# Operating System

## Pintos Project #01

핀토스 환경구축


컴퓨터 과학과

201710957

이유진

# PINTOS PROJECT #01

## INDEX

**Intro**



**PINTOS 자료구조 분석**



**PINTOS 함수 분석**



**주요 프로그램 실행경로 분석**

# INTRO

## <RUN PINTOS ALARM-MULTIPLE>

pintos 를 테스트 하기 위해, 터미널에서 cd 명령어를 이용해 ~/pintos/src/threads 위치로 이동한후에, run pintos alarm-multiple 을 입력하여 명령을 수행하였다. 이 테스트를 수행하기 까지의 함수분석, 자료구조분석, 프로그램 실행경로 분석을 하도록 하겠다.

```
pintos@pintos-VirtualBox:~/pintos/src/threads$ pintos run alarm-multiple
Use of literal control characters in variable names is deprecated at /home/pintos/pintos/src/u
11.
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/pintos/pintos/src/utils/pintos li
Constant subroutine SIGVTALRM redefined at /home/pintos/pintos/src/utils/pintos line 927.
warning: can't find squish-pty, so terminal input will fail
bochs -q
========================================================================
                       Bochs x86 Emulator 2.6.2
               Built from SVN snapshot on May 26, 2013
                 Compiled on Feb 19 2019 at 17:39:30
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading...........
Kernel command line: run alarm-multiple
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
```

```
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

# PINTOS 자료구조 분석

### < THREAD 자료구조 >

```
struct thread
{
   tid_t tid;
   enum thread_status status;
   char name[16];
   uint8_t *stack;
   int priority;
   struct list_elem allelem;
   struct list_elem elem;

   #ifdef USERPROG
     uint32_t *pagedir;
   #endif

   unsigned magic;
};
```

- tid_t : 스레드의 id. unique 한 값이다.
- status: 스레드의 실행 상태를 나타낸다. 실행 상태는 running, ready, wait, terminated, new 가 있으며 THREAD_RUNNING, THREAD_DYING 등으로 나타낸다.
- name : char 배열이며, thread 의 이름을 저장한다
- stack : stack 을 가리키는 포인터를 담고있다
- priority : thread 의 우선순위를 담고 있다.
- allelem : 모든 element 를 담고있는 list_elem 구조체형식이다.
- elem : ready queue 등에 push 할 수 있는 element 이다.

  magic : 미리 초기화 해둔 magic 값의 비교를 통해 stack 의 overflow 를 탐지 할 수 있다.

### <KERNEL THREAD FRAME 자료구조>

```
struct kernel_thread_frame
{
   void *eip;                  /* Return address. */
   thread_func *function;      /* Function to call. */
   void *aux;                  /* Auxiliary data for function. */
};
```

- eip : 리턴 주소 값을 저장한다.
- function : thread 가 실행할 함수의 주소를 저장한다
- aux : thread 가 실행할 함수의 파라미터 데이터를 저장한다.

### <SWITCH TRHEAD FRAME 자료구조>

```
struct switch_threads_frame
{
   uint32_t edi;               /*  0: Saved %edi. */
   uint32_t esi;               /*  4: Saved %esi. */
   uint32_t ebp;               /*  8: Saved %ebp. */
   uint32_t ebx;               /* 12: Saved %ebx. */
   void (*eip) (void);         /* 16: Return address. */
   struct thread *cur;         /* 20: switch_threads()'s CUR argument.
   struct thread *next;        /* 24: switch_threads()'s NEXT argument.
};
```

- edi : edi 레지스터 값을 저장한다.
- esi : esi 레지스터 값을 저장한다
- ebp : ebp 레지스터 값을 저장한다.
- eip : eip 레지스터 값, 즉 Return address 값을 저장한다.
- curr, next : switch_thread 함수에서 사용하는 CURR, NEXT argument 값을 저장한다.

문맥 교환에 필요한 reg 값을 저장할 때 사용하는 자료구조 이다.

# PINTOS 함수 분석

### < MAIN 함수 >

```
int main (void)
{
   char **argv;                                                    (1)
   bss_init ();                                                    (2)

   argv = read_command_line ();                                    (3)
   argv = parse_options (argv);                                    (4)

   thread_init ();                                                 (5)
   console_init ();                                                (6)

   printf ("Pintos booting with %'"PRIu32" kB RAM...\n",           (7)
      init_ram_pages * PGSIZE / 1024);

   palloc_init (user_page_limit);                                  (8)
   malloc_init ();                                                 (9)
   paging_init ();                                                 (10)

      …
   intr_init ();                                                   (11)
   timer_init ();                                                  (12)
   kbd_init ();                                                    (13)
   input_init ();                                                  (14)

   thread_start ();                                                (15)
   serial_init_queue ();
   timer_calibrate ();
   …

   printf ("Boot complete.\n");                                    (16)
   run_actions (argv);                                             (17)

   shutdown ();
   thread_exit ();                                                 (18)
}
```

(1) 인자를 접근하기위한 더블 포인터형 변수 argv 선언

(2) bss 영역을 초기화 한다.

(3) commend line 에서 문자열을 읽어 argv 에 저장한다.

(4) option 에 따라 argv 를 나눈다.

(5) thread 를 초기화 한다

(6) console 을 초기화 한다.’

(7) booting 중임을 프린트한다.

(8), (9) ,(10) 메모리를 초기화한다. 각 함수들은 밑에서 다시 설명하겠다.

(11) 전체 인터럽트를 초기화 하는 함수를 실행한다.

(12) timer 을 초기화하는 함수를 실행한다.

(13) 키보드를 초기화하는 함수를 실행한다.

(14) input 을 초기화 하는 함수를 실행한다

(15) thread_start 함수를 통해서 Timer 을 시작 시킨다. 즉 time sharing 을 시작한다. 본격적인 운영체제의 동작이 실행되며, thread scheduling 이 시작되며 time sharing 이 본격적으로 시작되는 시점이다.

(16) 부팅이 완료되었음을 출력한다

(17) 부팅은 끝났으나, 잘 동작하는지 테스트를 하는 함수이다. 우리는 argv 에 alarm-multiple 이라는 문자열을 주었기 때문에 alarm-multiple 을 실행시켜 잘 부팅 및 초기화가 되었는지 확인한다.

이 함수는 return 형식이 int type 이라고 명시 되어 있지만 return 하지 않는다. 커널은 메모리 상에 계속 존재 하고 있어야 한다. 하지만 메인 함수에서 return 하게 되면 함수가 종료되고 메모리상에 머물 수 없기 때문에, 메모리에 계속 머물기 위해서는 return 하지 말아야 한다.

**bss_init()**

```
static void bss_init (void)
{
	extern char _start_bss, _end_bss;
	memset (&_start_bss, 0, &_end_bss - &_start_bss);
}
```

bss 는 초기화 되지 않은 전역변수의 메모리 영역을 말한다. bss 영역을 memset 을 이용하여 _start_bss ( 처음주소) 부터 _end_bss(끝 주소) 까지 초기화 해주었다.

**thread_init ()**

```
void  thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);                                    (1)

  lock_init (&tid_lock);
  list_init (&ready_list);                                                   (2)
  list_init (&all_list);                                                     (2)

  initial_thread = running_thread ();                                        (3)
  init_thread (initial_thread, "main", PRI_DEFAULT);                         (4)
  initial_thread->status = THREAD_RUNNING;                                   (5)
  initial_thread->tid = allocate_tid ();                                     (6)
}
```

initial_thread 는 thread 자료구조를 가지며, 이 자료구조는 아래에서 따로 다시 정리하도록 하겠다.

(1) Assert 는 매크로 함수로, 실행을 하기 전에 조건을 한번 더 확인하는 함수이다. 따라서 interrupt 가 꺼져 있는지 재확인을 한다.

(2) ready 상태의 thread 들의 리스트인 ready_list 와 모든 thread 를 모아 놓은 all_list 를 초기화한다

(3) 최초의 thread 인 initial_thread 에 현재 실행중인 스레드의 TCB 포인터를 리턴시켜준다.

(4) initial_thread 를 main 이라는 함수와 PRI_DEFAULT 라는 우선순위로 초기화 시킨다. PRI_DEFAULT 는 0~63 의 중간 인 32 를 나타낸다.

(5) initial_thread 의 상태를 THREAD_RUNNING 으로 바꾼다

(6) initial_thread 의 tid 를 allocate_tid() 함수를 통해 할당 받아 저장한다.

**palloc_init()**

```
void palloc_init (size_t user_page_limit)
{
   uint8_t *free_start = ptov (1024 * 1024);                              (1)
   uint8_t *free_end = ptov (init_ram_pages * PGSIZE);                    (2)
   size_t free_pages = (free_end - free_start) / PGSIZE;                  (3)
   size_t user_pages = free_pages / 2;                                    (4)
   size_t kernel_pages;

   if (user_pages > user_page_limit)                                      (5)
      user_pages = user_page_limit;

   kernel_pages = free_pages - user_pages;                                (6)

   init_pool (&kernel_pool, free_start, kernel_pages, "kernel pool");     (7)
   init_pool (&user_pool, free_start + kernel_pages * PGSIZE,             (8)
                   user_pages, "user pool");
}
```

page 의 allocator 를 초기화 시키는 함수.

(1) 1024* 1024 의 값을 ptov 함수를 통해서 physical 에서 virtual 으로 변경하여 free_start 에 저장한다.

(2) page 의 개수(init_ram_page)와 page 의 크기(PGSIZE)를 곱하여 전체 영역의 크기를 free_end 에 저장한다.

(3) free_page 에 free_end – free_start (free page 의 크기)를 PGSIZE 로 나누어 page 의 수를 저장한다.

(4) 전체 페이지 개수 중 절반을 user_pages 에 저장한다.

(5) user_pages 의 수가 user_page_limit 보다 더 많으면 user_page_limit 으로 초기화한다.

(6) 커널 페이지 수(kernel_pages)는 free_pages 에서 user_pages 를 제외한 수이므로, free_pages-user_pages 를 저장한다.

(7) kernel_pool 을 시작주소는 free_start, 페이지 수는 kernel_pages, 이름은 kernel pool 로 초기화한다.

(8) user_pool 을 시작주소는 free_start + kernel_pages*PGSIZE, 개수는 user_pages, 이름은 user pool 로 초기화한다.

**malloc_init ()**

```
void malloc_init (void)
{
   size_t block_size;                                                    (1)
   for (block_size = 16; block_size < PGSIZE / 2; blocksize *= 2)        (2)
   {
      struct desc *d = &descs[desc_cnt++];
      ASSERT (desc_cnt <= sizeof descs / sizeof *descs);
      d->block_size = block_size;
      d->blocks_per_arena = (PGSIZE - sizeof (struct arena)) / block_size;
      list_init (&d->free_list);
      lock_init (&d->lock);
   }
}
```

메모리 malloc 을 초기화 시켜주는 함수

(1) block_size 라는 변수를 선언한다

(2) blocksize 를 두배 증가시키면서, PGSIZE/2 보다 작을 때 반복한다.

**intr_init ()**

```
void intr_init (void)
{
   uint64_t idtr_operand;                                                (1)
   int i;
   pic_init ();                                                          (2)
   for (i = 0; i < INTR_CNT; i++)                                        (3)
      idt[i] = make_intr_gate (intr_stubs[i], 0);

   idtr_operand = make_idtr_operand (sizeof idt - 1, idt);               (4)

   asm volatile ("lidt %0" : : "m" (idtr_operand));                      (5)

   for (i = 0; i < INTR_CNT; i++)                                        (6)
      intr_names[i] = "unknown";
   intr_names[0] = "#DE Divide Error";                                   (7)

      …

   intr_names[19] = "#XF SIMD Floating-Point Exception";                 (8)
}
```

(1) idtr_operand 라는 변수를 선언한다

(2) pic_init()을 통해, 인터럽트를 처리할 때 항상 거치게 되는 program interrupt controller 8295 를 초기화를 한다.

(3) 인터럽트의 수(INTR_CNT)만큼, 인터럽트 테이블을 초기화 한다.

(4) idtr_operand 는 make_idtr_operand 의 함수를 통해 값을 지정 받는다

(5) asm volatile 은 assembly 언어를 사용하여 idtr_operand 를 처리한다.

(6) 인터럽트의 수만큼, intr_name 을 unknown 으로 초기화한다

(7)~ (8) 0~19 까지의 각각의 intr 이름을 지정한다.

**timer_init ()**

```
void timer_init (void)
{
   pit_configure_channel (0, 2, TIMER_FREQ);
   intr_register_ext (0x20, timer_interrupt, "8254 Timer");            (1)
}
```

(1) 인터럽트 번호 0x20, timer_interrupt 라는 함수로 “8254 Timer”이라는 이름으로 초기화한다.

**kbd_init()**

```
void kbd_init (void)
{
   intr_register_ext (0x21, keyboard_interrupt, "8042 Keyboard");
}
```

인터럽트 번호 0x21, keyboard_interrupt 라는 함수로 “8042 Keyboard”라는 이름으로 초기화한다

**input_init ()**

```
void input_init (void)
{
   intq_init (&buffer);
}
```

input system 에 필요한 buffer 을 초기화 한다.

**thread_start ()**

```
void thread_start (void)
{
   struct semaphore idle_started;                                    (1)
   sema_init (&idle_started, 0);
   thread_create ("idle", PRI_MIN, idle, &idle_started);             (2)

   intr_enable ();                                                   (3)

   sema_down (&idle_started);
}
```

(1) semaphore 구조체인 idle_started 를 선언한다.

(2) thread_create 라는 함수를 통해서, 스레드를 생성한다. 이 스레드는 idle 라는 이름과 PRI_MIN 이라는 우선순위를 가지고, idle 이라는 함수를 실행하고, idle_started 를 매개변수로 가진다.

(3) 인터럽트 처리를 실행 할 수 있도록 처리한다.

**thread_create**

```
tid_t thread_create (const char *name, int priority, thread_func *function,
void *aux)
{
   struct thread *t;
   struct kernel_thread_frame *kf;
   struct switch_entry_frame *ef;
   struct switch_threads_frame *sf;
   tid_t tid;
   enum intr_level old_level;

   ASSERT (function != NULL);                                          (1)
   t = palloc_get_page (PAL_ZERO);                                     (2)
   if (t == NULL)                                                      (3)
      return TID_ERROR;
   init_thread (t, name, priority);                                    (4)
   tid = t->tid = allocate_tid ();                                     (5)
   old_level = intr_disable ();                                        (6)

   kf = alloc_frame (t, sizeof *kf);                                   (7)
   kf->eip = NULL;
   kf->function = function;
   kf->aux = aux;

   ef = alloc_frame (t, sizeof *ef);                                   (8)
   ef->eip = (void (*) (void)) kernel_thread;

   sf = alloc_frame (t, sizeof *sf);                                   (9)
   sf->eip = switch_entry;
   sf->ebp = 0;

   intr_set_level (old_level);                                         (10)
   thread_unblock (t);                                                 (11)
   return tid;                                                         (12)
}
```

다음장에 계속

(1) thread 를 초기화 할 함수가 NULL 이 아니라는 것을 재확인한다

(2) thread 의 주소 값인 t 에 polloc_get_page 함수를 통하여 값을 할당한다

(3) 할당된 주소 값이 NULL 인 경우 TID_ERROR 를 반환하고 함수를 끝낸다

(4) init_thread 를 통하여, t 를 name 이라는 이름과 priority 라는 우선순위로 초기화한다.

(5) t 의 tid 는 allocate_tid 함수를 통해 할당 받고, tid 에 t 를 저장한다.

(6) old_level 은 intr_disable() 의 반환 값을 저장한다.

(7) kf 에 kernel_thread_frame 을 할당하고 각 값을 초기화 한다.

(8) ef 에 switch_entry_frame 을 할당하고, 각 값을 초기화 한다.

(9) sf 에 switch_thread_frame 을 할당하고, 각 값을 초기화한다.

(10) 인터럽트의 레벨을 old_level 로 바꾼다.

(11) thread_unblock 을 이용하여 t 를 unblock 한다.

(12) tid 를 반환하고 함수를 마친다.

thread_create 를 마친 후의 thread 자료 구조의 값은 아래와 같다.

| aux |
|---|
| Function |
| Return address (NULL) |
| eip |
| Next |
| Curr |
| Return address( = ptr to switch_entry) |
| ebx |
| … |

**kernel_thread**

```
static void kernel_thread (thread_func *function, void *aux)
{
   ASSERT (function != NULL);
   intr_enable ();       /* The scheduler runs with interrupts off. */
   function (aux);       /* Execute the thread function. */
   thread_exit ();       /* If function() returns, kill the thread. */
}
```

(1) 인터럽트를 킨다

(2) thread 의 함수를 실행시킨다

(3) 함수가 종료되면, thread 를 종료시킨다.

**init_thread**

```
init_thread (struct thread *t, const char *name, int priority)
{
   ASSERT (t != NULL);                                    (1)
   ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);   (2)
   ASSERT (name != NULL);                                 (3)
   memset (t, 0, sizeof *t);                              (4)
   t->status = THREAD_BLOCKED;                            (5)
   strlcpy (t->name, name, sizeof t->name);               (6)
   t->stack = (uint8_t *) t + PGSIZE;                     (7)
   t->priority = priority;                                (8)
   t->magic = THREAD_MAGIC;                               (9)
   list_push_back (&all_list, &t->allelem);               (10)
}
```

(1) thread 의 주소 값인 t 가 NULL 이 아님을 확인한다.

(2) priority 가 PRI_MIN 과 PRI_MAX 사이에 존재하는지 확인한다.

(3) name 이 NULL 이 아님을 확인한다.

(4) t 의 memeory 를 set 한다.

(5) t 의 이름을 name 으로 바꾼다

(6) t 의 stack 을 초기화한다

(7) t 의 magic 을 THREAD_MAGIC 으로 초기화한다.

(8) all_list 에 t 의 allelem 을 push_back 한다.ㄴ

**allocate_tid**

```
static tid_t allocate_tid (void)
{
   static tid_t next_tid = 1;                                    (1)
   tid_t tid;
   lock_acquire (&tid_lock);
   tid = next_tid++;                                             (2)
   lock_release (&tid_lock);
   return tid;                                                   (3)
}
```

(1) static 변수인 next_tid 를 1 로 초기화한다

(2) tid 는 next_tid ++ 한 값을 저장한다

(3) tid 를 return 한다.

**alloc_frame**

```
alloc_frame (struct thread *t, size_t size)
{
   ASSERT (is_thread (t));                                       (1)
   ASSERT (size % sizeof (uint32_t) == 0);                       (2)
   t->stack -= size;                                             (3)
   return t->stack;                                              (4)
}
```

(1) t 가 스레드인지 확인한다

(2) size 가 unit32_t 의 배수크기인지 확인한다.

(3) t 의 stack 을 size 만큼 줄인다.

(4) t 의 stack 을 반환한다

**thread_unblock**

```
thread_unblock (struct thread *t)
{
   enum intr_level old_level;
   ASSERT (is_thread (t));                                    (1)
   old_level = intr_disable ();                               (2)
   ASSERT (t->status == THREAD_BLOCKED);                      (3)
   list_push_back (&ready_list, &t->elem);                    (4)
   t->status = THREAD_READY;                                  (5)
   intr_set_level (old_level);                                (6)
}
```

(1) t 가 스레드인지 확인한다

(2) intr_disable()을 실행하고, 반환값을 old_level 에 저장한다.

**thread_exit ()**

```
void thread_exit (void)
{
   ASSERT (!intr_context ());                                 (1)

   #ifdef USERPROG
      process_exit ();
   #endif

   intr_disable ();                                           (2)
   list_remove (&thread_current()->allelem);                  (3)
   thread_current ()->status = THREAD_DYING;                  (4)
   schedule ();                                               (5)
   NOT_REACHED ();                                            (6)

}
```

(1) inter_context()가 아니라는 것을 재확인 한다.

(2) 인터럽트 처리를 멈춘다.

(3) thread 를 thread_current 리스트에서 제거한다.

(4) thread 의 상태를 THREAD_DYING 상태로 바꾼다

(5) thread 에 schedule 함수를 사용한다

(6) 예외가 없다고 가정 했을 때, 이미 schedule 에서 처리가 마무리 되었기 때문에 실행 되지 않는 부분 이다.

**schedule()**

```
static void schedule (void)

{

   struct thread *cur = running_thread ();                    (1)
   struct thread *next = next_thread_to_run ();               (2)
   struct thread *prev = NULL;                                (3)

   ASSERT (intr_get_level () == INTR_OFF);                    (4)
   ASSERT (cur->status != THREAD_RUNNING);                    (5)
   ASSERT (is_thread (next));                                 (6)

   if (cur != next)                                           (7)
      prev = switch_threads (cur, next);
   thread_schedule_tail (prev);                               (8)
}
```

(1) 현재 실행중인 thread 를 cur 에 저장한다.

(2) 다음에 실행 해야 할 thread 를 next 에 저장한다.

(3) prev 를 NULL 로 초기화한다

(4) 인터럽트 처리가 중단 상태인지 확인한다.

(5) cur 의 상태가 running 상태가 아니라는 것을 확인한다. (running 상태라면 switch 할 필요가 없기 때문)

(6) 현재 thread 가 next 임을 확인한다.

(7) cur 와 next 가 동일하지 않다는 것을 확인한후에, thread 의 context 를 switch 한 후 리턴 값을 prev 에 저장한다.

(8) switch 가 끝난 후에 thread_schedule_tail 을 prev 를 이용하여 실행하고 마무리한다.

**thread_schedule_tail**

```
void thread_schedule_tail (struct thread *prev)
{
    struct thread *cur = running_thread ();                          (1)

    ASSERT (intr_get_level () == INTR_OFF);                          (2)


    cur->status = THREAD_RUNNING;                                    (3)

    thread_ticks = 0;                                                (4)

   #ifdef USERPROG
      process_activate ();
   #endif

   if (prev != NULL && prev->status == THREAD_DYING && prev !=   (5)
      initial_thread)
   {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```

(1) cur 에 현재 실행중인 thread 를 저장한다.이미 switch 가 일어난 상태이므로 switch_thread() 에서 처리한 next 스레드 와 동일한다.

(2) 인터럽트 처리가 꺼져 있는지 다시 확인한다.

(3) cur 의 상태를 THREAD_RUNNING 상태로 바꾼다.

(4) timer interrupt 를 이용하기 위해서, thread_ticks 를 초기화한다.

(5) prev 의 상태가 THREAD_DYING, initial_thread 가 아닐 때, prev 와 cur 이 동일하지 않은지 확인한후, prev 의 메모리를 free 한다.

# 프로그램 실행 경로

**<주요 프로그램 실행 경로 그림>**

주요 프로그램은 아래와 같이 실행된다. thread 를 실행하면, 커널 자신을 thread 로 만들고 초기화한다. 커널 자신이 thread 가 된 이후는 부팅이 마무리 된 상태이며, run_action 을 이용하여 테스트 혹은, 커널이 실행 되야 할 때 커널을 실행한다.