost Chap3]# ./pthread_pt
before pthread_create
TGID(17672), PID(17673), pthread_self(356116224) : Child
TGID(17672), PID(17674), pthread_self(347723520) : Child
pthread_join(0)
pthread_join(0)
TGID(17672), PID(17672) : Parent
[root@localhost Chap3]#
```

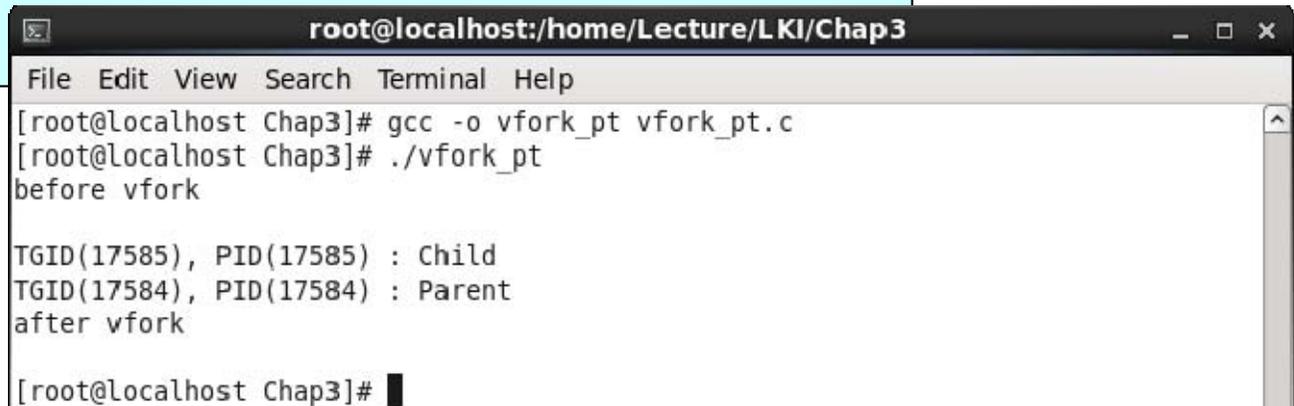
# Task example - vfork

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

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("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");
    exit(0);
}
```



```
root@localhost:/home/Lecture/LKI/Chap3
File Edit View Search Terminal Help
[root@localhost Chap3]# gcc -o vfork_pt vfork_pt.c
[root@localhost Chap3]# ./vfork_pt
before vfork

TGID(17585), PID(17585) : Child
TGID(17584), PID(17584) : Parent
after vfork

[root@localhost Chap3]#
```

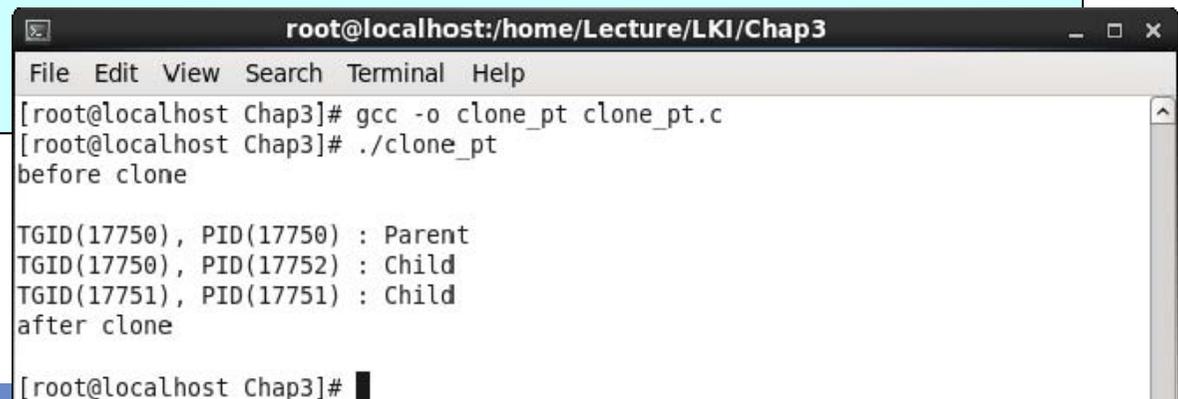
# Task example - clone

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <sched.h>
int sub_a(void *arg)
{
    printf("TGID(%d), PID(%d): Child %d\n", getpid(), syscall(__NR_gettid));
    sleep(2);
    return 0;
}
int main(void)
{
    int child_a_stack[4096], child_b_stack[4096];

    printf("before clone\n");
    printf("TGID(%d), PID(%d)\n", getpid(), syscall(__NR_gettid));

    clone (sub_a, (void *) (child_a_stack+4095), CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID, NULL);
    clone (sub_a, (void *) (child_b_stack+4095), CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID, NULL);
    sleep(1);
    printf("after clone %d %d\n");
    return 0;
}
```

CLONE\_THREAD  
option 추가 시 같은  
tgid를 가짐



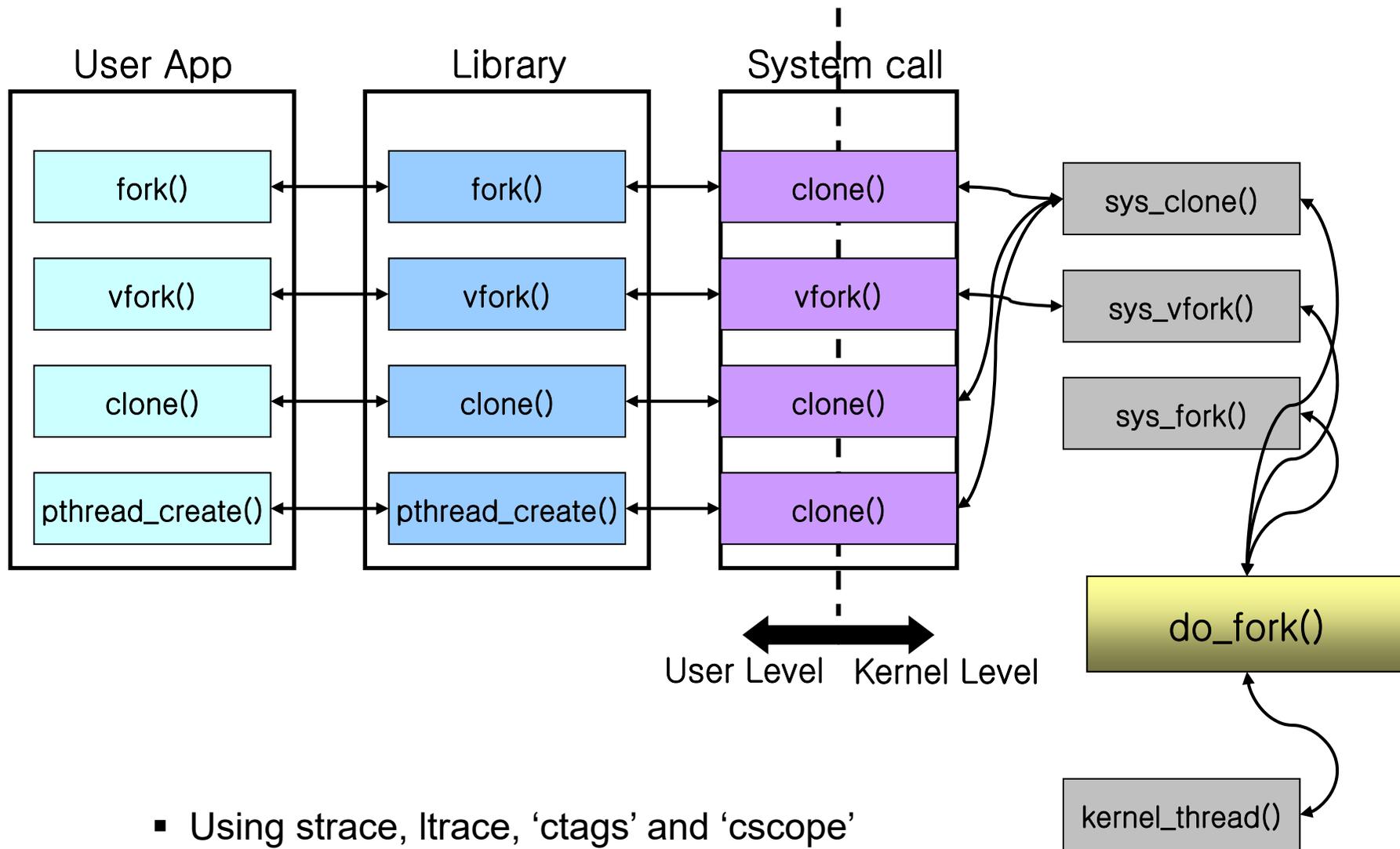
```
root@localhost:/home/Lecture/LKI/Chap3
File Edit View Search Terminal Help
[root@localhost Chap3]# gcc -o clone_pt clone_pt.c
[root@localhost Chap3]# ./clone_pt
before clone

TGID(17750), PID(17750) : Parent
TGID(17750), PID(17752) : Child
TGID(17751), PID(17751) : Child
after clone

[root@localhost Chap3]#
```

# Task implementation in Linux

## ■ Flow controls



# Task example – kernel thread

```
int __init module_thread_init(void);
void __exit module_thread_exit(void);

void test_kernel_thread(void *arg){
    int val, i, j;
    val = (int)(long)arg;
    printk("<0>kernel_thread %d CPU : %d\n", val, smp_processor_id());
    for(j=0 ; j < 100000000 ; j++){
        for(i=0 ; i < 100000000 ; i++){
            printk("<0>kernel_thread %d CPU : %d\n", val, smp_processor_id());
        }
    }
}

int __init module_thread_init(){
    int i;
    for(i = 0 ; i < 4 ; i++){
        kernel_thread((int (*)(void *))test_kernel_thread, (void *)i, 0);
    }
    return 0;
}

void __exit module_thread_exit(){
    printk("<0>Module Thread Test Exit\n");
}

module_init(module_thread_init);
module_exit(module_thread_exit);
```

# Task example – kernel thread

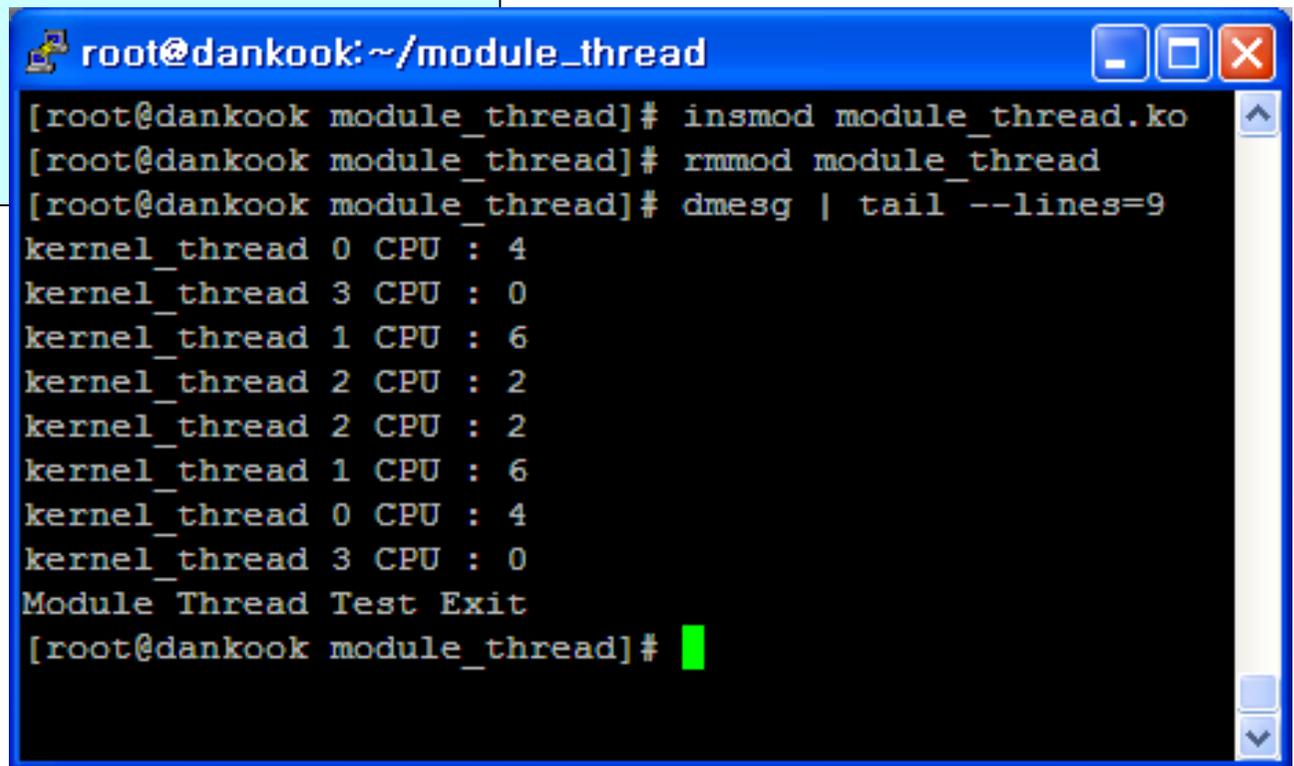
16

```
obj-m      := module_thread.o

KDIR       := /lib/modules/$(shell uname -r)/build
PWD        := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
    rm -rf *.ko
    rm -rf *.mod.*
    rm -rf *.cmd
    rm -rf *.o
```

A terminal window titled 'root@dankook:~/module\_thread' with standard window controls. The terminal shows the following commands and output:

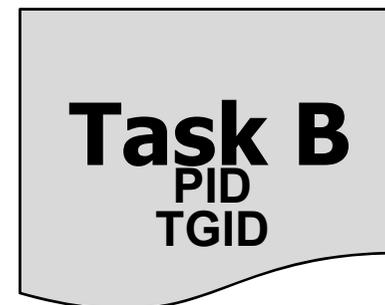
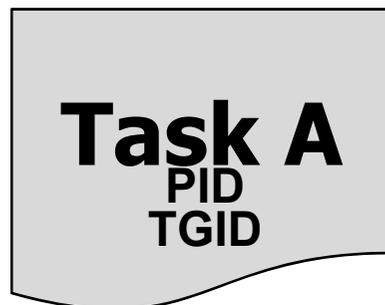
```
[root@dankook module_thread]# insmod module_thread.ko
[root@dankook module_thread]# rmmmod module_thread
[root@dankook module_thread]# dmesg | tail --lines=9
kernel_thread 0 CPU : 4
kernel_thread 3 CPU : 0
kernel_thread 1 CPU : 6
kernel_thread 2 CPU : 2
kernel_thread 2 CPU : 2
kernel_thread 1 CPU : 6
kernel_thread 0 CPU : 4
kernel_thread 3 CPU : 0
Module Thread Test Exit
[root@dankook module_thread]#
```

# Task, Process and Thread

---

## 1. PID

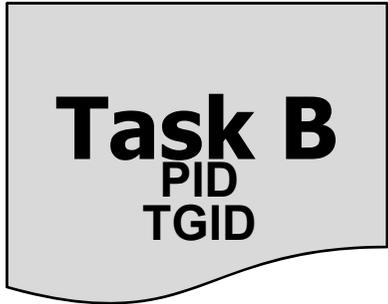
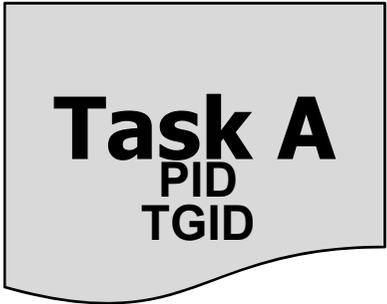
2. POSIX: 한 process내의 thread는 동일한 PID를 공유해야 한다



# Task, Process and Thread

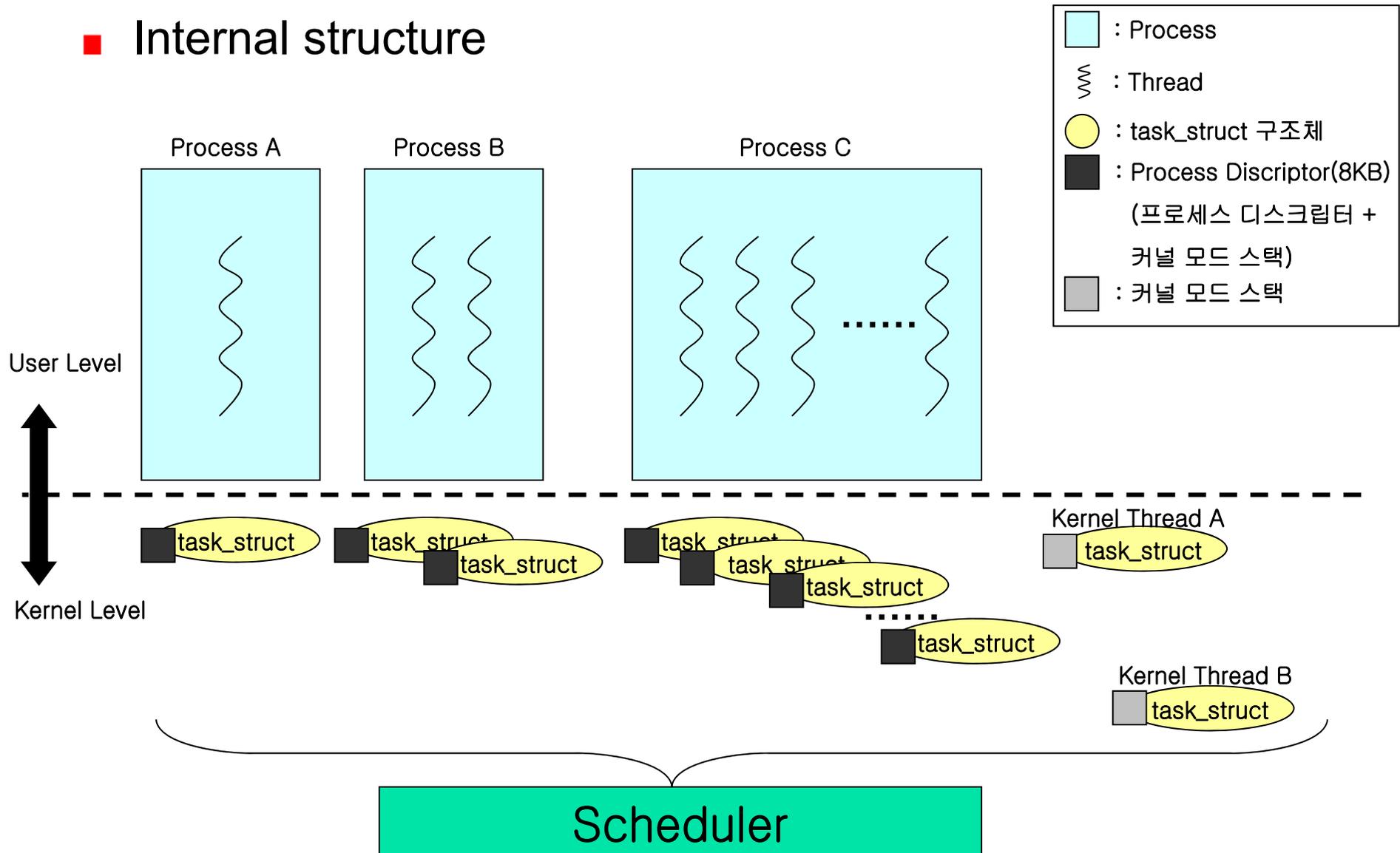
---

```
struct task_struct {  
    pid,  
    tgid,  
    ...  
};
```



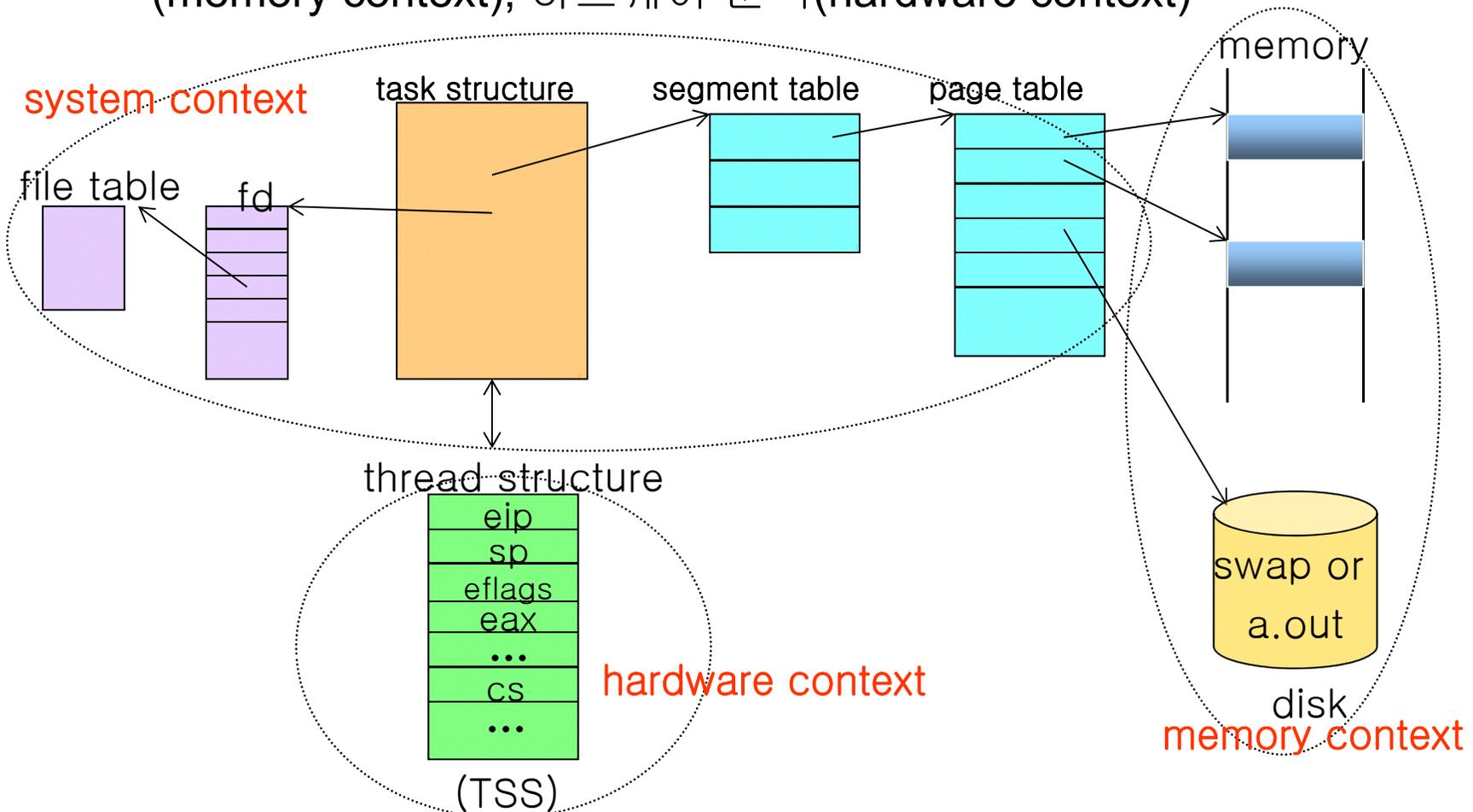
...

## Internal structure



## ■ 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\_cptr, next\_task, next\_run
  - ✓ 스케줄링 정보: policy, priority, counter, rt\_priority, need\_resched
  - ✓ 태스크가 접근한 파일 정보: file descriptor
  - ✓ 시그널 정보
  - ✓ 그 외: 수행 시간 정보, 수행 파일 이름, 등등 (kernel dependent)

# 태스크 생성과 전이(커널 수준으로 진입/수면)의 예

```
/* 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);
}
```

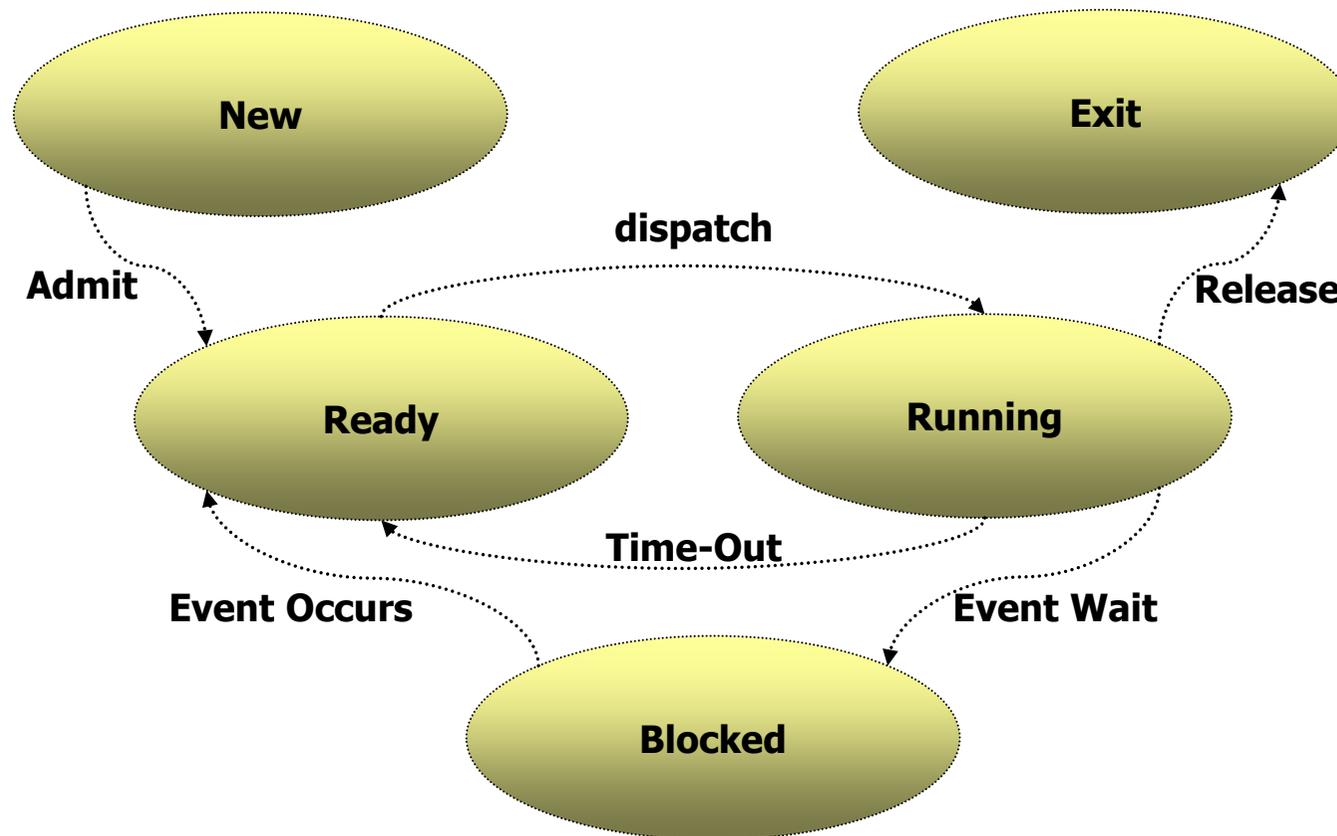
(Source : Adv. programming in the UNIX Env., pgm 8.1)

# Task 생성과 주소공간 예제 결과

```
choijm@embedded:~/syspro/exam_task/4_fork
파일(F) 편집(E) 보기(V) 터미널(T) 가기(G) 도움말(H)
CLONE(2) Linux Programmer's Manual CLONE(2)
이름
__clone - 프로세스 생성
사용법
#include <unistd.h>
int __clone(int (*fn)(void*), void* arg, unsigned long flags, void* arg2, void* arg3, void* arg4);
설명
__clone는 fork(2)와 유사한 시스템 호출이다. __clone는 메모리에서 자식 프로세스를 생성한다.
fn(arg).
되는 함수

choijm@embedded:~/syspro/exam_task/5_fork2
파일(F) 편집(E) 보기(V) 터미널(T) 가기(G) 도움말(H)
[choijm@embedded 5_fork2]$
[choijm@embedded 5_fork2]$ make
gcc -c fork_test2.c
gcc -g -o fork_test2 fork_test2.c
[choijm@embedded 5_fork2]$
[choijm@embedded 5_fork2]$ ls
fork_test2 fork_test2.c fork_test2.o makefile
[choijm@embedded 5_fork2]$
[choijm@embedded 5_fork2]$ ./fork_test2
a write to stdout
before fork
pid = 8328, glob = 7, var = 89
pid = 8327, glob = 6, var = 88
[choijm@embedded 5_fork2]$
[choijm@embedded 5_fork2]$
[choijm@embedded 5_fork2]$
```

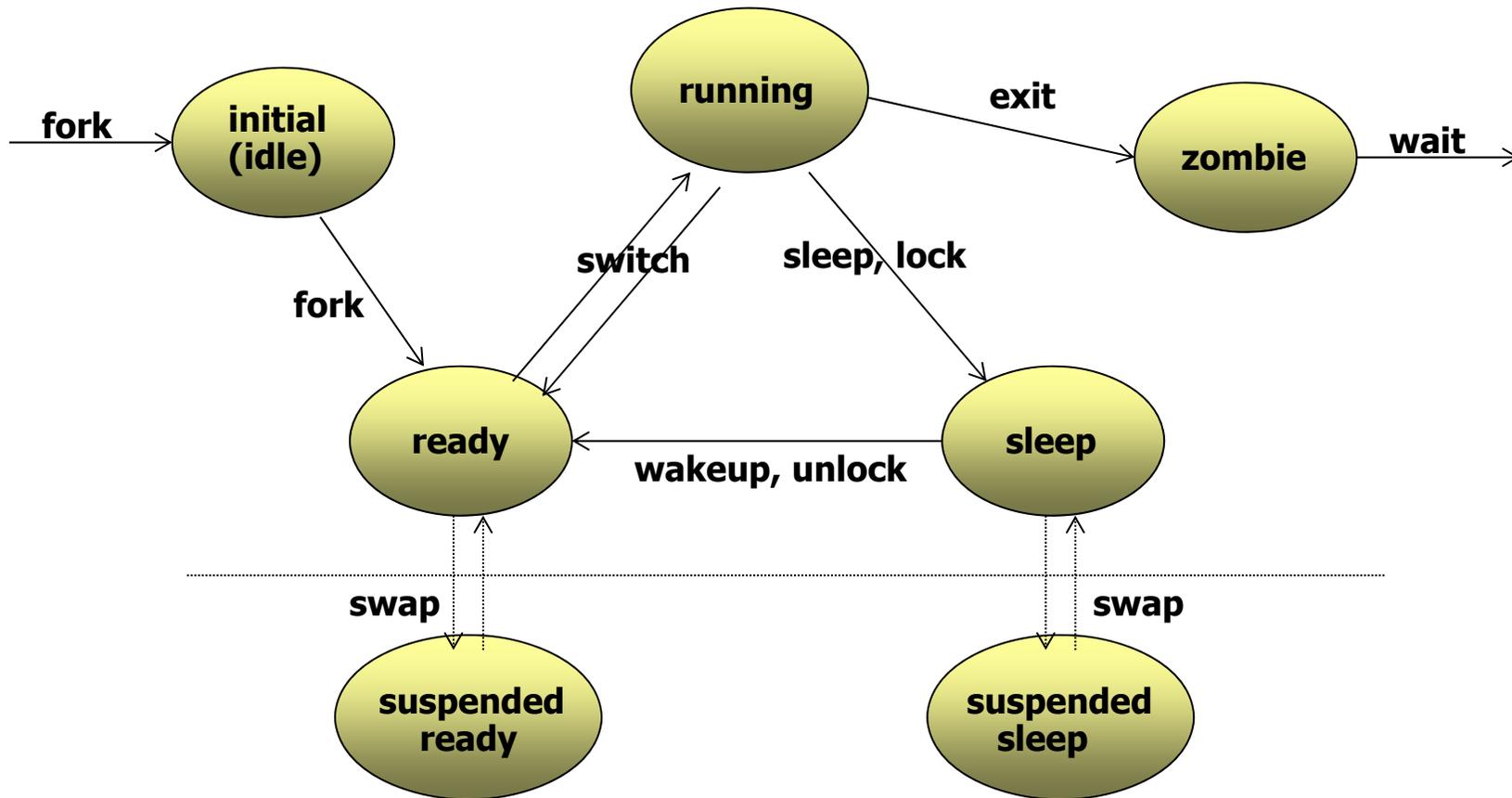
# Process State Diagram(1/2)



👉 **Five-State Process Model**

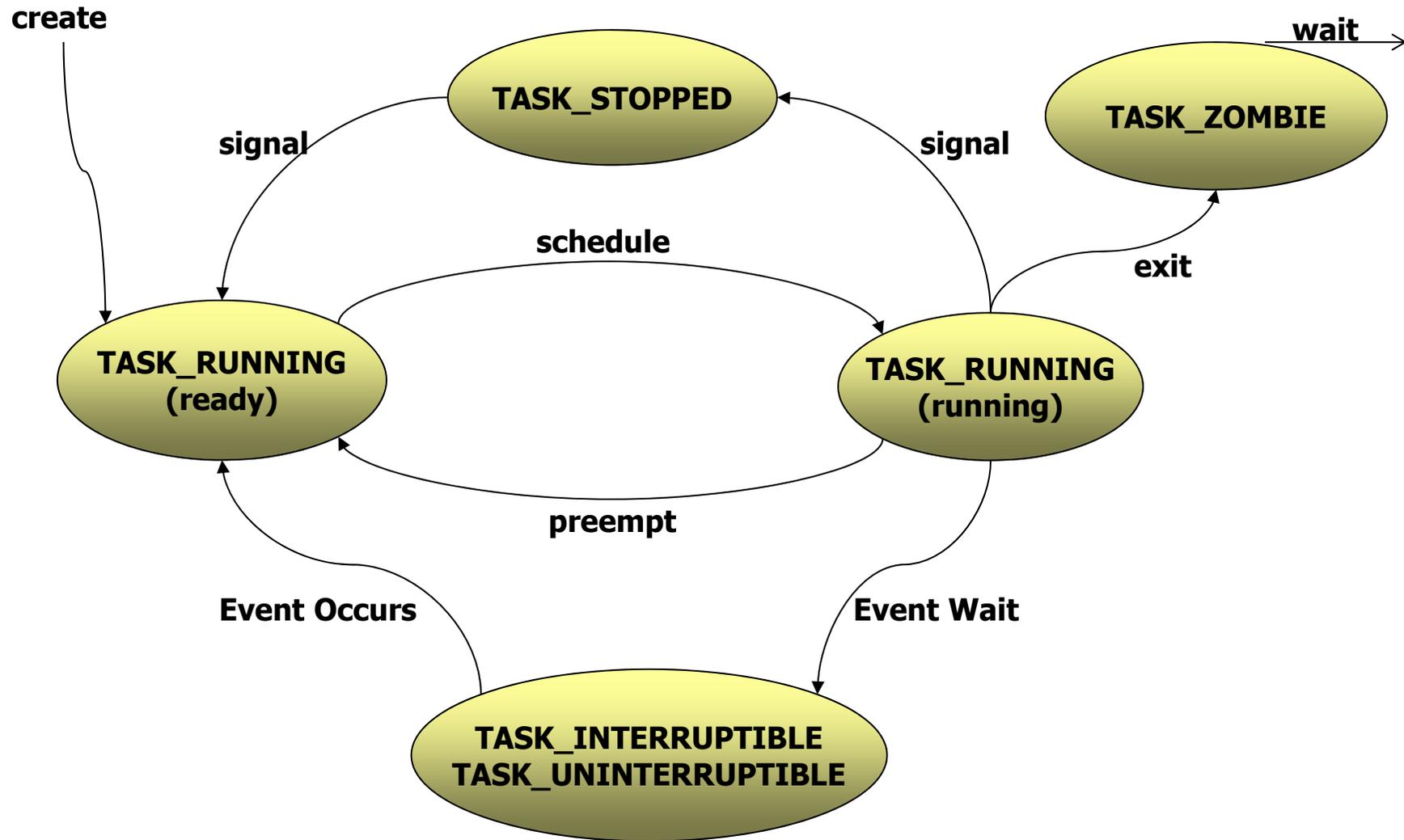
(Source : OS(stallings))

# Process State Diagram(2/2)

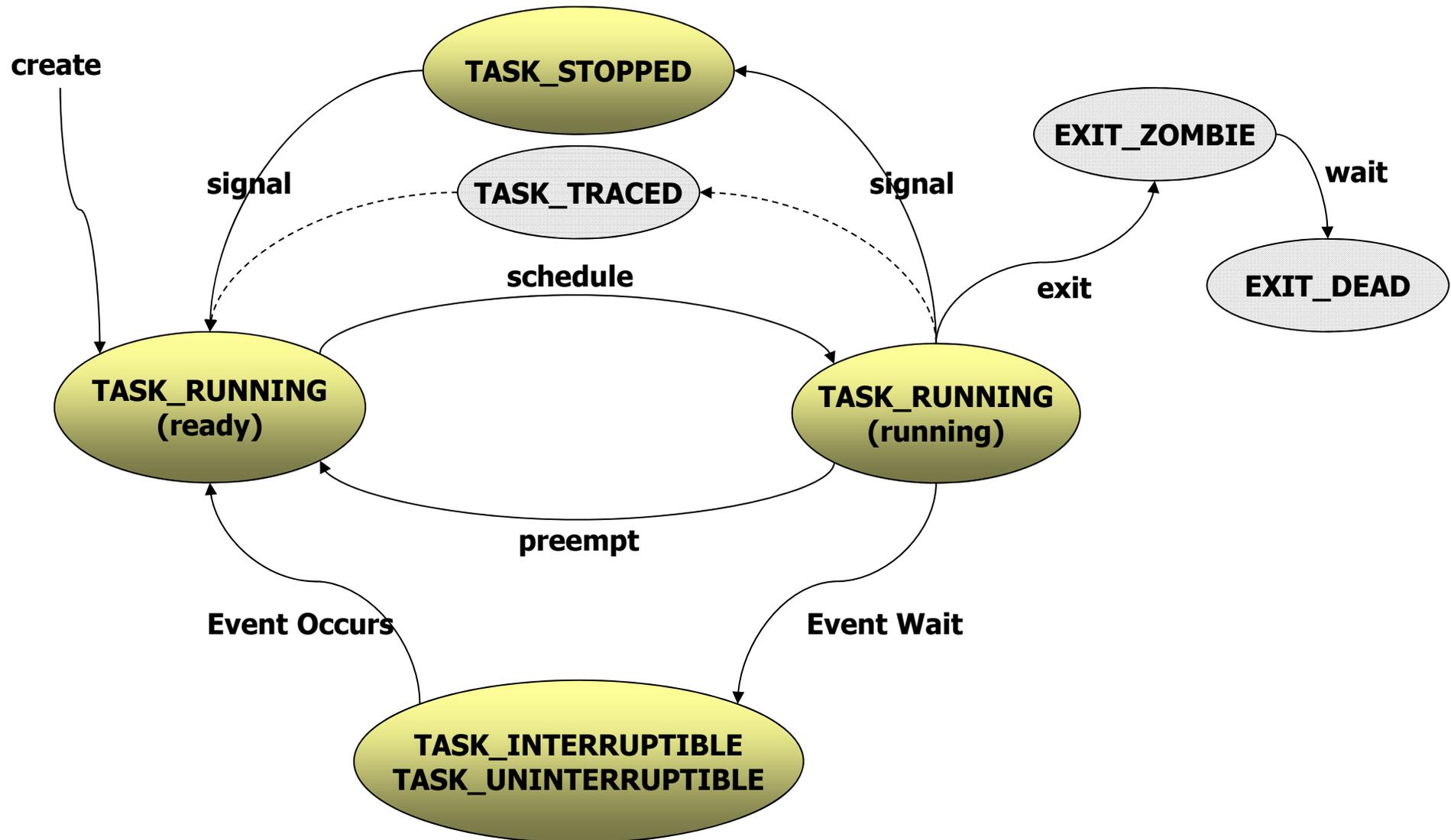


(Source : UNIX Internals)

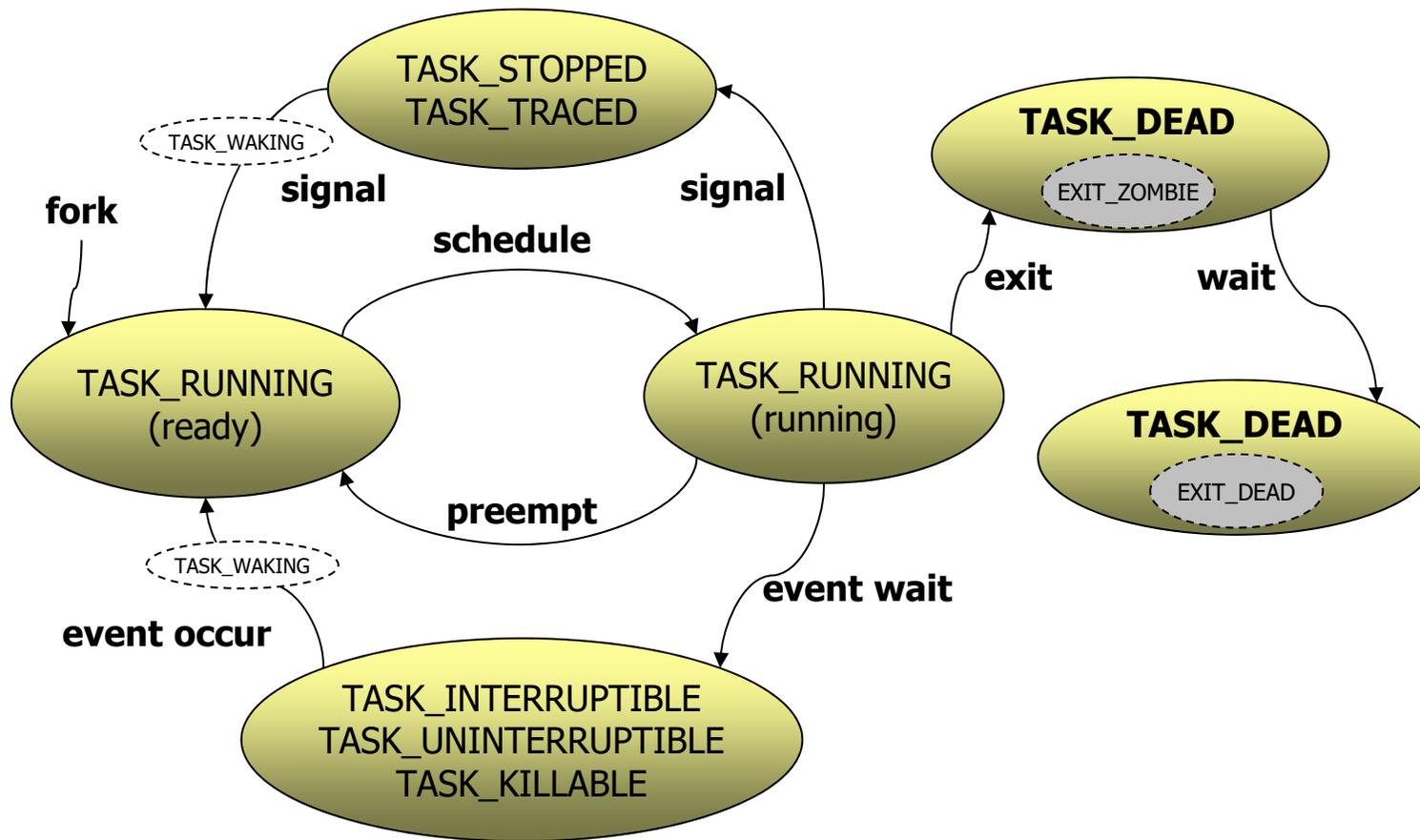
# Task State Diagram of LINUX(2.4)



# Task State Diagram of LINUX(2.6)

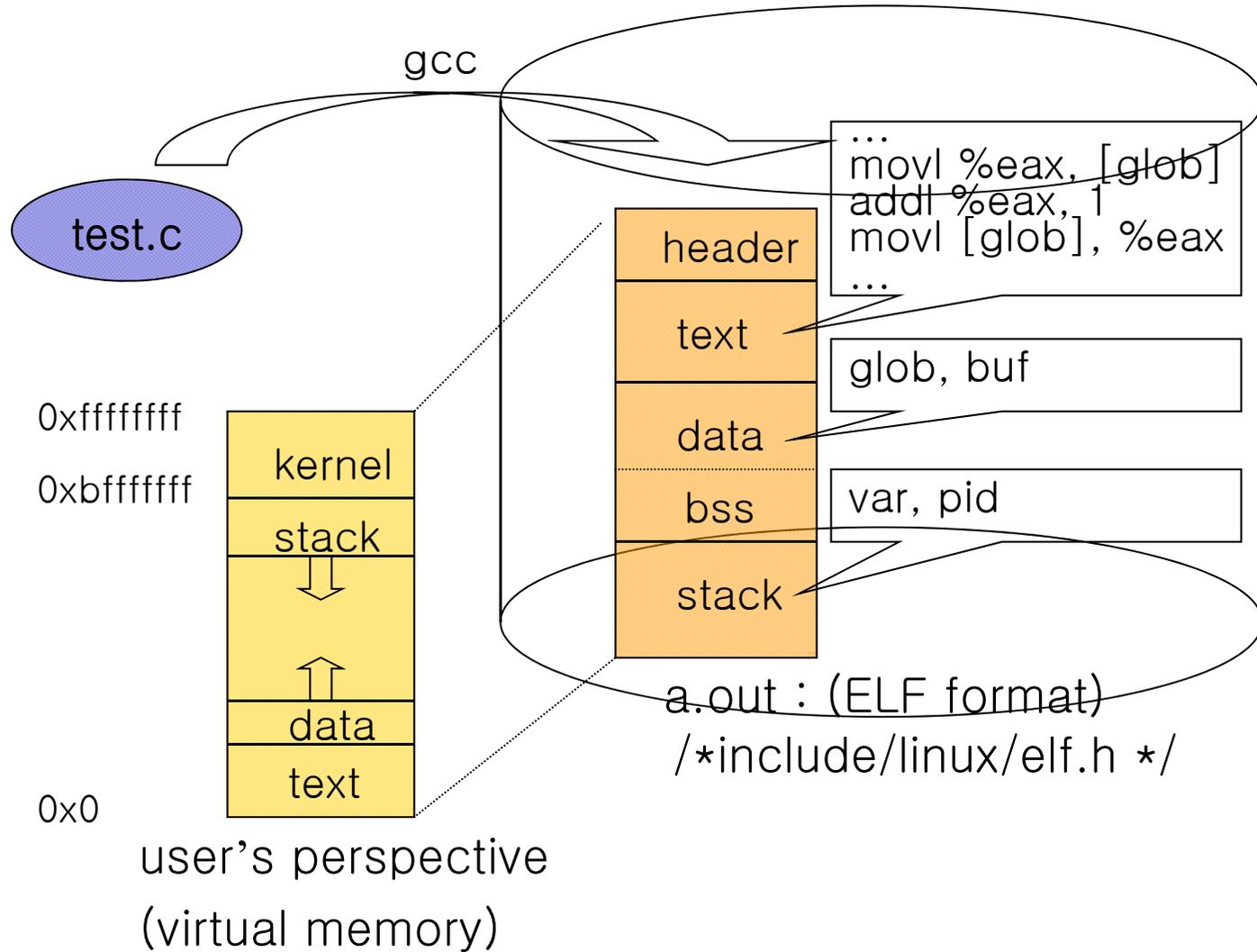


# Task State Diagram of LINUX(3.18)

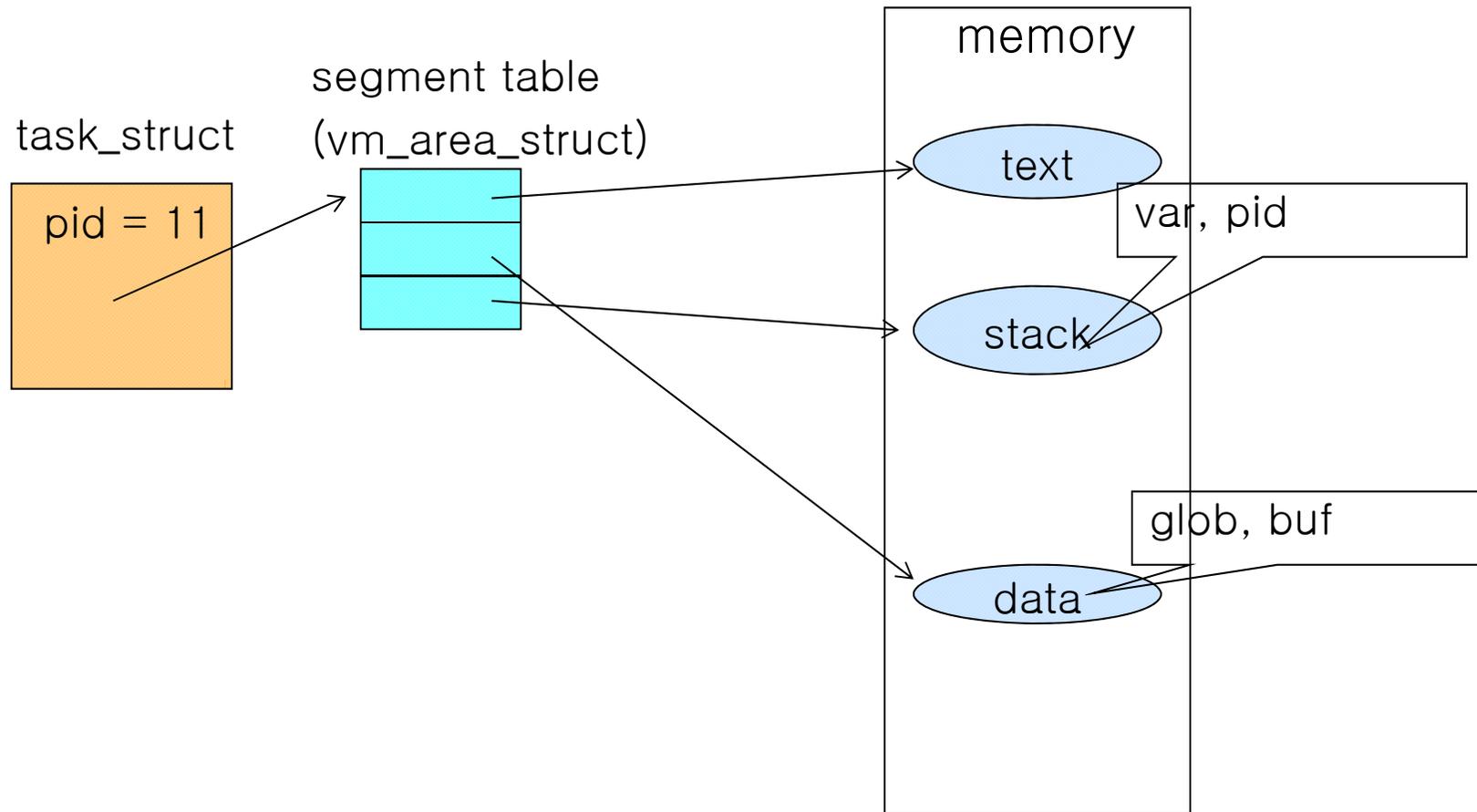


# Memory Context (1/5)

- fork internal : compile results

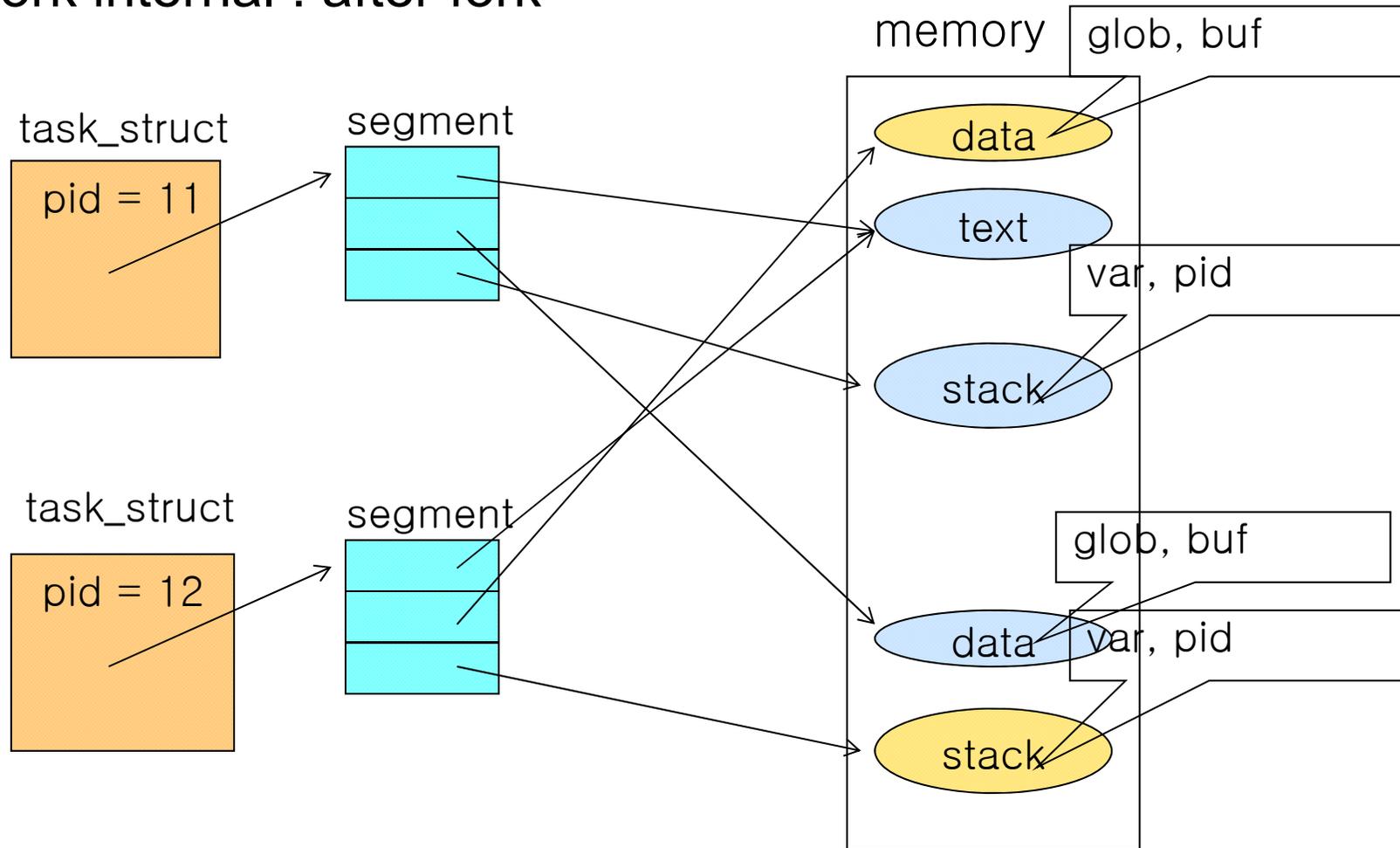


- fork internal : after loading (after run a.out) & before fork



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

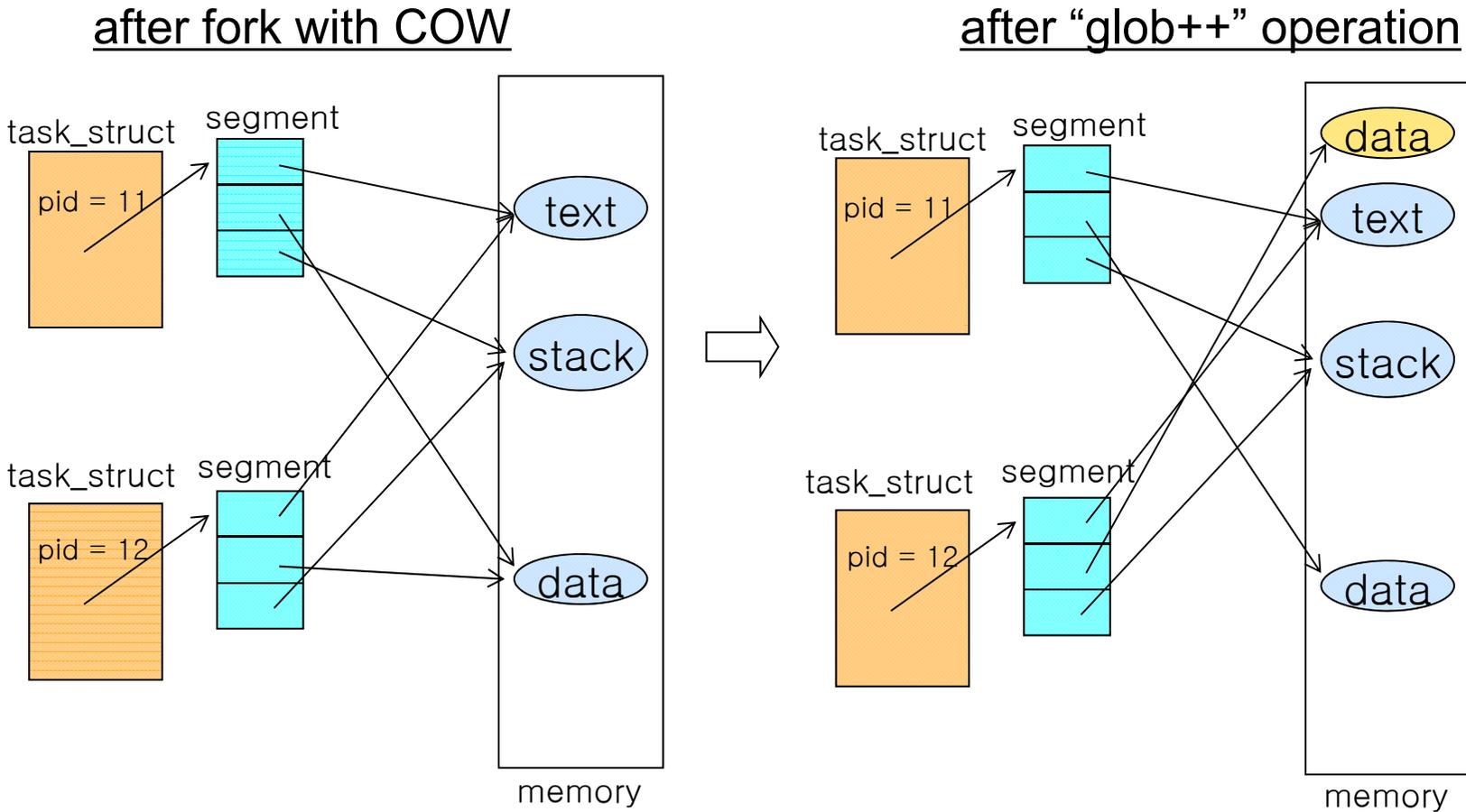
- fork internal : after fork



✓ address space : basic protection barrier

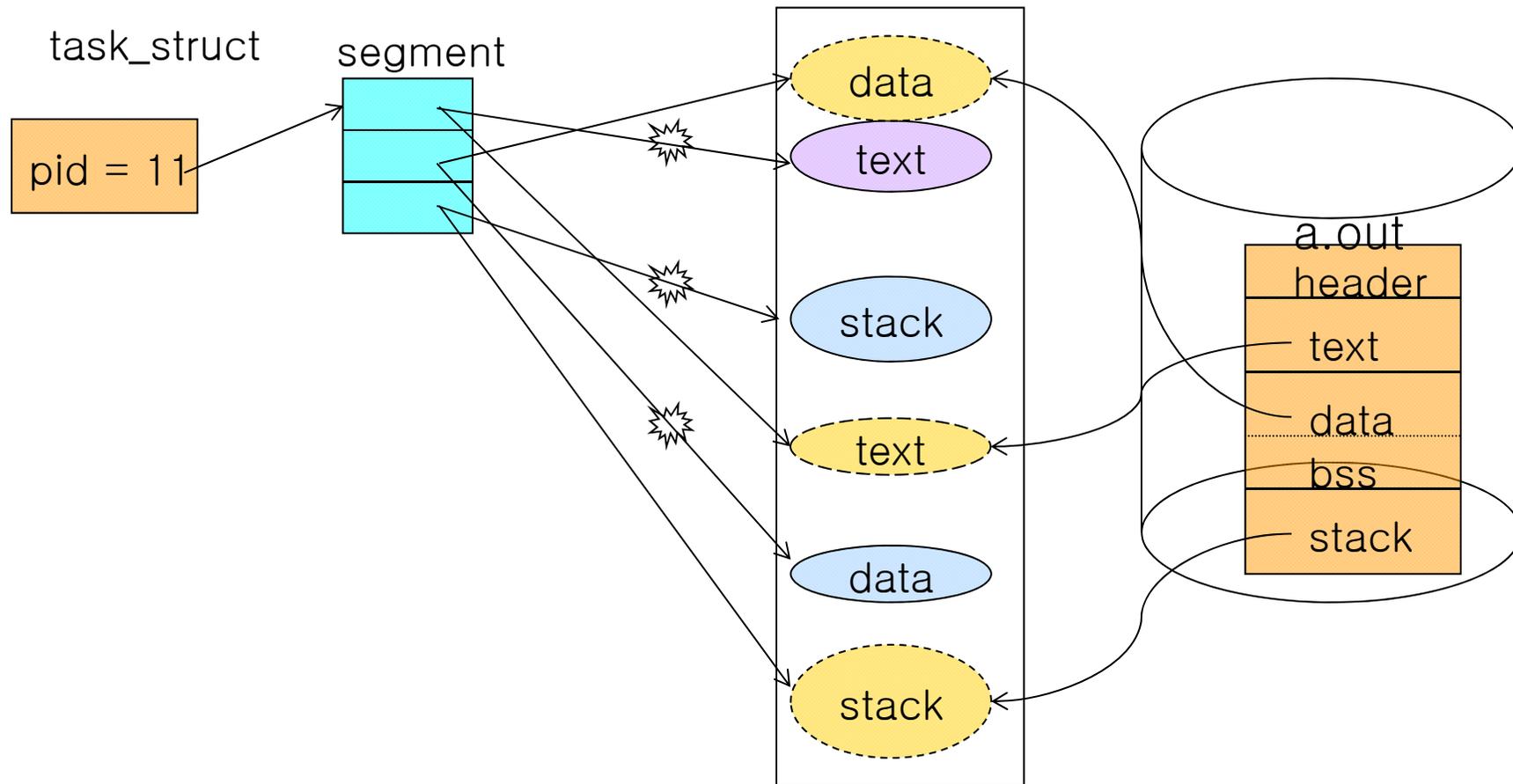
# Memory Context (4/5)

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



# Memory Context (5/5)

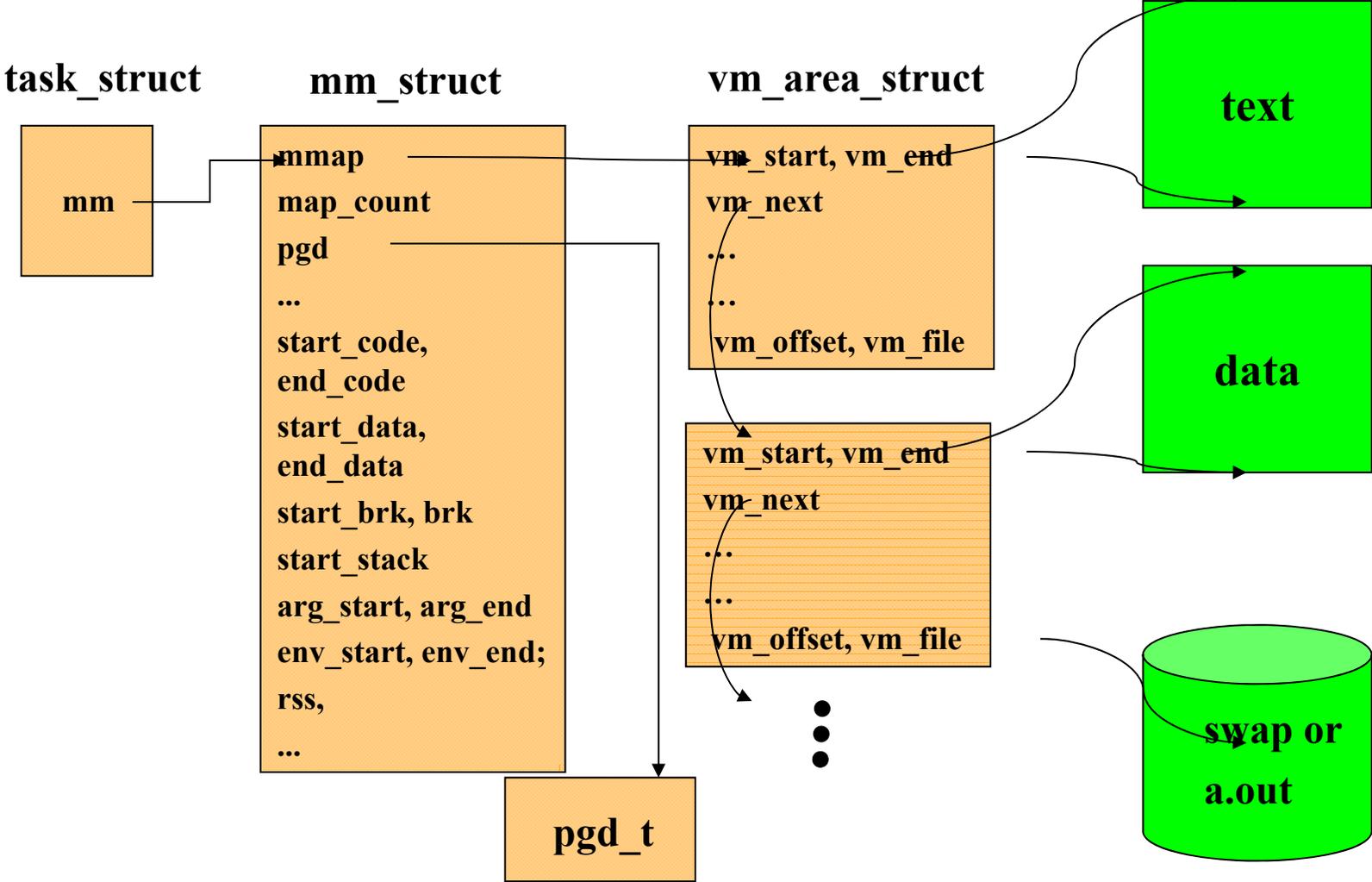
- 새로운 프로그램 수행: `execve()` internal memory



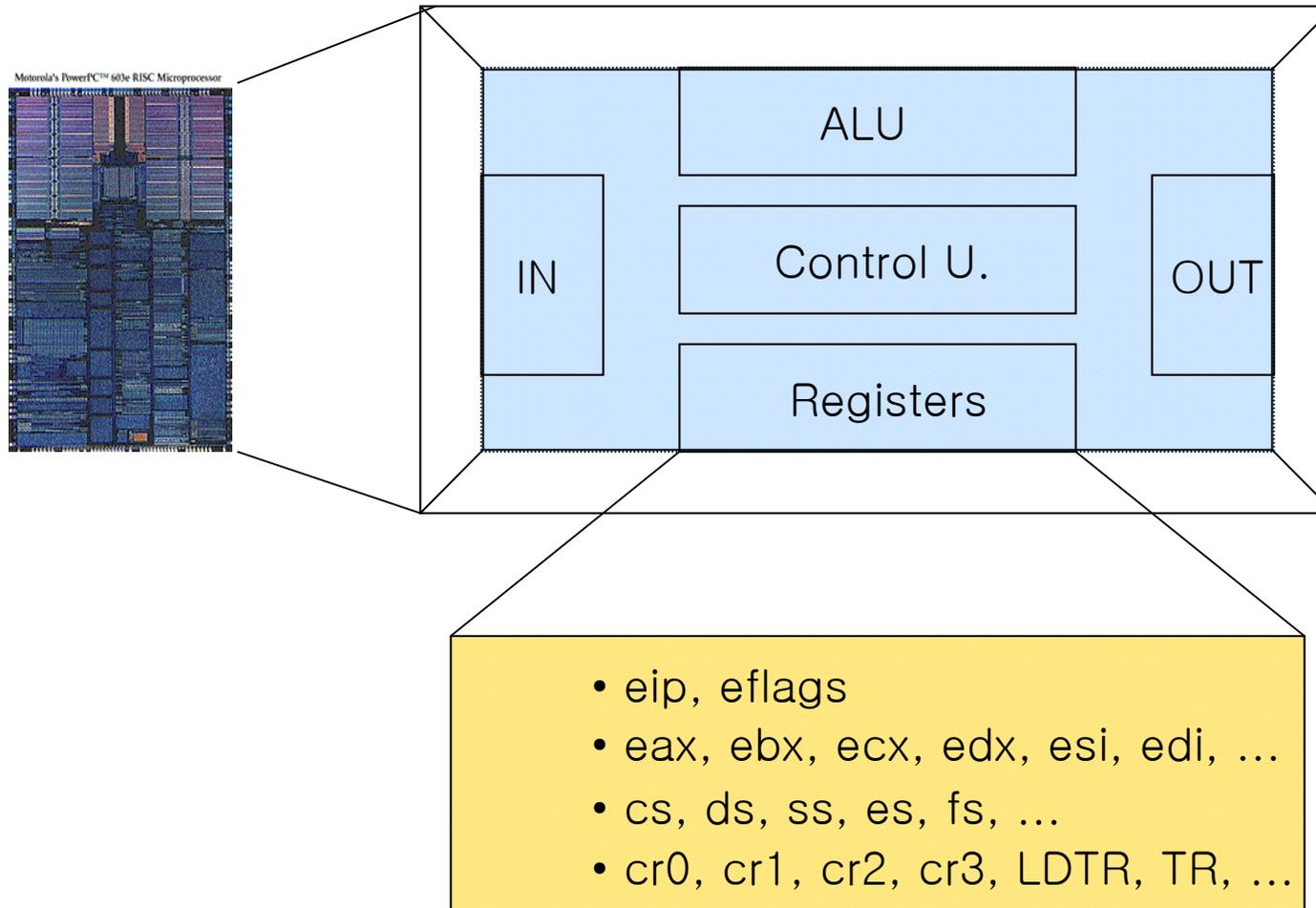
# Virtual Memory Describing

- 메모리 관리 자료 구조

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

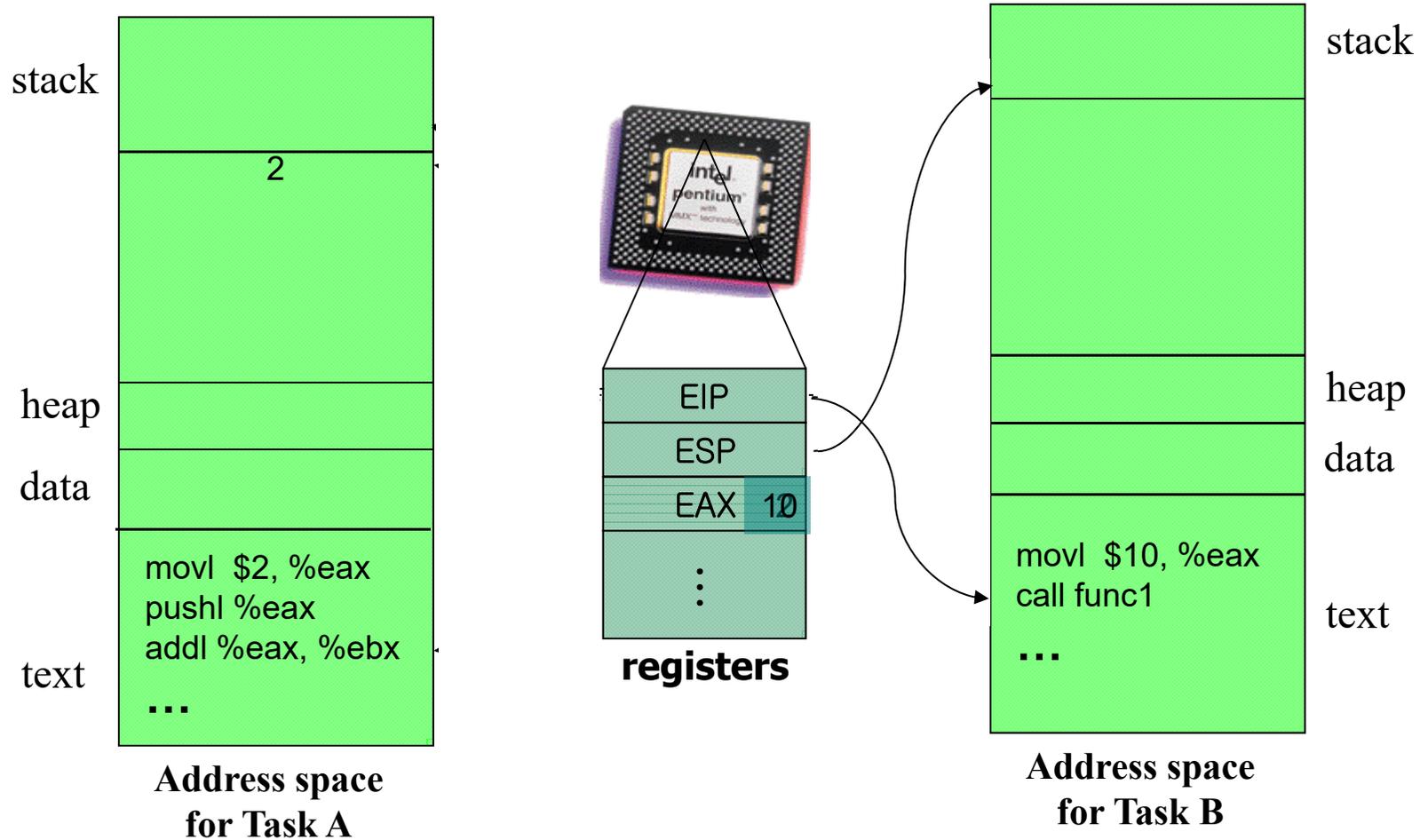


- brief reminds the 80x86 architecture



# Hardware Context(2/3)

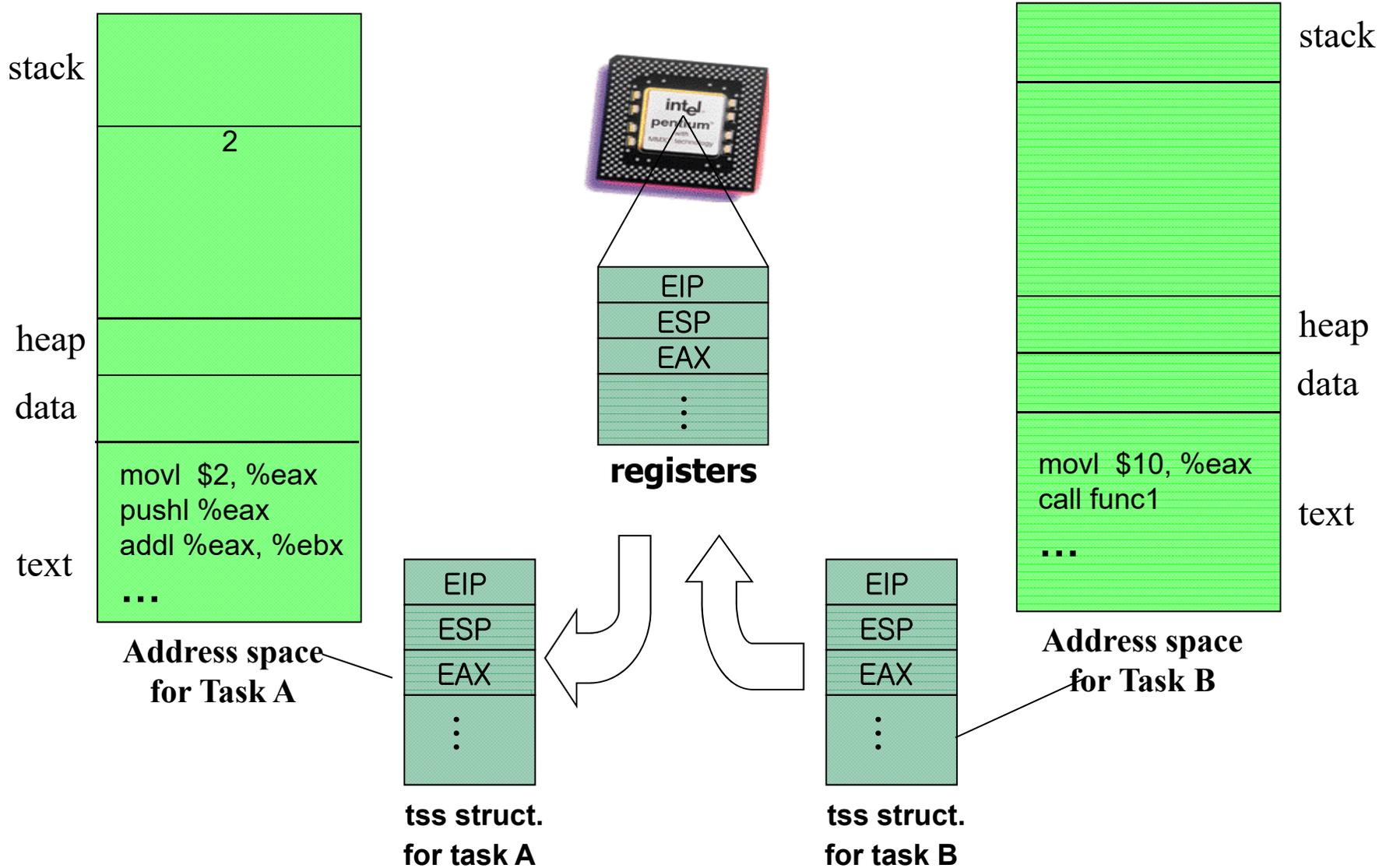
■ 쓰레드 자료 구조 : CPU 추상화



- ☞ 다시 **Task A**가 수행되려면?
- ☞ 문맥 교환(**Context Switch**) → **thread structure**

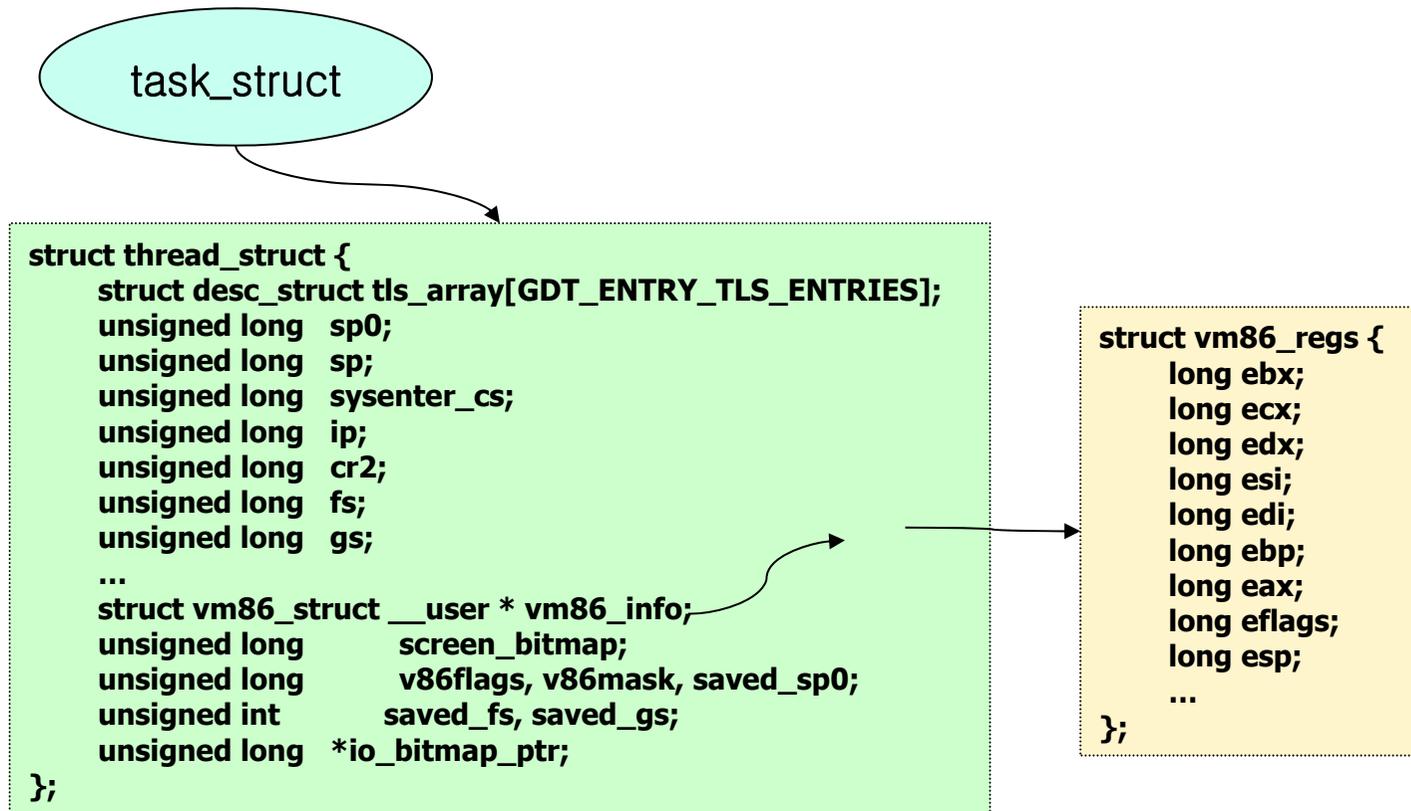
# Hardware Context(3/3)

- 스레드 자료 구조 : CPU 추상화



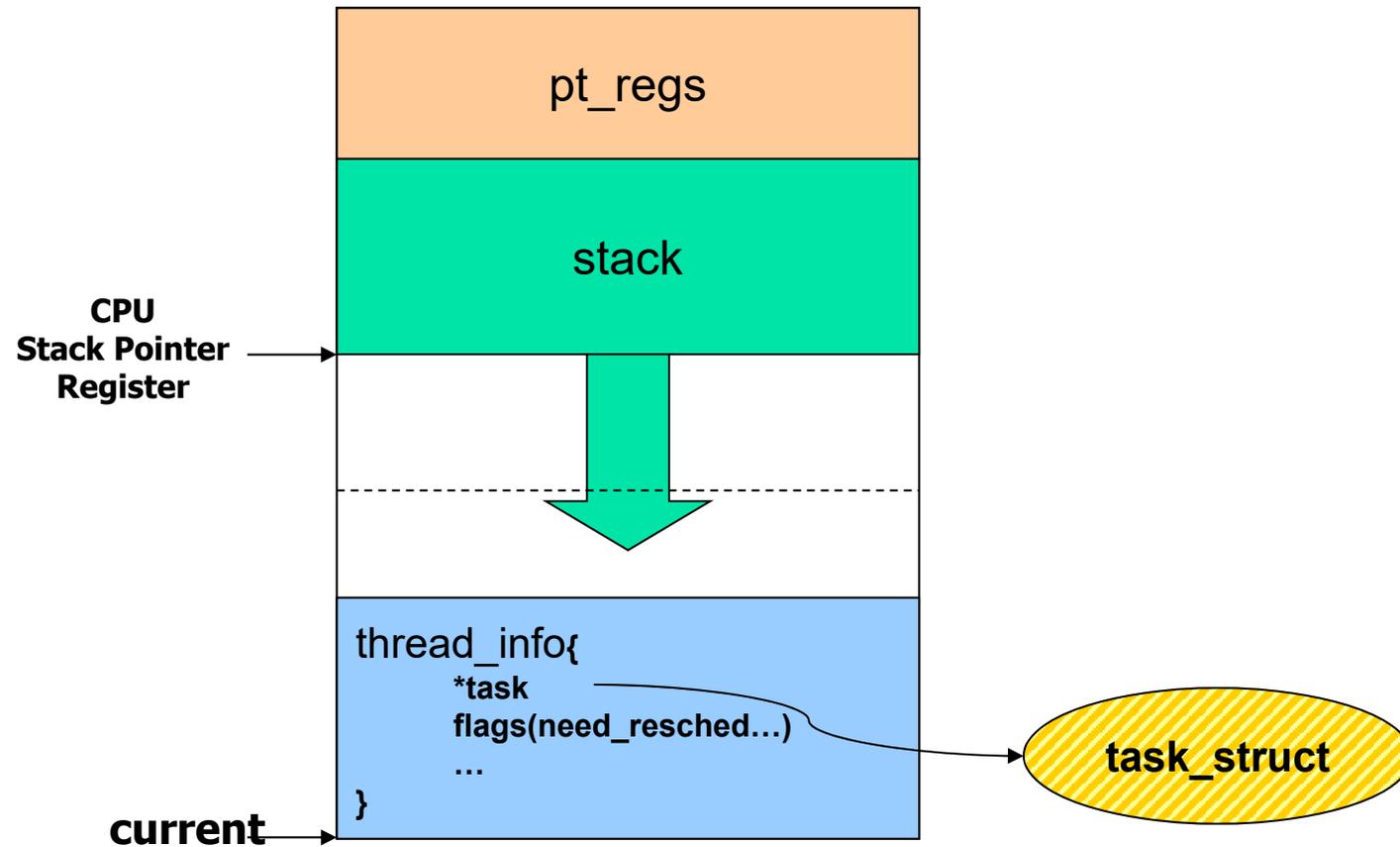
# task\_struct & H/W context

- 쓰레드 자료 구조 : CPU 추상화



## ■ Thread union

- ✓ 커널은 각 태스크 별로 8KB메모리 할당
- ✓ thread\_info 구조체와 kernel stack
- ✓ alloc\_thread\_struct, free\_thread\_struct



# User VS kernel mode

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

```

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

```

```

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}

```

## LinuxThreads vs NPTL

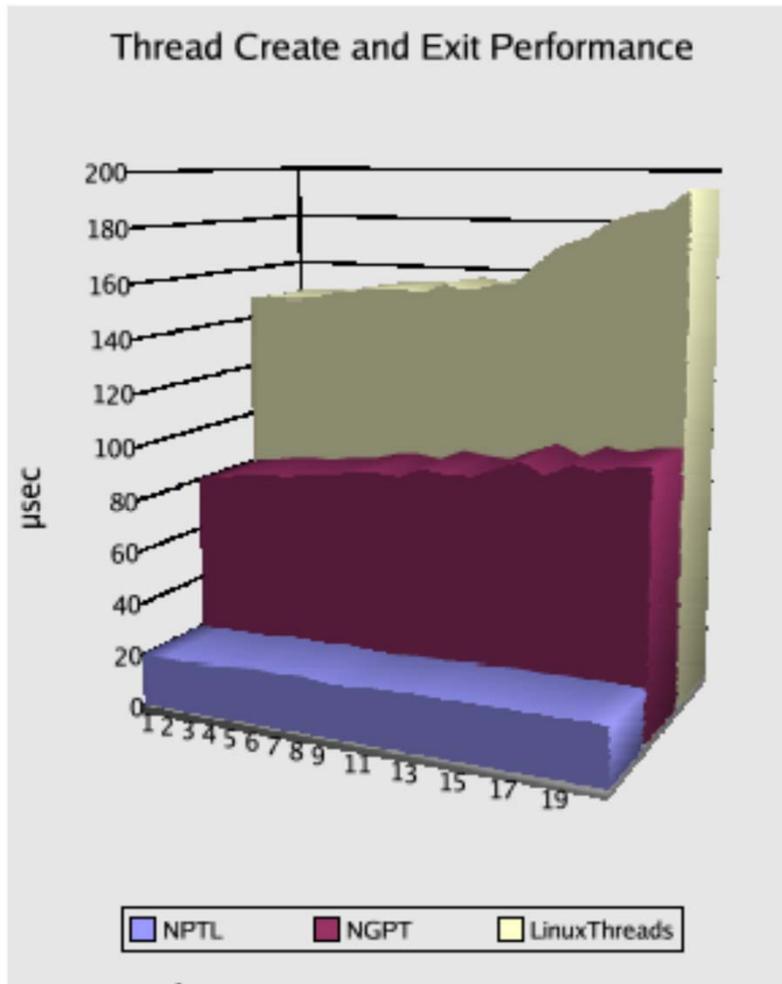


Figure 1: Varying number of Toplevel Threads

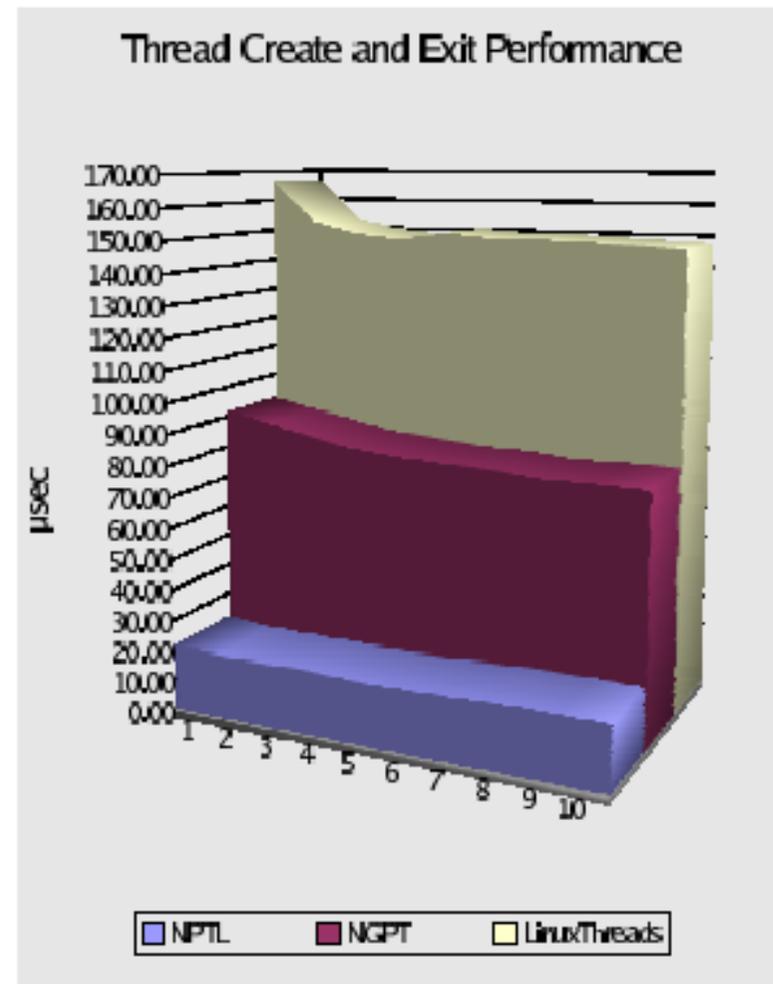


Figure 2: Varying number of Concurrent Children

# Task 계층 구조 확인

The image shows a terminal window on the left and a TRACE32 task manager window on the right. The terminal window displays the command `./nfs/sched` and the resulting task configuration. The configuration is divided into two sections: "Before set" and "FIFO set".

**Before set**  
Param.priority = 0  
Sched policy = 0

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

The TRACE32 window shows a list of tasks in the left pane, with a red box highlighting the following commands: `swapper`, `init`, `keventd`, `ksoftirqd_CPU0`, `kswapd`, `bdflush`, `kupdated`, `mtdblockd`, `mmcblockd`, `khubd`, `syslogd`, `inetd`, `bash`, `rpciod`, and `sched`.

The right pane shows the task hierarchy for `B::Task,DTask 0xC0194000` (swapper) and `B::Task,DTask 0xC02E2000` (init). The swapper task is in a "running" state and has a "parent" of `init`. The init task is in a "sleeping" state and has a "parent" of `swapper`. The swapper task also has a "youngest child" of `rpciod`.

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

# Task 계층 구조 확인

The screenshot displays two windows from Windows Task Manager, showing the hierarchy of processes. The first window, titled 'B::Task, DTask 0xC0C54000', shows a 'bash' process (PID 104) in a 'sleeping' state. Its parent is 'init' (PID 0), which is the youngest child of 'sched' (PID 0). 'sched' is the youngest child of 'rpciod' (PID 0), which is the younger sibling of 'inetd' (PID 0). The second window, titled 'B::Task, DTask 0xC0C36000', shows a 'sched' process (PID 125) in a 'current' state. Its parent is 'bash' (PID 0), which is the youngest child of '-' (PID 0). The 'arguments', 'environment', 'open files', 'addresses', 'code files', and 'times' sections for the 'bash' process are also visible.

```
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  000001D0  C0C4C000  ttyS2  /bin/bash

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

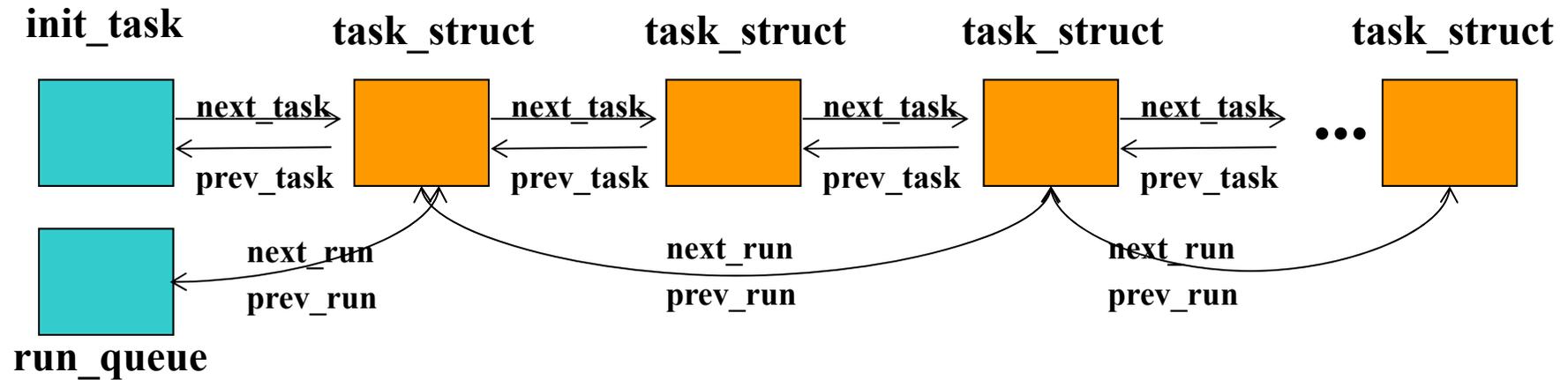
B::Task, DTask 0xC0C36000
magic  command  state  uid  pid  spaceid  tty  flags  nic
C0C36000  sched  current  0.  125.  007D  S-71  00000100

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

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

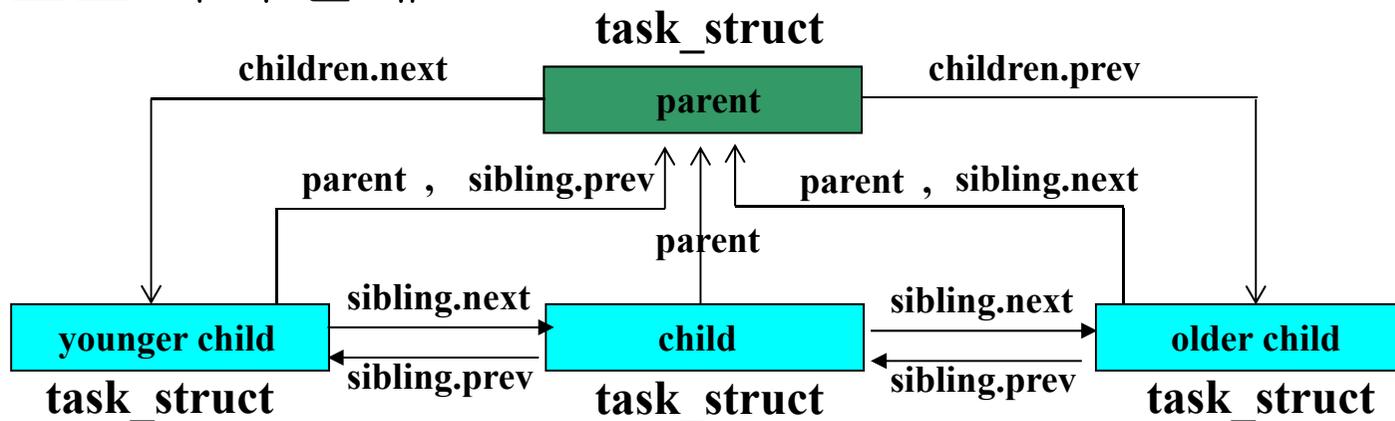
+ arguments
+ environment
+ open files
+ addresses
+ code files
+ times
```

## ■ 태스크 연결 구조

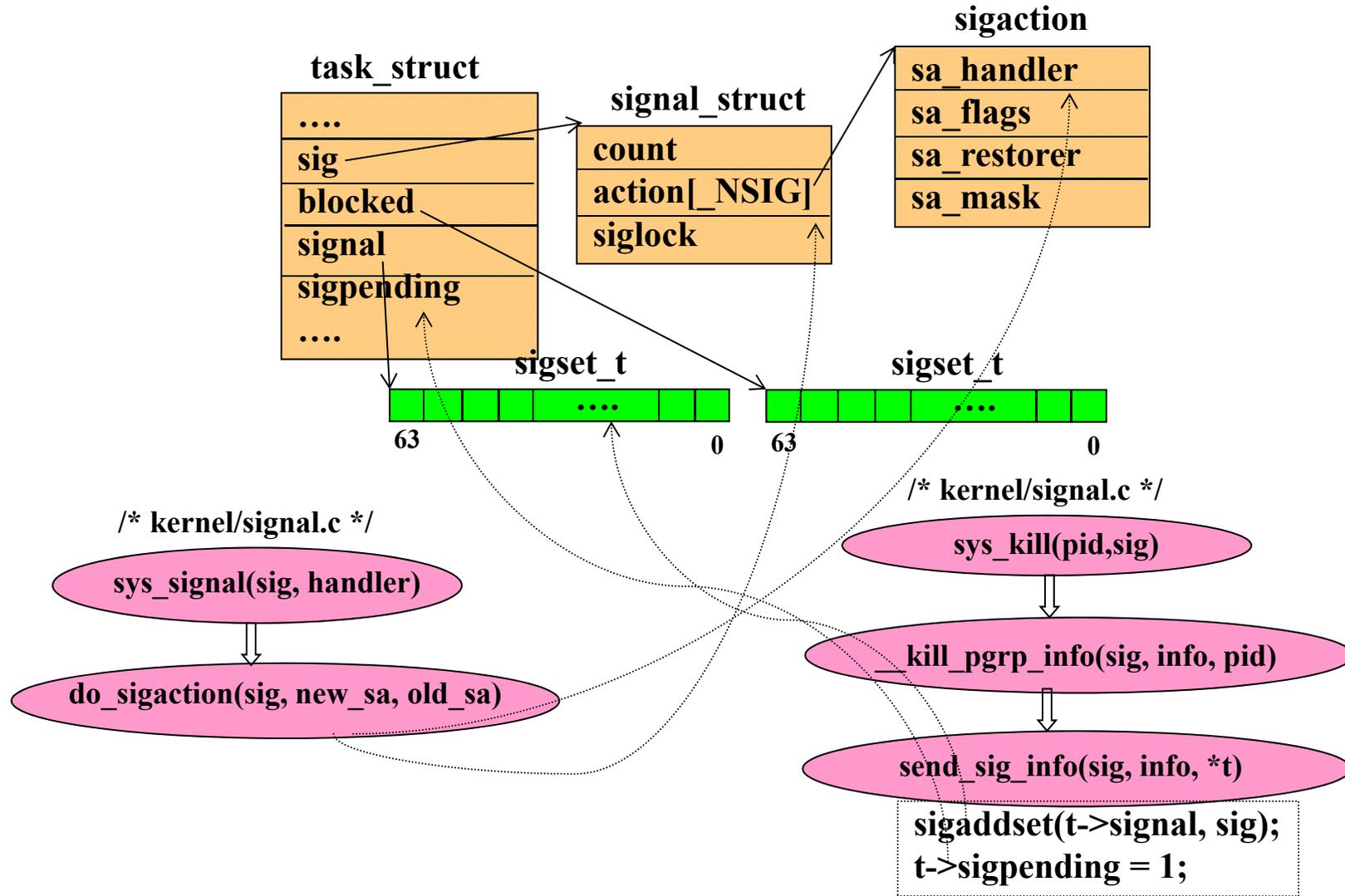


☞ where is run\_queue ? prev\_task? next\_task?

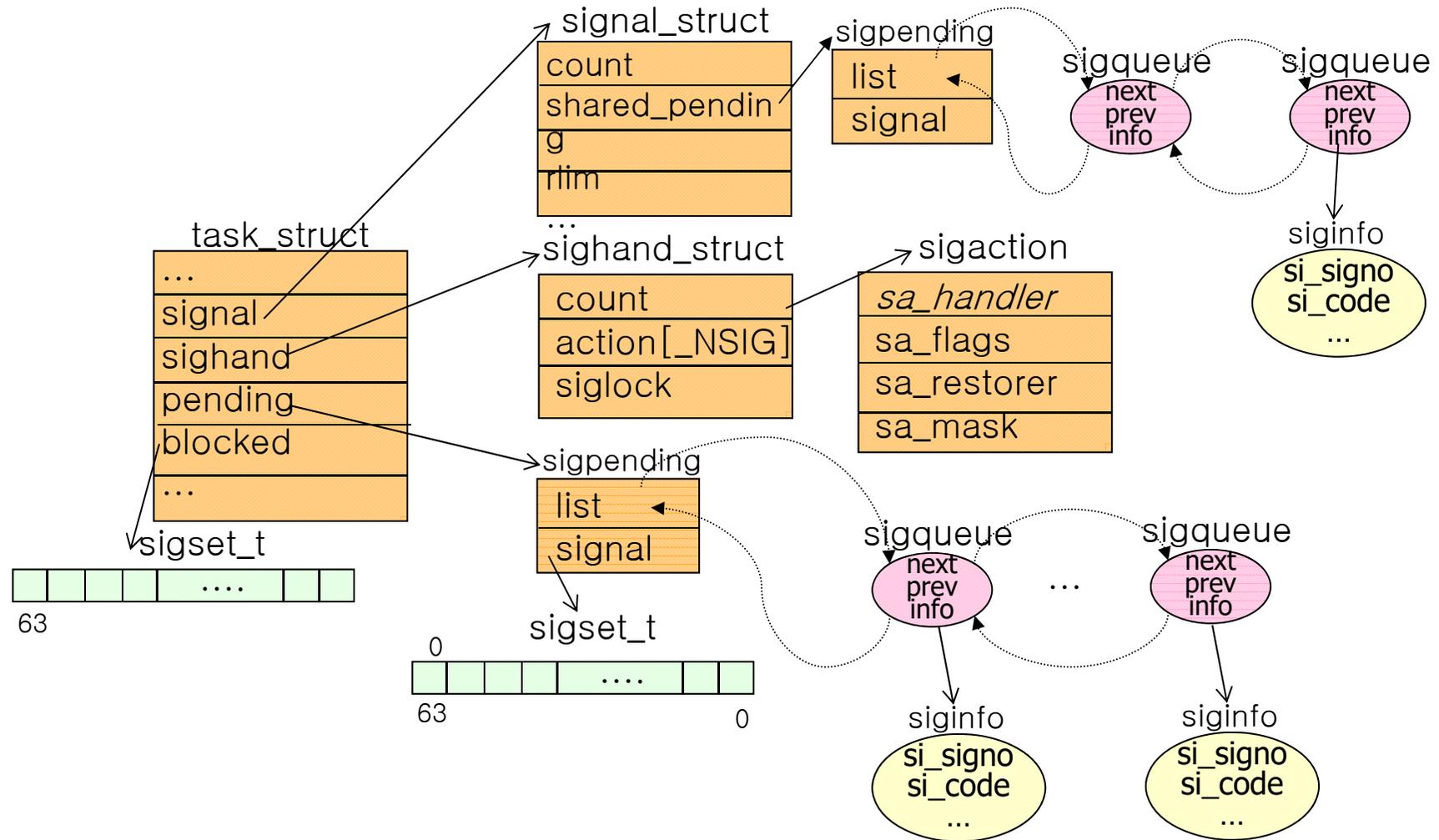
## ■ 태스크 가족 관계



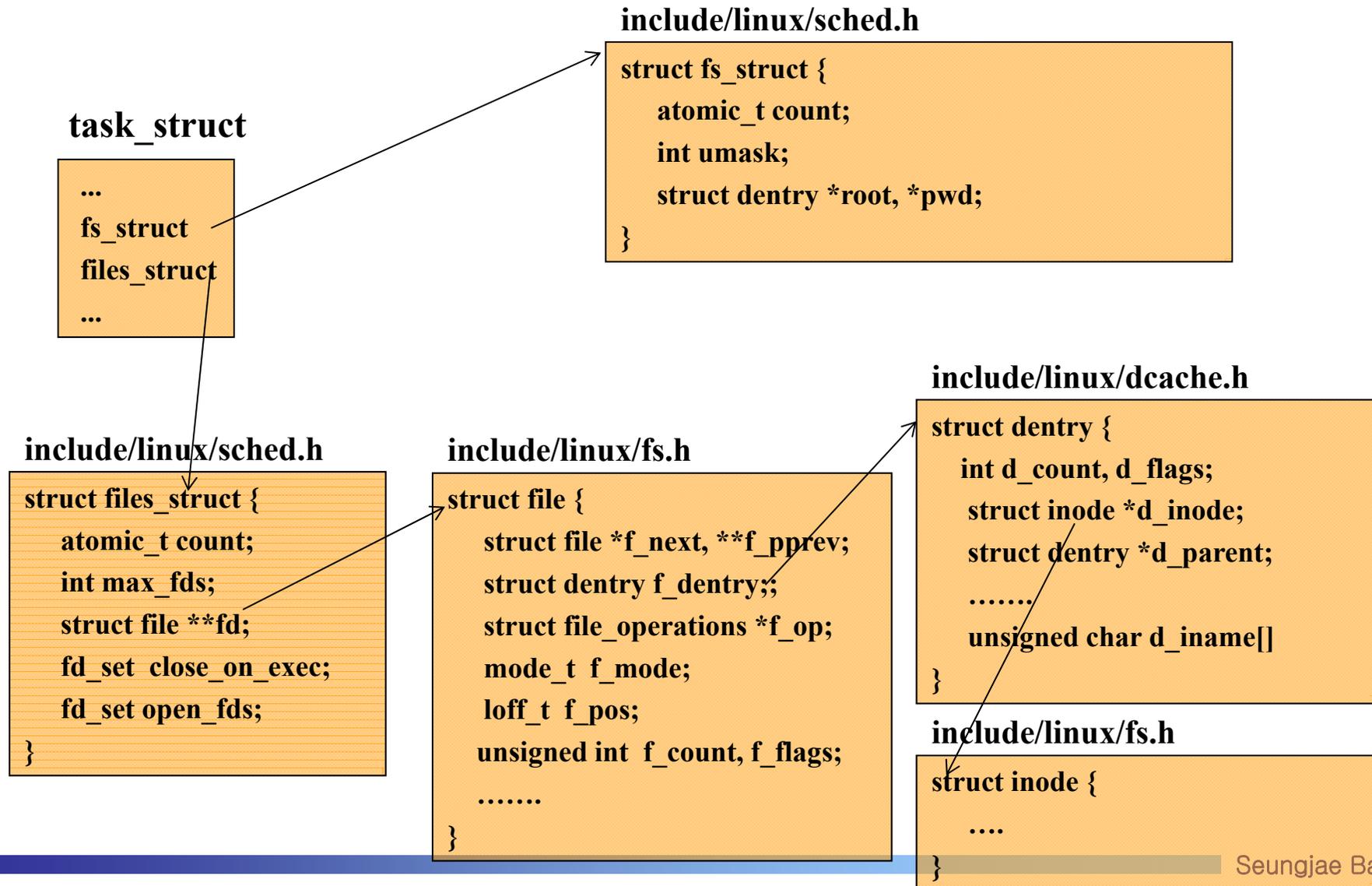
## ■ 시그널 처리 자료 구조



## ■ 시그널 처리 자료 구조



## ■ 파일 관리 자료 구조 : fd, inode

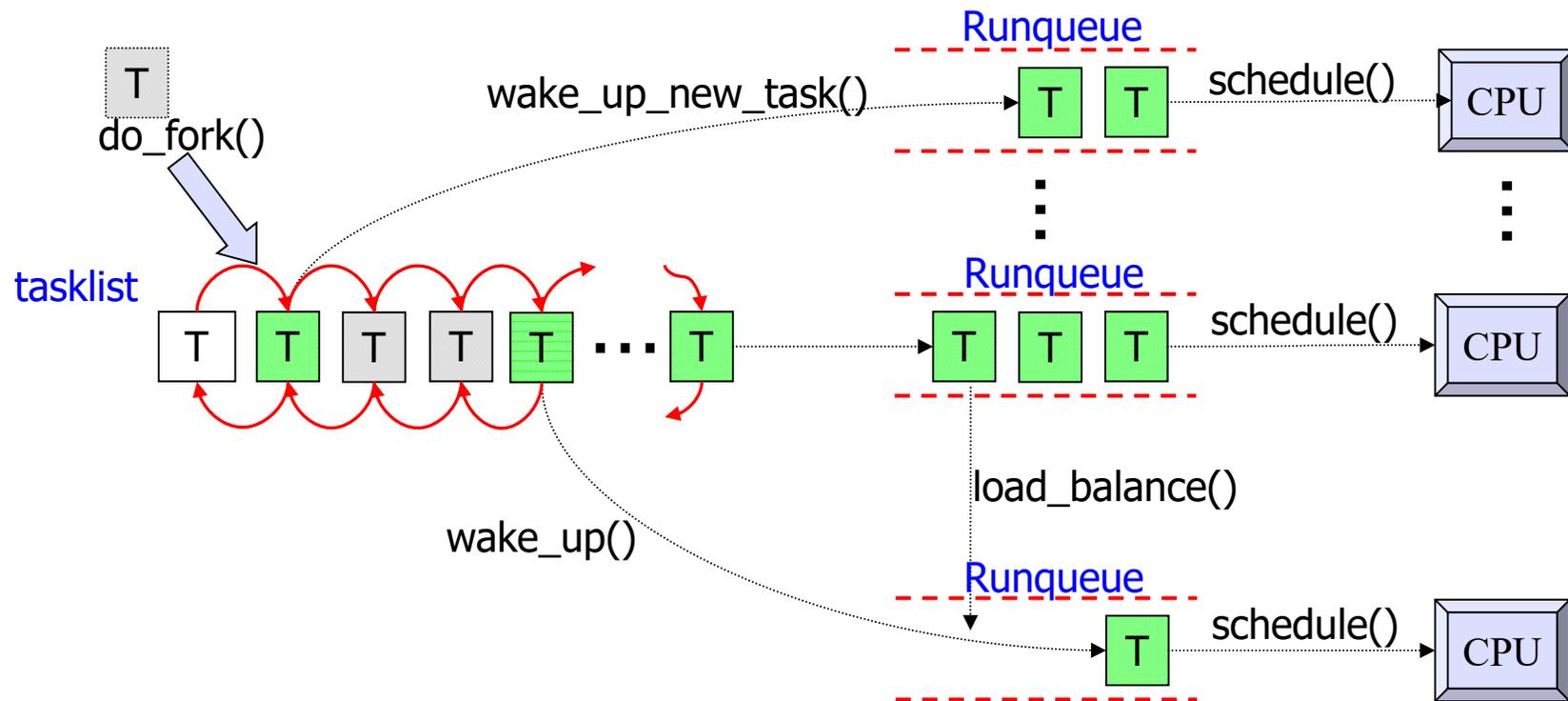


## ■ 스케줄링

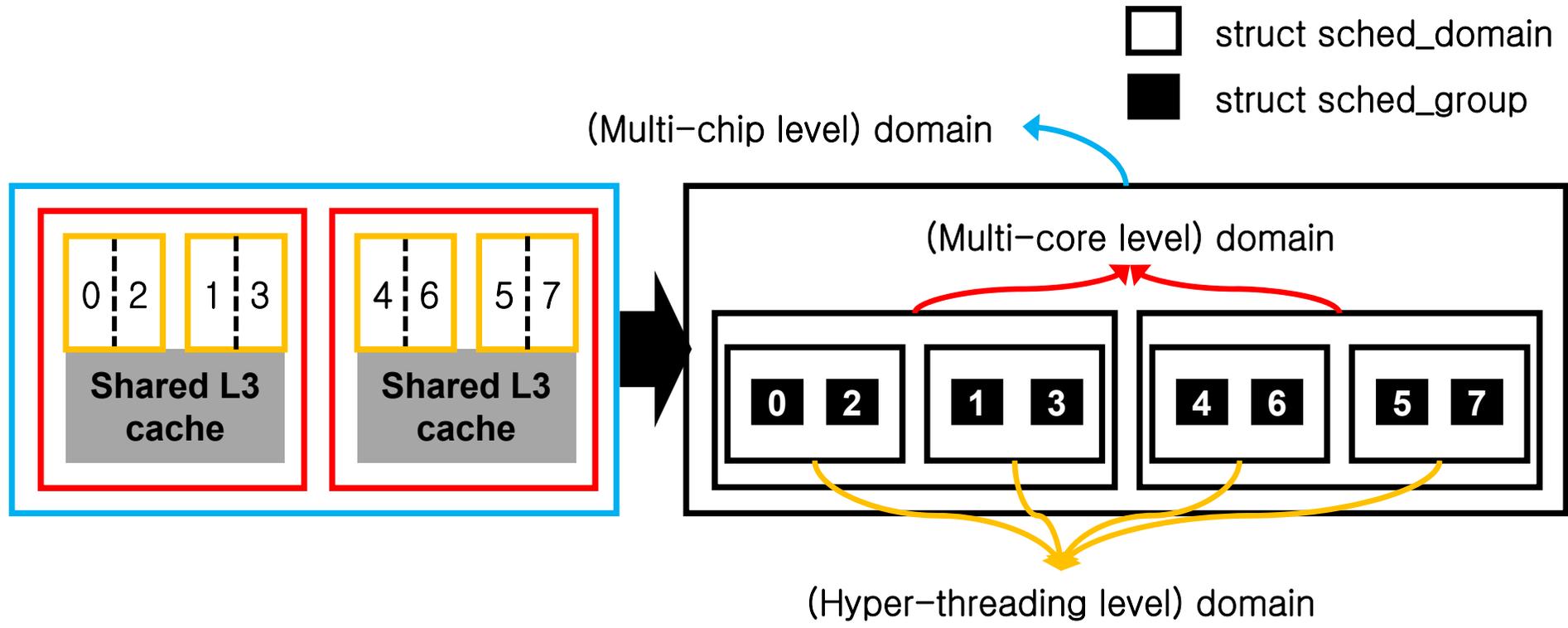
### ✓ Run queue per each CPU

- 일반 task : SCHED\_NORMAL, SCHED\_BATCH, SCHED\_IDLE  
100~139(-20~19)
- 실시간 task : SCHED\_FIFO, SCHED\_RR, SCHED\_DEADLINE, 0~99

### ✓ Load balancing

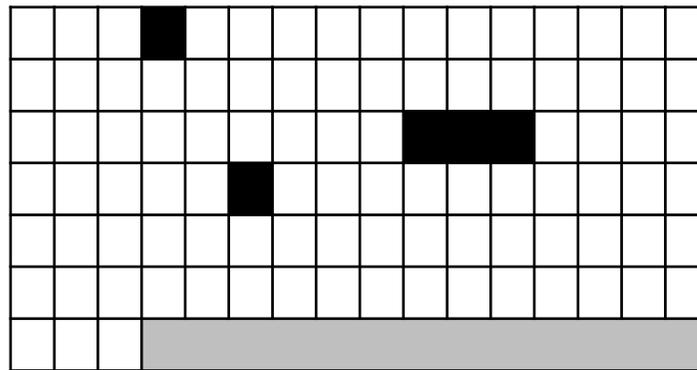


# Scheduling Domain & Group

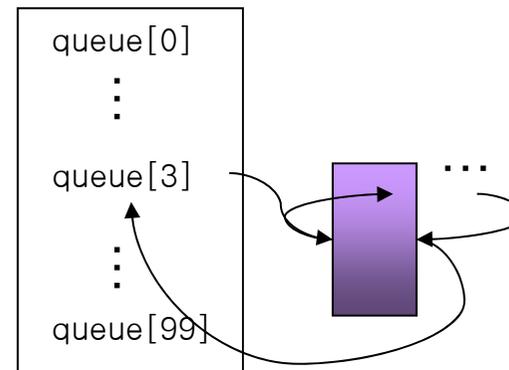


## ■ FIFO & RR

- ✓ O(1) scheduler



rt\_prio\_array.bitmap



rt\_prio\_array.queue

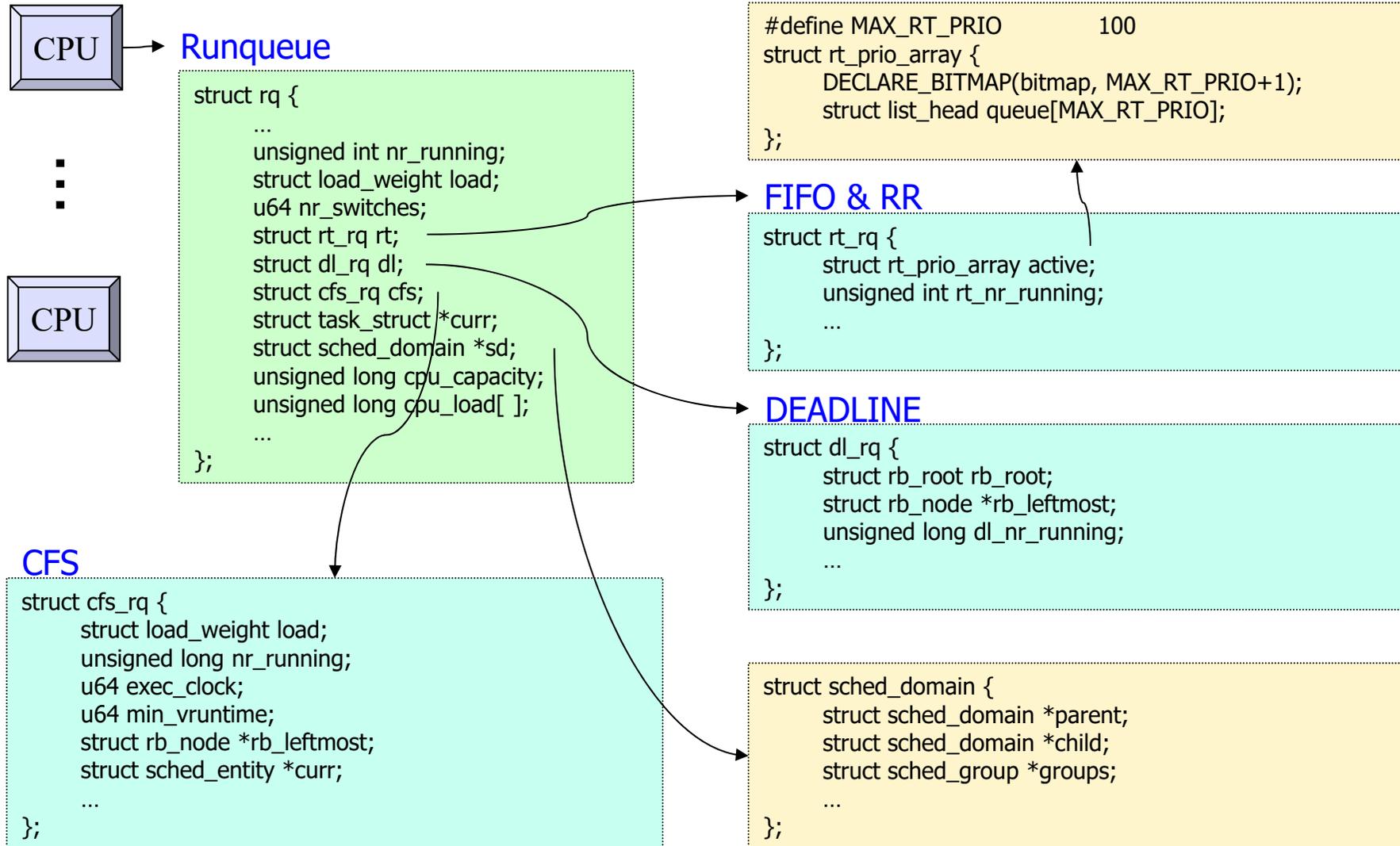
## ■ DEADLINE (a.k.a., EDF)

- ✓ deadline, runtime, period
- ✓ RB-tree (with deadline)
- ✓ Provides deterministic scheduling

$$\mathbf{current}_{time} + \mathbf{runtime} < \mathbf{deadline}$$

$$\mathbf{CPU}_{capability} \geq \sum_{i=0}^n \mathbf{Task}_{(i, runtime)}$$

# Runqueue and Scheduler Data Structure



# 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));

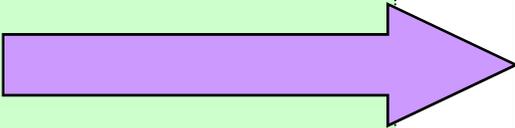
    param.sched_priority = 10;
    sched_setscheduler(0, SCHED_FIFO, &param);
    sched_getparam( 0, &param);

    printf(" \nFIFO set\n");
    printf(" Param.priority = %d\n", param.sched_priority);
    printf(" Sched policy = %d\n", sched_getscheduler(0));

    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));

    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]#
```

①

②

③

- ☞ 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++);
```

## ■ CFS

- ✓ Completely Fair Scheduler
- ✓ RB-tree (with vruntime)
  - Timer tick에 의해 현재 수행 중인 태스크의 vruntime 갱신

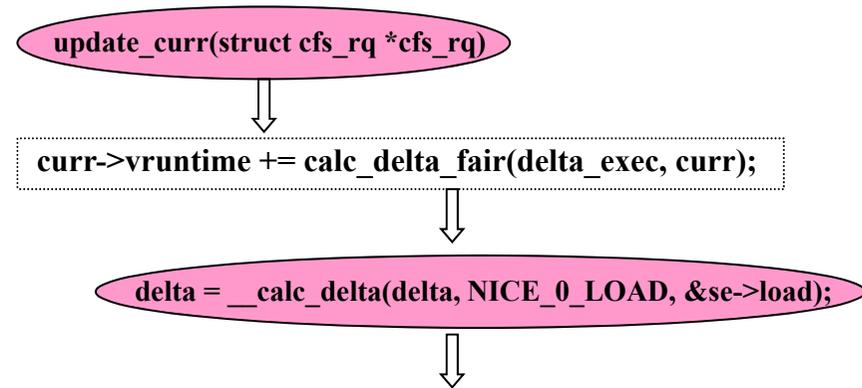
```

struct task_struct {
    ...
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
};

struct sched_entity {
    ...
    struct load_weight load;
    u64 vruntime;
    ...
};

struct load_weight {
    unsigned long weight;
    u32 inv_weight;
};

static const int prio_to_weight[40] = {
/* -20 */    88761,    71755,    56483,    46273,    36291,
/* -15 */    29154,    23254,    18705,    14949,    11916,
/* -10 */    9548,    7620,    6100,    4904,    3906,
/* -5 */     3121,    2501,    1991,    1586,    1277,
/* 0 */     1024,    820,    655,    526,    423,
/* 5 */     335,    272,    215,    172,    137,
/* 10 */     110,    87,    70,    56,    45,
/* 15 */     36,    29,    23,    18,    15,
};
    
```



$$vruntime += physicalruntime \times \frac{weight_0}{weight_{curr}}$$

# Normal Task Scheduling

## ■ CFS

- ✓ Completely Fair Scheduler
- ✓ RB-tree (with vruntime)
  - Timer tick에 의해 현재 수행 중인 태스크의 vruntime 갱신
  - To avoid frequent context switch, there are minimum runtime and

```

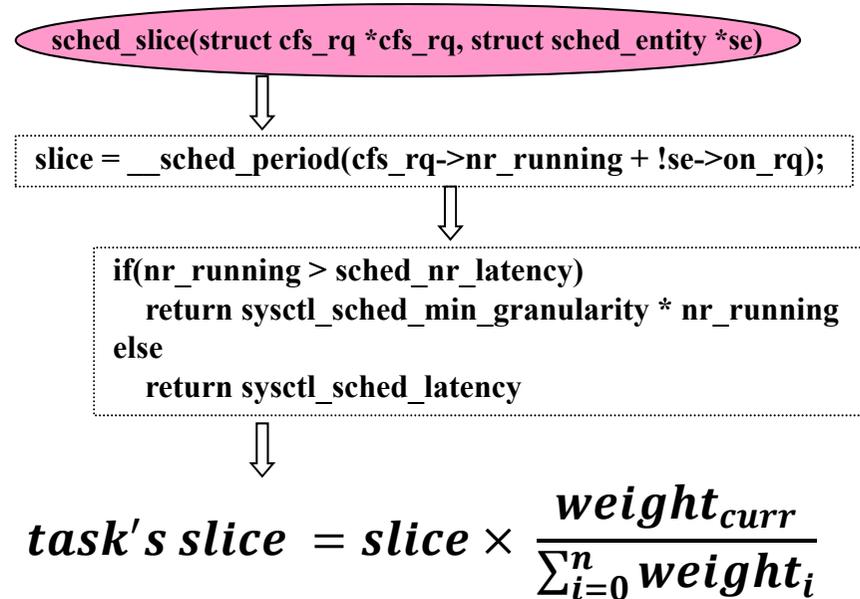
struct task_struct {
    ...
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
};

struct sched_entity {
    ...
    struct load_weight load;
    u64 vruntime;
    ...
};

struct load_weight {
    unsigned long weight;
    u32 inv_weight;
};

static const int prio_to_weight[40] = {
/* -20 */    88761,    71755,    56483,    46273,    36291,
/* -15 */    29154,    23254,    18705,    14949,    11916,
/* -10 */    9548,    7620,    6100,    4904,    3906,
/* -5 */     3121,    2501,    1991,    1586,    1277,
/*  0 */     1024,    820,    655,    526,    423,
/*  5 */      335,    272,    215,    172,    137,
/* 10 */     110,    87,    70,    56,    45,
/* 15 */      36,    29,    23,    18,    15,
};

```



## ■ How

- ✓ Directly call `schedule()`
- ✓ Indirectly, mark `need_resched` flag

## ■ When

Check `check_preempt_tick()` function for more details!

- ✓ Timer tick
- ✓ 수행 중이던 태스크가 자신의 타임 슬라이스를 다 썼거나 대기 상태로 전환
- ✓ 현재 태스크보다 높은 우선순위를 가진 태스크가 깨어난 경우
- ✓ 스케줄링 관련 시스템 콜 호출

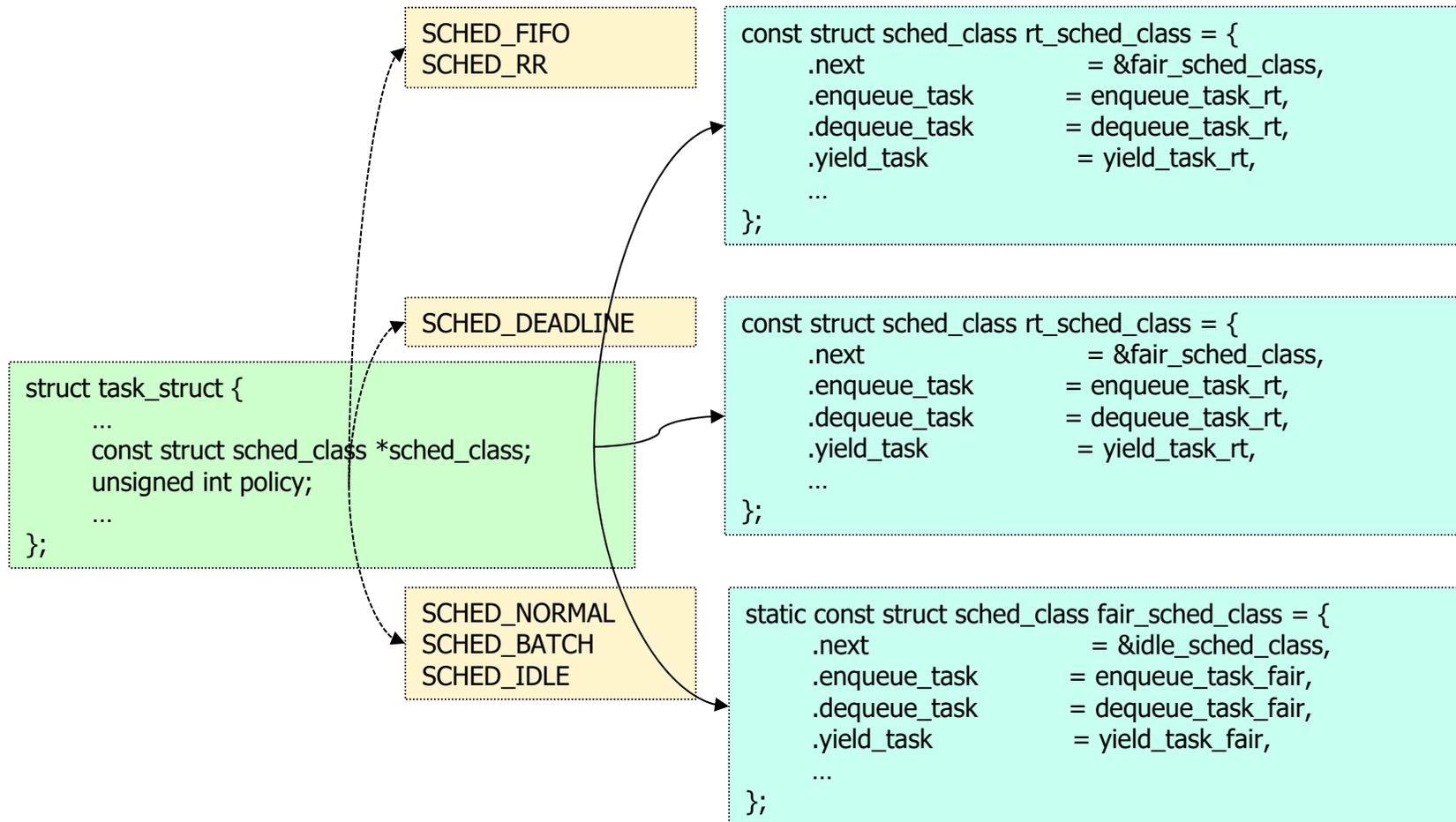
함수 명	목적
<code>set_tsk_need_resched(task)</code>	주어진 프로세스의 <code>need_resched</code> 플래그를 설정
<code>clear_tsk_need_resched(task)</code>	주어진 프로세스의 <code>need_resched</code> 플래그를 해제
<code>need_resched()</code>	<code>need_resched</code> 플래그를 검사. Set이면 <code>true</code> , clear이면 <code>false</code> 를 리턴

- 스케줄링 관련 시스템 콜
  - ✓ nice() (just for backward compatibility)
    - sys\_nice()
    - 절대값 40 넘으면 40으로 맞춤
    - 음수는 관리자 권한 필요 → capable() 함수 호출해 CAP\_SYS\_NICE 특성이 있는지 확인
    - 권한이 있다면 current의 nice 필드에 increment 값 더함
    - 단, -20~19 사이 유지하게 함

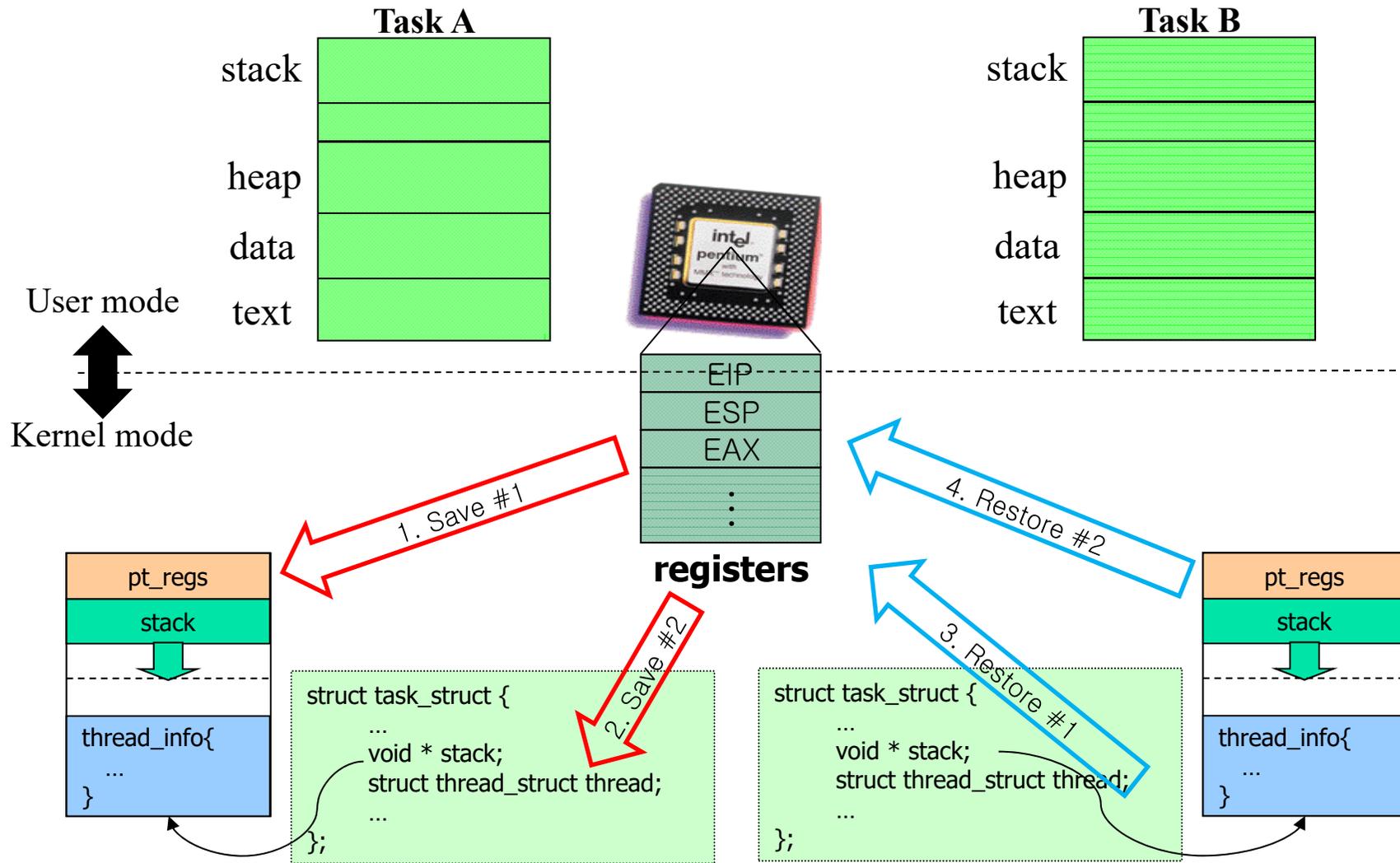
- ✓ `getpriority()`, `setpriority()`
  - 지정한 process 그룹에 있는 모든 process의 기본 우선 순위에 영향을 미침
  - `getpriority()`는 그룹 내 모든 process 중 가장 낮은 nice 필드 값 반환
  - `setpriority()`는 지정 그룹 내 모든 process의 기본 우선 순위 설정
  - `sys_getpriority()`, `sys_setpriority()`
  - 매개변수
    - `which` : 프로세스 그룹을 식별
      - `PRIO_PROCESS`
      - `PRIO_PGRP`
      - `PRIO_USER`
    - `who` : process 선택에 사용하는 pid나, pgrp, uid 필드의 값, 0이면 current
    - `niceval` : 새로운 우선 순위 값

- 실시간 process 관련 시스템 콜
  - ✓ sched\_getscheduler(), sched\_setscheduler()
    - 매개변수 pid로 지정한 process에 현재 적용중인 스케줄링 정책 질의 / 설정
    - Pid를 가지고 find\_process\_by\_pid()호출해, policy값을 반환
  - ✓ sched\_getparam(), sched\_setparam()
    - pid에 해당하는 process의 스케줄링 인자 얻어오거나/설정
    - rt\_priority 필드를 sched\_param 타입 지역변수로 저장 후
    - copy\_to\_user()호출해 복사해줌
  - ✓ sched\_yield()
    - Process 를 보류상태로 만들지 않고 자발적으로 CPU반납
  - ✓ sched\_get\_priority\_min(), sched\_get\_priority\_max()
    - policy매개 변수로 지정한 스케줄링 정책에서 사용할 수 있는 실시간 정적 우선 순위의 최소, 최대 값을 반환
  - ✓ sched\_rr\_get\_interval()
    - pid 매개 변수로 지정한 실시간 P의 RR타임 쿼텀을 사용자 모드 주소 공간에 있는 구조체에 기록 / 가져옴

# Scheduling Policy and Class



# Context Switching



# \_\_switch\_to() in ARM (1/16)

The screenshot displays the TRACE32 debugger interface with three main windows:

- Code Window (B::Data.List):** Shows source code for the `__switch_to()` function. Line 697 is highlighted, showing the call to `switch_to(prev, next, prev);`. The `next` variable is set to `0xC02CC000` and `prev` is set to `0xC02C0000`.
- Memory Dump Window (B::d 0xc02c1f8c):** Shows a memory dump starting at address `SD:0000:C02C1F20`. The dump includes hexadecimal values and their corresponding ASCII characters. A red box highlights the memory region from `SD:0000:C02C1F80` to `SD:0000:C02C1FC0`.
- CPU register window (B::Register):** Shows the state of CPU registers. Register `R13` is highlighted with a red circle and contains the value `C02C1F8C`. Other registers like `R8`, `R9`, `R10`, `R11`, `R12`, `R14`, `SPSR`, and `ACCD` are also visible.

Memory Dump Window

CPU register window

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

The screenshot displays a debugger interface with three main windows:

- Assembly Window (B::Data.List):** Shows assembly code for the `__switch_to` function. The current instruction is `stmdb r13!, {r4-r11, r14}` at address `002480C`. A red arrow points from this instruction to the register window.
- Register Window (B::Register):** Shows the state of registers. The PC register is highlighted with a red circle and contains the value `002480C`. Other registers like R8, R9, R10, and R11 also have values circled in red.
- Memory Window (B::d 0xc02c1f8c):** Shows a memory dump starting at address `0xc02c1f40`. The value `0003AEE0` is highlighted in blue, corresponding to the PC register value.

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

The image shows a debugger interface with two main windows: 'B::Data\_List' and 'B::Register'.

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

```
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 0x0001C630 ; 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
```

**B::Register** (Register Window):

Register	Value	Offset	Address
R0	C02C0000		
R1	C02CC000		
R2	60000013		
R3	2		
R4	C02C0000		
R5	C00140A0		
R6	C02CC000		
R7	0		
SPSR	20000013		
PC	C001C578		
CPSR	20000013		

The CPSR register value 20000013 is circled in red. An arrow points from this value to the CPSR field in the register window below.

**Register Window Details:**

- USR: R8: C0194000, R9: 69052D06, R10: A0014B30, R11: C02C1FAC, R12: 1E, R13: BFFFFFFA0, R14: C66C
- FIQ: R8: 0, R9: 0, R10: 0, R11: 0, R12: 0, R13: 0, R14: 0, SPSR: 10
- SVC: R13: C02C1F68, R14: C0024D8C, SPSR: 20000013
- IRQ: R13: 60000093, R14: C001C280, SPSR: 60000093
- UND: R13: 60000093, R14: C001C4A0, SPSR: 60000093
- ABT: R13: 20000093, R14: C001C3C0, SPSR: 20000093

**Memory Dump (B::d 0xc02c1f8c):**

address	0	4	8	C
SD:0000:C02C1F40	C001C2A0	C0196F80	60000093	60000013
SD:0000:C02C1F50	00000002	C02C0000	C00140A0	C02CC000
SD:0000:C02C1F60	00000000	C0194000	C02C0000	C00140A0
SD:0000:C02C1F70	C02CC000	00000000	C0194000	69052D06
SD:0000:C02C1F80	A0014B30	C02C1FAC	C0024D8C	0003AEED
SD:0000:C02C1F90	000001F4	C02C1FB0	C01D2EA4	C0196BB4
SD:0000:C02C1FA0	C02C1FE0	C02C1FB0	C0024B0C	C0024B80
SD:0000:C02C1FB0	00000000	00000000	0003AEED	C02C0000
SD:0000:C02C1FC0	C0024A84	C02C0340	C02C0330	C02C0000
SD:0000:C02C1FD0	C02C0000	00000000	C02C1FE4	C004CBC8
SD:0000:C02C1FE0	C0024A88	C004CB24	C02E2000	C01ED980

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

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 a register dump with R11 and R12 circled in red. The bottom pane shows a memory dump at address 0xc02c1f68.

**Assembly Code:**

```
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}^
SR:0000:C001C5AC E1A00000 nop
SR:0000:C001C5B0 E1A00000 nop
```

**Register Dump:**

Register	Value	Register	Value	Register	Value
R0	C02C0000	R8	C0194000	SP	20000013
R1	C02CC000	R9	69052D06	+04	C02C0000
R2	60000013	R10	A0014B30	+08	C00140A0
R3	0	R11	C02C1FAC	+0C	C02CC000
R4	C02C0000	R12	20000013	+10	00000000
R5	C00140A0	R13	C02C1F64	+14	C0194000
R6	C02CC000	R14	C0024D8C	+18	69052D06
R7	0	PC	C001C580	+1C	A0014B30
SPSR	20000013	CPSR	20000013	+20	C02C1FAC
ACCO	0	CO	0	+24	C0024D8C
				+28	0003AEEO
				+2C	000001F4
R8	C0194000	R8	0	+30	C02C1FB0
R9	69052D06	R9	0	+34	C01D2EA4
R10	A0014B30	R10	0	+38	C0196BB4
R11	C02C1FAC	R11	0	+3C	C02C1FE0
R12	20000013	R12	0	+40	C02C1FB0
R13	BFFFFFFA	R13	0	+44	C0024B0C
R14	C66C	R14	0	+48	C0024B80
		SPSR	10	+4C	00000000
				+50	00000000
				+54	0003AEEO
R13	C02C1F64	R13	60000093	+58	C02C0000
R14	C0024D8C	R14	C001C280	+5C	C0024A84
SPSR	20000013	SPSR	60000093	+60	C02C0340
				+64	C02C0330
				+68	C02C0000
R13	60000093	R13	20000093	+6C	C02C0000
R14	C001C4A0	R14	C001C3C0	+70	00000000
SPSR	60000093	SPSR	20000093	+74	C02C1FE4
				+78	C004CBC8
				+7C	C0024AA8
				+80	C004CB24
				+84	C02E2000
				+88	C01ED980
				+8C	C01CB2A8
				+90	C001DD98

**Memory Dump:**

address	0	4	8	C	0123456789ABCDEF
SD:0000:C02C1F20	FFFFFFFE	60000093	20000013	C002C554	FFFFFFNN^INN^TcSc
SD:0000:C02C1F30	C02C1F78	F8D00000	04000000	C0024D80	FFFFFF3UU3UU^3W0
SD:0000:C02C1F40	C001C2A0	C0196F80	60000093	60000013	Xf^cHNDfHHHESMSc
SD:0000:C02C1F50	00000002	C02C0000	C00140A0	C02CC000	02H0003000^INN^
SD:0000:C02C1F60	00000000	20000013	C02C0000	C00140A0	NNNN^cA@ScNc^G
SD:0000:C02C1F70	C02CC000	00000000	C0194000	69052D06	XUUUU^00^H0U0^G
SD:0000:C02C1F80	A0014B30	C02C1FAC	C0024D8C	0003AEEO	NS^cHNNNN^lCA^Ei
SD:0000:C02C1F90	000001F4	C02C1FB0	C01D2EA4	C0196BB4	U0^0UUUUU^30K^0
SD:0000:C02C1FA0	C02C1FE0	C02C1FB0	C0024B0C	C0024B80	OK^P^l^c^MScEA^E
SD:0000:C02C1FB0	00000000	00000000	0003AEEO	C02C0000	F^NNB^l^cA^lCBk^lC
SD:0000:C02C1FC0	C0024A84	C02C0340	C02C0330	C02C0000	4HUUF^04^b0k^30

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

The screenshot displays 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`, `mov r9,r9,lsr #0x0D`, `b 0xC001C630 ; ret_to_user`, `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. Register R12 contains the value 0180, which is circled in red. The PC register contains the address C02C1F64, also circled in red. Other registers like R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R13, R14, SPSR, and ACC0 are also listed with their values.
- Memory Dump:** Shows the contents of memory starting at address 0180. The address C02C1F64 is highlighted, and a red arrow points from the register window to this address. The dump shows hexadecimal values and their corresponding ASCII representations.

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

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 list with `C02C0000` circled in red. The bottom pane shows a memory dump with `00000180` circled in red.

**Assembly Code:**

```
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 E5800318 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
```

**Register List:**

N	R0	C02C0000	R8	C0194000	SP>	00000180
Z	R1	C02CC000	R9	69052D06	+04	00000000
C	R2	60000013	R10	A0014B30	+08	20000013
V	R3	2	R11	C02C1FAC	+0C	C02C0000
I	R4	0180	R12	20000013	+10	C00140A0
F	R5	0	R13	C02C1F5C	+14	C02CC000
T	R6	C02CC000	R14	C0024D8C	+18	00000000
	R7	0	PC	C001C588	+1C	C0194000
svc	SPSR	20000013	CPSR	20000013	+20	69052D06
Q	ACCO				+24	A0014B30
					+28	C02C1FAC
					+2C	C0024D8C
	USR:		FIQ:		+30	0003AEED
	R8	C0194000	R8	0	+34	000001F4
	R9	69052D06	R9	0	+38	C02C1FB0
	R10	A0014B30	R10	0	+3C	C01D2EA4
	R11	C02C1FAC	R11	0	+40	C0196BB4
	R12	20000013	R12	0	+44	C02C1FE0
	R13	BFFFFFFD	R13	0	+48	C02C1FB0
	R14	C66C	R14	0	+4C	C0024B0C
			SPSR	10	+50	C0024B80
					+54	00000000
	SVC:		IRQ:		+58	00000000
	R13	C02C1F5C	R13	60000093	+5C	0003AEED
	R14	C0024D8C	R14	C001C280	+60	C02C0000
	SPSR	20000013	SPSR	60000093	+64	C0024A84
					+68	C02C0340
	UND:		ABT:		+6C	C02C0330
	R13	60000093	R13	20000093	+70	C02C0000
	R14	C001C4A0	R14	C001C3C0	+74	C02C0000
	SPSR	60000093	SPSR	20000093	+78	00000000
					+7C	C02C1FE4
					+80	C004CBC8
					+84	C0024AA8
					+88	C004CB24
					+8C	C02E2000
					+90	C01ED980

**Memory Dump:**

address	0	4	8	C	0123456789ABCDEF
SD:0000:C02C1F20	FFFFFFFE	60000093	20000013	C002C554	FFFFFFNNYINN_TESC
SD:0000:C02C1F30	C02C1F78	F8D00000	04000000	C0024D80	EFFFB3UU3UU_5W0
SD:0000:C02C1F40	C001C2A0	C0196F80	60000093	60000013	X1_CNNDFNNH8MSC
SD:0000:C02C1F50	00000002	C02C0000	C00140A0	00000180	8H8033NNYINN
SD:0000:C02C1F60	00000000	20000013	C02C0000	C00140A0	XU0UUU00H00HU
SD:0000:C02C1F70	C02CC000	00000000	C0194000	69052D06	NS_CNNNNN@lCA-Ei
SD:0000:C02C1F80	A0014B30	C02C1FAC	C0024D8C	0003AEED	U0_0UUUUUU0K0
SD:0000:C02C1F90	000001F4	C02C1FB0	C01D2EA4	C0196BB4	OKR0A1_CMSCFAEN
SD:0000:C02C1FA0	C02C1FE0	C02C1FB0	C0024B0C	C0024B80	F5NNB1_CACBKAIC
SD:0000:C02C1FB0	00000000	00000000	0003AEED	C02C0000	4HU0UF_04_B0AK30
SD:0000:C02C1FC0	C0024A84	C02C0340	C02C0330	C02C0000	E1_CB1_CKCKSCKSC

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

The screenshot displays three debugger windows:

- Assembly Window (B::Data\_List):** Shows assembly instructions for the `__switch_to:` function. The instruction `stmdb r13!, {r4-r11, r14}` is highlighted, indicating the saving of registers R4 through R14.
- Registers Window (B::Register):** Lists the state of registers R0 through R14 and SPSR. Register R14 is highlighted with a red circle and contains the value `C02C1F5C`.
- Memory Dump Window (B::d 0xc02c0318):** Shows a memory dump starting at address `0xc02c0310`. The address `C02C1F5C` is highlighted with a red box, and a red arrow points from the R14 register to this address.

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

The image shows a debugger interface with three main windows:

- Assembly View (B::Data\_List):** Displays assembly code for the `__switch_to:` function. The instruction at address `E92D0030` is `stmdb r13!, {r4-r11, r14}`, which is highlighted. The PC register value `C02CDF60` is also visible in the assembly view.
- 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 starting at address `00000000`. The value `C02CDF60` is visible at address `00000004`.

# \_\_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 a register dump with several values circled in red. The bottom pane shows a memory dump at address `0xc02cc31c`.

**Assembly Code (Left Pane):**

```
SR:0000:C001C568 E1A096AD mov r9,r13,lsr #0x00
SR:0000:C001C56C E1A09689 mov r9,r9,lsr #0x00
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 Dump (Right Pane):**

Register	Value	Register	Value	Register	Value
R0	C02C0000	R8	C0194000	SP	00000300
R1	C02CC000	R9	69052D06	+04	00000000
R2	10	R10	A0014B30	+08	20000013
R3	2	R11	C02C1FAC	+0C	C02CC000
R4	0180	R12	20000013	+10	C00140A0
R5	0	R13	C02CDF60	+14	C02E2000
R6	C02CC000	R14	C0024D8C	+18	00000000
R7	0	PC	C001C594	+1C	C0194000
SVC	SPSR 20000013	CPSR	20000013	+20	C02CDFC8
Q	ACCO			+24	C02CC000

**Memory Dump (Bottom Pane):**

Address	0	4	8	C	0123456789ABCDEF
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. The top-left pane displays assembly code for the `__switch_to()` function. The top-right pane shows the register window with R13 highlighted at 0300. The bottom pane shows a memory dump at address 0xc02cdf60.

**Assembly Code (B::Data List):**

```
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 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
```

**Register Window (B::Register):**

N	R0	C02C0000	R8	C0194000	SP>	20000013
Z	R1	C02CC000	R9	69052D06	+04	C02CC000
C	R2	1D	R10	A0014B30	+08	C00140A0
V	R3	2	R11	C02C1FAC	+0C	C02E2000
I	R4	0300	R12	20000013	+10	00000000
F	R5	0	R13	C02CDF68	+14	C0194000
T	R6	C02CC000	R14	C0024D8C	+18	C02CDFC8
SVC	R7	0	PC	C001C598	+1C	C02CC000
Q	SPSR	20000013	CPSR	20000013	+20	C02CDFB0
	ACCO			C0	+24	C0024D8C
					+28	00000001
					+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	BFFFFAD0	R13	0	+44	C0034C90
	R14	C66C	R14	0	+48	C0024880
	SPSR	10	SPSR	10	+4C	00000001
					+50	00000000
					+54	00000000
	R13	C02CDF68	R13	60000093	+58	00010000
	R14	C0024D8C	R14	C001C280	+5C	00000000
	SPSR	20000013	SPSR	60000093	+60	00000000
					+64	C02CC000
					+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	EFEFF7F

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

address	0	4	8	C	0123456789ABCDEF
SD:0000:C02CDF20	C001C2A0	C02CC000	C02E2000	60000013	AC5GNC,CH...C1NN
SD:0000:C02CDF30	00000001	C02CC000	C00140A0	C02E2000	02H0U0,0U...0SUU
SD:0000:C02CDF40	00000000	C0194000	C02CDFC8	C02CC000	NNNN01C0D,CNC,C
SD:0000:C02CDF50	C02CDFB0	C02CDF9C	C01F79E4	C02CDF6C	0F,0CF,04VF0,0F,0
SD:0000:C02CDF60	00000300	00000000	20000013	C02CC000	NNNNNNNNNN,NC,C
SD:0000:C02CDF70	C00140A0	C02E2000	00000000	C0194000	A@SCN...CHNNNN01C
SD:0000:C02CDF80	C02CDFC8	C02CC000	C02CDFB0	C0024D8C	0D,CNC,C0D,C0M0C
SD:0000:C02CDF90	00000001	C0197F80	C02CC330	C02CC340	NNNS7160C,0@S,C
SD:0000:C02CFA0	C02CC000	00000000	C02CDB4	C0034C90	NC,CHNNNED,C3LE
SD:0000:C02CFB0	C0024880	00000001	00000000	00000000	0K0,0UUUU0F,0LW
SD:0000:C02CFC0	00010000	00000000	00000000	C02CC000	NNNNNNNNNNNNNC,C

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

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 `stmdb r13!,{r4-r11,r14}` at address `0xc001c630`. Other instructions include `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`, `mrs spsr,r12`, and `ldmia r13!,{r4-r11,pc}^`.
- B::Register**: Shows the current state of registers. R13 is highlighted with a red circle and contains the value `20000013`. Other registers include R0, R1, R2, R3, R4, R5, R6, R7, R14, SPSR, and ACC0.
- Memory Dump**: Shows the contents of memory starting at address `0xc02cdf20`. The value at `0xc02cdf60` is `00000300`, which is circled in red. The dump shows hexadecimal values and their corresponding ASCII representations.

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

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 a register dump. The bottom pane shows a memory dump at address `0xc02cdf68`.

**Assembly Code:**

```
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
SR:0000:C001C5B4 E1A00000 nop
SR:0000:C001C5B8 E1A00000 nop
SR:0000:C001C5BC E1A00000 nop
```

**Register Dump:**

```
N - R0 C02C0000 R8 C0194000 SP> C02CC000
Z - R1 C02CC000 R9 69052D06 +04 C00140A0
C C R2 1D R10 A0014B30 +08 C02E2000
V - R3 2 R11 C02C1FAC +0C 00000000
I - R4 0300 R12 20000013 +10 C0194000
F - R5 0 R13 C02CDF6C +14 C02CDFC8
T - R6 C02CC000 R14 C0024D8C +18 C02CC000
R7 0 PC C001C5A0 +1C C02CDFB0
svc SPSR 20000013 CPSR 20000013 +20 C0024D8C
Q - ACC0 C0 +24 00000001
+28 C0197F80
USR: FIQ: +2C C02CC330
R8 C0194000 R8 0 +30 C02CC340
R9 69052D06 R9 0 +34 C02CC000
R10 A0014B30 R10 0 +38 00000000
R11 C02C1FAC R11 0 +3C C02CDFB4
R12 20000013 R12 0 +40 C0034C90
R13 BFFFFAD0 R13 0 +44 C0024B80
R14 C66C R14 0 +48 00000001
SPSR 10 +4C 00000000
+50 00000000
SVC: IRQ: +54 00010000
R13 C02CDF6C R13 60000093 +58 00000000
R14 C0024D8C R14 C001C280 +5C 00000000
SPSR 20000013 SPSR 60000093 +60 C02CC000
+64 C0197F88
UND: ABT: +68 C0197F88
R13 60000093 R13 20000093 +6C C0034B74
R14 C001C4A0 R14 C001C3C0 +70 C02E2000
SPSR 60000093 SPSR 20000093 +74 C01ED980
+78 C01CB2A8
+7C C0196BB4
+80 69052D06
+84 A0014B30
+88 C001DD98
+8C EFEFFF7F
+90 FFFFFFFF
```

**Memory Dump:**

```
address 0 4 8 C 0123456789ABCDEF
SD:0000:C02CDF60 00000300 00000000 20000013 C02CC000
SD:0000:C02CDF70 C00140A0 C02E2000 00000000 C0194000
SD:0000:C02CDF80 C02CDFC8 C02CC000 C02CDFB0 C0024D8C
SD:0000:C02CDF90 00000001 C0197F80 C02CC330 C02CC340
SD:0000:C02C DFA0 C02CC000 00000000 C02CDFB4 C0034C90
SD:0000:C02CDFB0 C0024B80 00000001 00000000 00000000
SD:0000:C02CDFC0 00010000 00000000 00000000 C02CC000
SD:0000:C02CDFD0 C0197F88 C0197F88 C0034B74 C02E2000
SD:0000:C02CDFE0 C01ED980 C01CB2A8 C0196BB4 69052D06
SD:0000:C02CDFF0 A0014B30 C001DD98 EFEFFF7F FFFFFFFF
SD:0000:C02CE000 C02B32DC C02B32DC 00000040 C02CE040
```

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

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 state of the processor registers. The bottom pane provides a detailed view of the CP15 register, with the `IBCRO` field highlighted by a red box.

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

```
addr/line source
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 E8B0D030 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
SR:0000:C001C5AC E1A00000
SR:0000:C001C5B0 E1A00000
SR:0000:C001C5B4 E1A00000
SR:0000:C001C5B8 E1A00000
SR:0000:C001C5BC E1A00000
```

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

N	-	R0	C02C0000	R8	C0194000	SP	>	C02CC000
Z	-	R1	C02CC000	R9	69052D06	+04		C00140A0
C	C	R2		1D	R10	A0014B30	+08	C02E2000
V	-	R3		2	R11	C02C1FAC	+0C	00000000
I	-	R4		0300	R12	20000013	+10	C0194000
F	-	R5		0	R13	C02CDF6C	+14	C02CDFC8
T	-	R6	C02CC000		R14	C0024D8C	+18	C02CC000
		R7		0	PC	C001C5A0	+1C	C02CDFB0
SVC		SPSR	20000013		CPSR	20000013	+20	C0024D8C
Q	-	ACCO					+24	00000001
							+28	C0197F80
		USR:			FIQ:		+2C	C02CC330
		R8	C0194000		R8		+30	C02CC340
		R9	69052D06		R9		+34	C02CC000
		R10	A0014B30		R10		+38	00000000
		R11	C02C1FAC		R11		+3C	C02CDFB4

**CP15 Register Details (B::PER):**

```
ID 69052D06 Trademark Intel Arch VSTE
CoreGen XScale CoreRev 4
ProdNum PXA250 ProdRev res
CTYPE 0B1AA1AA CLASS 5 H yes
DSIZE 32k DASS 32 DLENGTH 8
ISIZE 32k IAASS 32 ILENGTH 8
CR 0000397F V 0xffff0000 I enable Z enable R off S on
B little C enable A enable M enable
AuxCR 00000000 MD write back - read allocate P 0 K enable
TTB A0010000 TTBA A0010000
DAC 0000001D D15 no access D14 no access D13 no access D12 no access
D11 no access D10 no access D9 no access D8 no access
D7 no access D6 no access D5 no access D4 no access
D3 no access D2 client D1 manager D0 client
FSR 0000000F X 0 D no Domain 0 Status permission_page
FAR 00016E90
DCLR 00000000 L no locking
PTD 00000000 PTD 00000000
IBCRO C001C5A1 MVA C001C5A0 E enable
IBCR1 00000000 MVA 00000000 E disable
DBR0 00000000
DBR1 00000000
DBC0N 00000000 M Data Breakpoint Address E1 disable E0 disable
CPAR 00000001 CP7 denied CP0 allowed
```

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

The screenshot displays a debugger interface with three main panels:

- Assembly Panel (B::Data, List):** Shows assembly instructions with addresses and sources. The instruction `msr spsr, r12` is highlighted.
- Registers Panel (B::Register):** Lists registers R0 through R11. The value `20000013` is circled in red for registers R2 and CPSR.
- System Properties Panel (B::PER):** Shows system information for CP15. The entry `IBCRO C001C5A5 MVA C001C5A4 E enable` is circled in red.

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

The screenshot displays a debugger interface with three main windows:

- B::Data, List:** Shows assembly instructions. The instruction `ldmia r13!, {r4-r11, pc}` is highlighted, indicating a context switch.
- B::Register:** Shows the state of registers. R13 is highlighted with a red circle, containing the value `C02CDF6C`. Other registers like R8, R9, R10, R11, R12, R14, SPSR, and CPSR are also visible.
- B::d 0xc02cdf68:** Shows a memory dump. The address `C02CDF68` is highlighted with a red box, containing the value `00000300`.

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

The screenshot displays a debugger interface with three main windows:

- Source Window (B::Data.List):** Shows the source code for the `__switch_to()` function. The current execution point is at line 697: `switch_to(prev, next, prev);`. The code includes comments about switching register state and stack, and a `same_process` block that reacquires the kernel lock and checks for rescheduling. A detailed comment explains the core wakeup function, distinguishing between non-exclusive and exclusive wakeups.
- Register Window (B::Register):** Shows the current state of the processor registers. The Program Counter (PC) is at `C002408C`. The Stack Pointer (SPSR) is at `20000013`. The registers R8 through R14 contain various values, including `C0194000`, `C02CDFC8`, `C02CC000`, `20000013`, `C00140A0`, `C02E2000`, `C002408C`, and `C66C`.
- Memory Window (B::d 0xc02cdf6c):** Shows a memory dump starting at address `SD:0000:C02CDF20`. The dump displays hexadecimal values and their corresponding ASCII representations, such as `20000013`, `FFFFFFFF`, `C02CC000`, and `C02CDFB0`.