Operating System

Pintos Project #01

핀토스 환경구축

컴퓨터 과학과

201710957

이유진

pintos project #01

INDEX

**Intro**

[<run pintos alarm-multiple> 3](#_Toc7679362)

**PINTOS 자료구조 분석**

[< thread 자료구조 > 4](#_Toc7679363)

[<kernel thread frame 자료구조> 5](#_Toc7679364)

[<switch trhead frame 자료구조> 5](#_Toc7679365)

**PINTOS 함수 분석**

[< main 함수 > 6](#_Toc7679366)

[thread\_init () 8](#_Toc7679367)

[palloc\_init() 9](#_Toc7679368)

[malloc\_init () 10](#_Toc7679369)

[intr\_init () 10](#_Toc7679370)

[thread\_start () 11](#_Toc7679371)

[thread\_create 12](#_Toc7679372)

[kernel\_thread 14](#_Toc7679373)

[init\_thread 14](#_Toc7679374)

[allocate\_tid 15](#_Toc7679375)

[alloc\_frame 15](#_Toc7679376)

[thread\_unblock 16](#_Toc7679377)

[thread\_exit () 16](#_Toc7679378)

[schedule() 17](#_Toc7679379)

[thread\_schedule\_tail 18](#_Toc7679380)

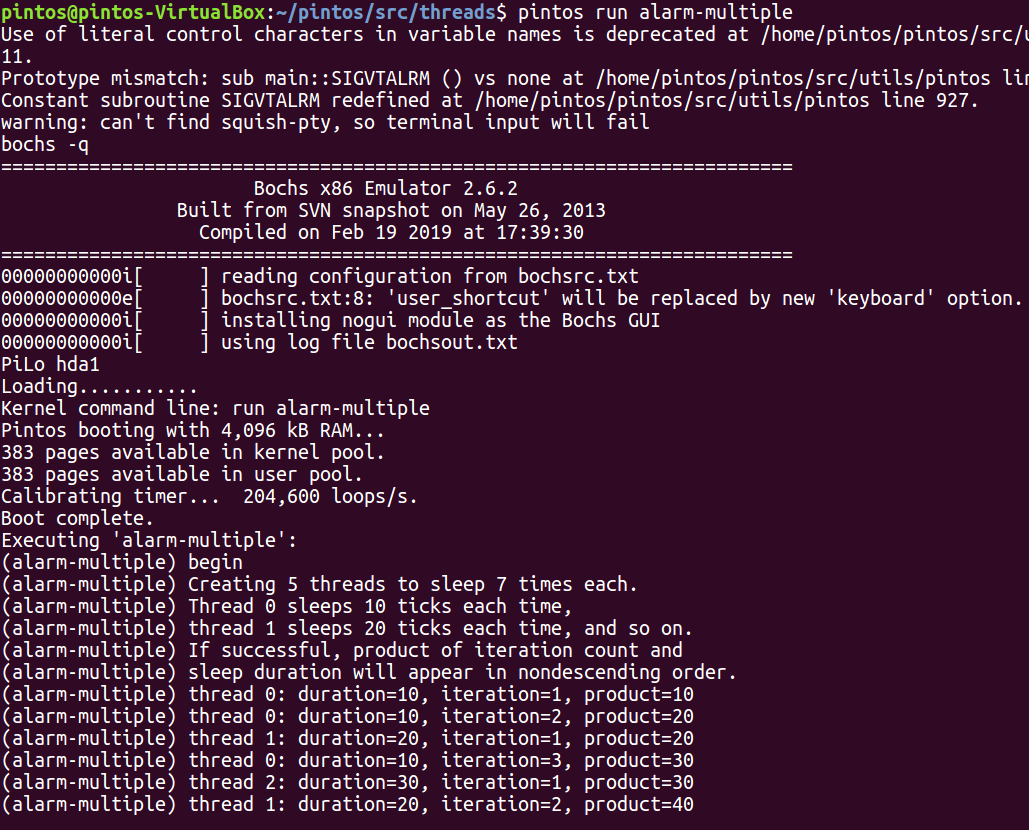
**주요 프로그램 실행경로 분석**

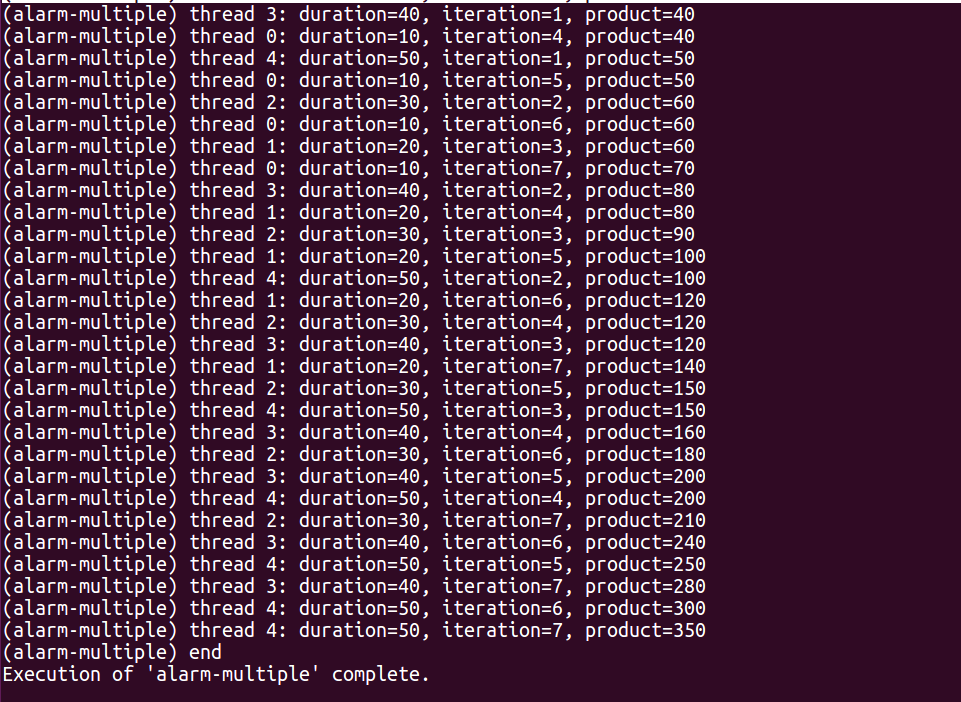
[<주요 프로그램 실행 경로 그림> 19](#_Toc7679381)

intro

# <run pintos alarm-multiple>

pintos를 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 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 overflow를 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. \*/*

uintr32\_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 : eip레지스터 값, 즉 Return address 값을 저장한다.
* curr, next : switch\_thread 함수에서 사용하는 CURR, NEXT argument값을 저장한다.

문맥 교환에 필요한 reg 값을 저장할 때 사용하는 자료구조 이다.

pintos 함수 분석

# < main 함수 >

**int** main (**void**)

{

char \*\*argv; (1)

[bss\_init ();](#_bss_init()" \o "Defined at 147 in src/threads/init.c.) (2)

argv = [read\_command\_line](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\156.html" \l "L194" \o "Defined at 194 in src/threads/init.c.) (); (3)

argv = [parse\_options](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\156.html" \l "L229" \o "Defined at 229 in src/threads/init.c.) (argv); (4)

**[thread\_init](#_thread_init_())** (); (5)

[console\_init](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\105.html" \l "L64" \o "Defined at 64 in src/lib/kernel/console.c.) (); (6)

[printf](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\115.html" \l "L79" \o "Defined at 79 in src/lib/stdio.c.) ("Pintos booting with %'"[PRIu32](file:///C:\Users\pp\Desktop\HTML\S\81.html#L23)" kB RAM...\n", (7)

init\_ram\_pages \* [PGSIZE](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\153.html" \l "L20" \o "Defined at 20 in src/threads/vaddr.h.) / 1024);

**[palloc\_init](#_palloc_init()" \o "Defined at 46 in src/threads/palloc.c.)** (user\_page\_limit); (8)

[**malloc\_init**](#_malloc_init_()) (); (9)

[paging\_init](file:///C:\Users\pp\Desktop\HTML\S\156.html#L158) (); (10)

…

[**intr\_init**](#_intr_init_()) **();** (11)

[**timer\_init**](#_timer_init_()) (); (12)

[**kbd\_init**](#_kbd_init()) (); (13)

[**input\_init**](#_input_init_()) (); (14)

[**thread\_start**](#_thread_start_()) (); (15)

serial\_init\_queue ();

timer\_calibrate ();

…

[printf](file:///C:\Users\pp\Desktop\HTML\S\115.html#L79) ("Boot complete.\n"); (16)

[**run\_actions**](file:///C:\Users\pp\Desktop\HTML\S\156.html#L298)(argv); (17)

[shutdown](file:///C:\Users\pp\Desktop\HTML\S\136.html#L29) ();

[**thread\_exit**](#_thread_exit_()) (); (18)

}

(1) 인자를 접근하기위한 더블 포인터형 변수 argv 선언

(2) bss영역을 초기화 한다.

(3) commend line 에서 문자열을 읽어 argv에 저장한다.

(4) option에 따라 argv를 나눈다.

(5) thread를 초기화 한다

(6) console을 초기화 한다.’

(7) booting 중임을 프린트한다.

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

(11) 전체 인터럽트를 초기화 하는 함수를 실행한다.

(12) timer을 초기화하는 함수를 실행한다.

(13) 키보드를 초기화하는 함수를 실행한다.

(14) input을 초기화 하는 함수를 실행한다

(15) thread\_start 함수를 통해서 Timer을 시작 시킨다. 즉 time sharing을 시작한다. 본격적인 운영체제의 동작이 실행되며, thread scheduling이 시작되며 time sharing이 본격적으로 시작되는 시점이다.

(16) 부팅이 완료되었음을 출력한다

(17) 부팅은 끝났으나, 잘 동작하는지 테스트를 하는 함수이다. 우리는 argv에 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);

}

bss는 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**](#_allocate_tid) (); (6)  
}

initial\_thread는 thread 자료구조를 가지며, 이 자료구조는 아래에서 따로 다시 정리하도록 하겠다.

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

(2) ready 상태의 thread들의 리스트인 ready\_list와 모든 thread를 모아 놓은all\_list 를 초기화한다

(3) 최초의 thread인 initial\_thread 에 현재 실행중인 스레드의 TCB포인터를 리턴시켜준다.

(4) initial\_thread를 main이라는 함수와 PRI\_DEFAULT 라는 우선순위로 초기화 시킨다. PRI\_DEFAULT 는 0~63의 중간 인 32 를 나타낸다.

(5) initial\_thread의 상태를 THREAD\_RUNNING 으로 바꾼다

(6) initial\_thread 의 tid를 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");

}

page의 allocator를 초기화 시키는 함수.

1. 1024\* 1024의 값을 ptov 함수를 통해서 physical 에서 virtual으로 변경하여 free\_start에 저장한다.
2. page의 개수(init\_ram\_page)와 page의 크기(PGSIZE)를 곱하여 전체 영역의 크기를 free\_end에 저장한다.
3. free\_page에 free\_end – free\_start (free page의 크기)를 PGSIZE로 나누어 page의 수를 저장한다.
4. 전체 페이지 개수 중 절반을 user\_pages에 저장한다.
5. user\_pages의 수가 user\_page\_limit 보다 더 많으면 user\_page\_limit으로 초기화한다.
6. 커널 페이지 수(kernel\_pages)는 free\_pages에서 user\_pages를 제외한 수이므로, free\_pages-user\_pages를 저장한다.
7. kernel\_pool을 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 pool로 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 malloc을 초기화 시켜주는 함수

1. block\_size라는 변수를 선언한다
2. blocksize를 두배 증가시키면서, PGSIZE/2 보다 작을 때 반복한다.

## intr\_init ()

**void** [intr\_init](file:///C:\Users\pp\Desktop\HTML\R\552.html) (**void**)

{

[uint64\_t](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\87.html" \l "L29" \o "Defined at 29 in src/lib/stdint.h.) idtr\_operand; (1)

**int** i;

[pic\_init](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\168.html" \l "L238" \o "Defined at 238 in src/threads/interrupt.c.) (); (2)

**for** (i = 0; i < [INTR\_CNT](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\168.html" \l "L22" \o "Defined at 22 in src/threads/interrupt.c.); i++) (3)

idt[i] = [make\_intr\_gate](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\168.html" \l "L317" \o "Defined at 317 in src/threads/interrupt.c.) (intr\_stubs[i], 0);

idtr\_operand = [make\_idtr\_operand](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\168.html" \l "L333" \o "Defined at 333 in src/threads/interrupt.c.) (**sizeof** idt - 1, idt); (4)

**asm** **volatile** ("lidt %0" : : "m" (idtr\_operand)); (5)

**for** (i = 0; i < [INTR\_CNT](file:///C:\\Users\\pp\\Desktop\\HTML\\S\\168.html" \l "L22" \o "Defined at 22 in src/threads/interrupt.c.); 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\_operand는 make\_idtr\_operand의 함수를 통해 값을 지정 받는다
5. asm volatile 은 assembly언어를 사용하여 idtr\_operand를 처리한다.
6. 인터럽트의 수만큼, intr\_name을 unknown으로 초기화한다

(7)~ (8) 0~19까지의 각각의intr이름을 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\_interrupt라는 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 필요한buffer을 초기화 한다.

## thread\_start ()

**void** thread\_start (**void**)

{

**struct** semaphore idle\_started; (1)

sema\_init (&idle\_started, 0);

[**thread\_create**](#_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\_create라는 함수를 통해서, 스레드를 생성한다. 이 스레드는 idle라는 이름과 PRI\_MIN 이라는 우선순위를 가지고, idle 이라는 함수를 실행하고, idle\_started를 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](#_init_thread) (t, name, priority); (4)

tid = t->tid = [allocate\_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. thread를 초기화 할 함수가 NULL 이 아니라는 것을 재확인한다
2. thread의 주소 값인 t 에 polloc\_get\_page함수를 통하여 값을 할당한다
3. 할당된 주소 값이 NULL인 경우 TID\_ERROR를 반환하고 함수를 끝낸다
4. init\_thread를 통하여, t를 name이라는 이름과 priority라는 우선순위로 초기화한다.
5. t의 tid는 allocate\_tid함수를 통해 할당 받고, tid에 t를 저장한다.
6. old\_level은 intr\_disable() 의 반환 값을 저장한다.
7. kf에 kernel\_thread\_frame을 할당하고 각 값을 초기화 한다.
8. ef에 switch\_entry\_frame을 할당하고, 각 값을 초기화 한다.
9. sf에 switch\_thread\_frame 을 할당하고, 각 값을 초기화한다.
10. 인터럽트의 레벨을 old\_level로 바꾼다.
11. thread\_unblock을 이용하여 t를 unblock한다.
12. tid를 반환하고 함수를 마친다.

thread\_create를 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](#_thread_exit_())** (); */\* If function() returns, kill the thread. \*/*

}

1. 인터럽트를 킨다
2. thread의 함수를 실행시킨다
3. 함수가 종료되면, thread를 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\_MAX사이에 존재하는지 확인한다.
3. name 이 NULL 이 아님을 확인한다.
4. t의 memeory를 set한다.
5. t의 이름을 name으로 바꾼다
6. t의 stack 을 초기화한다
7. t의 magic 을 THREAD\_MAGIC 으로 초기화한다.
8. all\_list 에 t의 allelem을 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 1로 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. t가 스레드인지 확인한다
2. size 가 unit32\_t의 배수크기인지 확인한다.
3. t의 stack을 size만큼 줄인다.
4. t의 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. t가 스레드인지 확인한다
2. intr\_disable()을 실행하고, 반환값을 old\_level에 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. thread의 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**](#_thread_schedule_tail) (prev); (8)

}

1. 현재 실행중인 thread 를 cur에 저장한다.
2. 다음에 실행 해야 할 thread를 next 에 저장한다.
3. prev 를 NULL 로 초기화한다
4. 인터럽트 처리가 중단 상태인지 확인한다.
5. cur의 상태가 running 상태가 아니라는 것을 확인한다. (running 상태라면 switch 할 필요가 없기 때문)
6. 현재 thread 가 next임을 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\_tail을 prev를 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. cur에 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\_ticks를 초기화한다.
5. prev의 상태가 THREAD\_DYING, initial\_thread 가 아닐 때, prev와 cur이 동일 하지 않은지 확인한후, prev 의 메모리를 free 한다.

프로그램 실행 경로

# <주요 프로그램 실행 경로 그림>

주요 프로그램은 아래와 같이 실행된다. thread를 실행하면, 커널 자신을 thread로 만들고 초기화한다. 커널 자신이 thread가 된 이후는 부팅이 마무리 된 상태이며, run\_action 을 이용하여 테스트 혹은, 커널이 실행 되야 할 때 커널을 실행한다.

<MAIN>

bss\_init()

thread\_init()

console\_init()

palloc\_init()

malloc\_init()

paging\_init()

intr\_init()

timer\_init()

kbd\_init()

thread\_start()

run\_action()

<thread\_create>

init\_thread()

\_thre

<init\_thread>