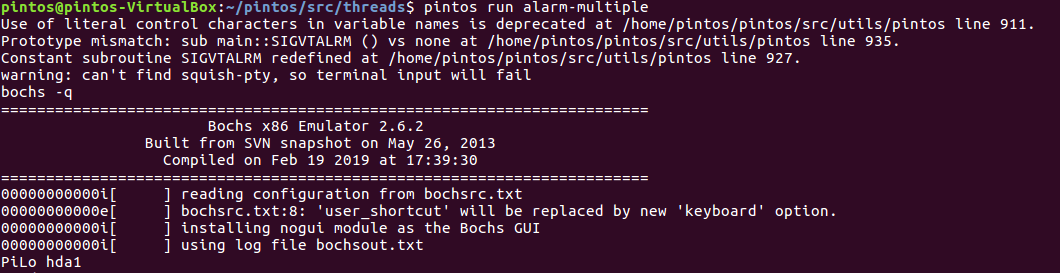
“pintos –v -- run alarm-multiple” 명령을 실행시키는 화면

command line으로 alarm-multiple을 실행시킨다.

**int** main (**void**) NO\_RETURN;

*/\* Pintos main program. \*/*

**int**

main (**void**)

{

**char** \*\*argv;

*/\* Clear BSS. \*/*

bss\_init ();

*/\* Break command line into arguments and parse options. \*/*

argv = read\_command\_line (); //1

argv = parse\_options (argv);

*/\* Initialize ourselves as a thread so we can use locks,*

*then enable console locking. \*/*

thread\_init (); // 2

console\_init ();

*/\* Greet user. \*/*

printf ("Pintos booting with %'"PRIu32" kB RAM...\n",

init\_ram\_pages \* PGSIZE / 1024);

**main 함수**

*/\* Initialize memory system. \*/*

palloc\_init (user\_page\_limit); // 3

malloc\_init (); // 4

paging\_init (); // 5

*/\* Segmentation. \*/*

#ifdef USERPROG

tss\_init ();

gdt\_init ();

#endif

*/\* Initialize interrupt handlers. \*/*

intr\_init (); // 6

timer\_init (); // 7

kbd\_init (); // 8

input\_init (); // 9

#ifdef USERPROG

exception\_init ();

syscall\_init ();

#endif

*/\* Start thread scheduler and enable interrupts. \*/*

thread\_start (); // 10

serial\_init\_queue (); //11

timer\_calibrate (); //12

#ifdef FILESYS

*/\* Initialize file system. \*/*

ide\_init ();

locate\_block\_devices ();

filesys\_init (format\_filesys);

#endif

printf ("Boot complete.\n"); // 13

*/\* Run actions specified on kernel command line. \*/*

run\_actions (argv); // 14

*/\* Finish up. \*/*

shutdown ();

thread\_exit ();

}

pintos –v -- run alarm-multiple 명령을 실행시키면 핀토스가 부팅을 시작하고 test를 수행하게 된다. 핀토스가 부팅을 하기 위해서는 위의 메인 함수가 실행되고, 메인함수 중 13의 코드 run\_actions (argv); 에서 alarm-multiple test를 수행하게 된다. 테스트가 수행되기까지 메인함수 실행의 흐름을 살펴보면 다음과 같다.

먼저 **bss\_init()**함수에서 bss영역을 초기화 한다. 그 다음으로는 **read\_command\_line()**을 호출하여 커널 command line을 인수로 구분하고, **parse\_options()**를 호출하여 command\_line의 시작부분에 있는 옵션을 읽는다. 그 다음에는 **threa\_init()**을 호출하여 커널을 하나의 스레드로 초기화하고 **console\_init()**을 통해 콘솔을 초기화 한 뒤에 콘솔에 시작 메시지를 출력한다. 그 다음에는 **palloc\_init(), malloc\_init(), paging\_init()**을 호출하여 커널의 메모리 시스템을 초기화 한다. 그리고 **intr\_init(), timer\_init(), kbd\_init(), input\_init()**을 차례로 호출하여 인터럽트 처리를 준비하고, 타이머 인터럽트와 키보드 인터럽트를 설정하여 전체적인 인터럽트 시스템을 초기화한다. 앞의 과정을 통해 인터럽트가 설정되었으므로 이제 **thread\_start()**를 통해 스케쥴링을 시작할 수 있게 되고, 인터럽트 기반의 serial port 입출력이 가능하게 되었으므로 **serial\_init\_queue()**를 통해 해당 모드로 전환한다. 타이머의 정확도를 위해 **timer\_calibrate()**를 통해서 타이머를 보정하면 부팅이 완료되고, 부팅이 완료되었다는 메시지를 출력한다. **run\_actions()**함수가 실행되면서 test가 수행되게 되는데, 메인함수내에서 호출되는 함수들의 자세한 내용을 살펴보면 다음과 같다.

**1. [function] read\_command\_line()**

kernel command line를 읽고 단어로 분해해서 argv 배열에 저장한다.

**static** **char** \*\*

read\_command\_line (**void**)

{

**static** **char** \*argv[LOADER\_ARGS\_LEN / 2 + 1];

**char** \*p, \*end;

**int** argc;

**int** i;

argc = \*(uint32\_t \*) ptov (LOADER\_ARG\_CNT);

p = ptov (LOADER\_ARGS);

end = p + LOADER\_ARGS\_LEN;

**for** (i = 0; i < argc; i++)

{

**if** (p >= end)

PANIC ("command line arguments overflow");

argv[i] = p;

p += strnlen (p, end - p) + 1;

}

argv[argc] = NULL;

*/\* Print kernel command line. \*/*

printf ("Kernel command line:");

**for** (i = 0; i < argc; i++) // (1)

**if** (strchr (argv[i], ' ') == NULL)

printf (" %s", argv[i]);

**else**

printf (" '%s'", argv[i]);

printf ("\n");

**return** argv;

}

(1) kernel command line을 출력한다.

**2. [function] thread\_init ()**

커널 자체도 하나의 스레드이다. thread\_init() 함수는 자신을 호출한 프로그램을 스레드화 하는 함수이다. 따라서 커널에서 이 함수를 호출하게 되면 커널을 하나의 스레드로 만든다.

void

thread\_init (void)

{

ASSERT (intr\_get\_level () == INTR\_OFF); // (1)

lock\_init (&tid\_lock);

list\_init (&ready\_list); // (2)

list\_init (&all\_list);

/\* Set up a thread structure for the running thread. \*/

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)

}

1. ASSERT 함수는 매개변수로 전달되는 조건이 만족되어 있는지 확인하는 함수이다. ASSERT 함수에 조건 “intr\_get\_level() == INTR\_OFF” 를 전달하여 인터럽트가 꺼져 있는지를 확인한다.
2. ready\_list(ready queue)를 초기화 시킨다.
3. initial\_thread에 running\_thread() 함수의 반환값을 저장한다. 스택의 탑을 가리키는 esp의 위치보다 작으면서 4096의 배수가 되는 가장 작은 값이 해당 스레드의 TCB의 시작 위치가 된다. running\_thread 함수는 이러한 방식을 이용하여 실행중인 스레드의 TCB의 시작 위치를 반환한다. 따라서 initial\_thread는 커널스레드의 TCB를 가리키는 포인터가 된다.
4. init\_thread 함수는 스레드를 초기화 시키는 함수이다.([function]1-1) 매개변수로는 초기화시킬 구조체(스레드의 TCB), 그리고 그 구조체에 부여할 이름과 우선순위를 전달한다. init\_thread(initial\_thread, “main”, PRI\_DEFAULT)는 initial\_thread가 가리키는 TCB의 이름을 main이라고 하고 우선순위로 PRI\_DEFAULT(31)을 부여할 것이라는 것이다. 현재 initial\_thread는 커널 스레드의 TCB를 가리키고 있으므로 해당 코드는 커널 스레드를 초기화시킨다.
5. initial\_thread(커널스레드)의 상태를 running으로 설정한다.
6. initial\_thread(커널스레드)의 tid에 새로운 tid 값을 부여한다. allocate\_tid()함수는 가장 최근에 만들어진 스레드의 tid값에 1을 더한 값을 반환한다.

**2-1 . [function] init\_thread**

매개변수로 전달받은 스레드, 이름, 우선순위가 올바른지 ASSERT함수를 이용하여 확인한 후

TCB의 각 영역을 초기화 한다..

**static** **void**

init\_thread (**struct** thread \*t, **const** **char** \*name, **int** priority)

{

ASSERT (t != NULL);

ASSERT (PRI\_MIN <= priority && priority <= PRI\_MAX);

ASSERT (name != NULL);

memset (t, 0, **sizeof** \*t);

t->status = THREAD\_BLOCKED; // (1)

strlcpy (t->name, name, **sizeof** t->name); // (2)

t->stack = (uint8\_t \*) t + PGSIZE; // (3)

t->priority = priority; // (4)

t->magic = THREAD\_MAGIC; // (5)

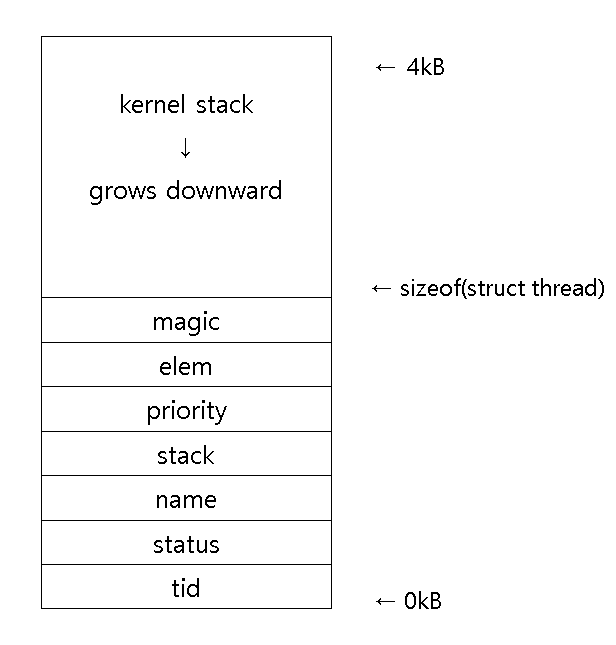
list\_push\_back (&all\_list, &t->allelem);

}

1. thread의 상태를 BLOCKED(waiting)으로 설정한다.
2. TCB의 name 영역에 이름을 저장한다.
3. stack 영역을 초기화한다. 이 영역은 thread가 사용중인 stack의 top을 가리킨다.
4. priority에 전달받은 우선순위를 저장한다.
5. magic영역을 THREAD\_MAGIC으로 설정한다. stack overflow가 발생하면 이 값이 다른 값으로 바뀌게 된다.

TCB의 자료구조는 다음과 같다.

**[struct] thread**



**struct** thread

{

tid\_t tid; //(1)

**enum** thread\_status status; //(2)

**char** name[16]; //(3)

uint8\_t \*stack; //(4)

**int** priority; //(5)

**struct** list\_elem allelem;

**struct** list\_elem elem; //(6)

#ifdef USERPROG

uint32\_t \*pagedir;

#endif

**unsigned** magic; //(7)

};

1. thread를 구분하는 tid를 저장하는 영역이다.
2. thread의 상태를 저장하는 영역이다. THREAD\_RUNNING, THREAD\_BLOCKED, THREAD\_READY, THREAD\_DYING의 네가지 중 하나의 상태가 저장된다
3. thread의 이름이 저장되는 영역이다. 디버깅 목적으로 사용된다.
4. thread가 사용하는 stack을 가리키는 stack pointer가 저장되는 영역이다.
5. thread의 우선순위를 저장하는 영역이다.
6. thread를 queue에 연결할 때 사용하는 영역이다.
7. stack과 가장 인접하여 stack의 overflow를 감지하는 역할을 한다.

**3. [function] palloc\_init()**

palloc\_init() 함수는 동적메모리 할당영역을 초기화 한다. 커널이 동작하면서 메모리를 할당받아야 할 필요가 있을 경우, palloc\_init()을 통해 초기화된 메모리 영역에서 할당받게 된다.

**void**

palloc\_init (size\_t user\_page\_limit)

{

*/\* Free memory starts at 1 MB and runs to the end of RAM. \*/*

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)

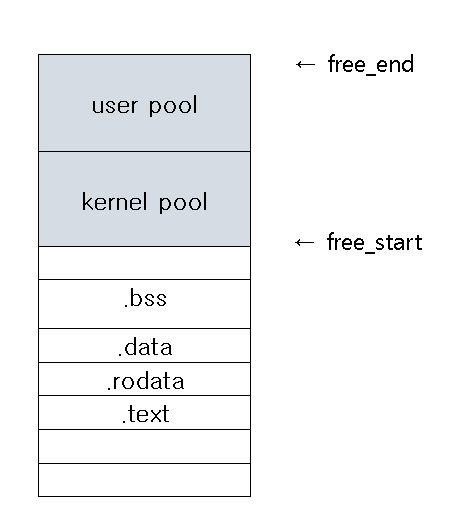
init\_pool (&user\_pool, free\_start + kernel\_pages \* PGSIZE, //(8)

user\_pages, "user pool");

}

1. ptov() 함수는 매개변수로 전달한 물리적 주소에 해당하는 커널 가상주소를 반환한다. 따라서 이 코드는 물리적 주소 1024\*1024에 해당하는 커널 가상주소를 반환하여 메모리 영역의 시작 주소를 변수 free\_start에 저장한다.
2. page의 수와 PGSIZE(page 크기)를 곱해서 구한 전체 영역의 크기를 free\_end에 저장한다.
3. 사용 가능한 전체 메모리 영역의 페이지 개수를 구하여 free\_pages에 저장한다.
4. 전체 페이지 개수 중 절반을 user\_pages에 저장한다.
5. user\_pages(유저 페이지 수)가 user\_page\_limit(유저 페이지의 최대 크기)보다 크면 user\_pages를 user\_page\_limit값으로 설정한다.
6. user\_page에 할당하고 남은 나머지 페이지 개수를 kernel\_pages에 할당한다.
7. base address가 free\_start이고, page 개수가 kernel\_pages개이며 이름이 “kernel pool”인 memory pool을 생성한다.
8. base address가 (free\_strart + kernel\_pages \* PGSIZE)이고 page 개수가 user\_pages개인 memory pool을 생성한다.

palloc\_init() 동작이 완료되면 다음 그림과 같이 동적 메모리 할당영역이 kernel pool과 user pool로 나뉘게 된다.



**4. [function] malloc\_init()**

malloc\_init()은 임의 크기의 메모리 블록 할당을 처리하는 allocator를 초기화한다.

**void**

malloc\_init (**void**)

{

size\_t block\_size;

**for** (block\_size = 16; block\_size < PGSIZE / 2; block\_size \*= 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);

}

}

**5.[function] paging\_init()**

paging system을 초기화하는 함수이다.

**static** **void**

paging\_init (**void**)

{

uint32\_t \*pd, \*pt;

size\_t page;

**extern** **char** \_start, \_end\_kernel\_text;

pd = init\_page\_dir = palloc\_get\_page (PAL\_ASSERT | PAL\_ZERO);

pt = NULL;

**for** (page = 0; page < init\_ram\_pages; page++)

{

uintptr\_t paddr = page \* PGSIZE;

**char** \*vaddr = ptov (paddr);

size\_t pde\_idx = pd\_no (vaddr);

size\_t pte\_idx = pt\_no (vaddr);

bool in\_kernel\_text = &\_start <= vaddr && vaddr < &\_end\_kernel\_text;

**if** (pd[pde\_idx] == 0)

{

pt = palloc\_get\_page (PAL\_ASSERT | PAL\_ZERO);

pd[pde\_idx] = pde\_create (pt);

}

pt[pte\_idx] = pte\_create\_kernel (vaddr, !in\_kernel\_text);

}

* **2,3,4 함수의 실행이 끝나면 커널의 메모리 영역이 초기화된다.**

**6.[function] intr\_init()**

intr\_init()은 IDT(interrupt descriptor table)를 초기화하여 interrupt 처리를 준비하는 함수이다. IDT에는 각 번호에 해당하는 인터럽트를 처리하는 함수가 존재한다. 함수는 0번부터 255번까지 총 256개가 존재한다.

**void**

intr\_init (**void**)

{

uint64\_t idtr\_operand;

**int** i;

*/\* Initialize interrupt controller. \*/*

pic\_init (); //(1)

*/\* Initialize IDT. \*/*

**for** (i = 0; i < INTR\_CNT; i++)

idt[i] = make\_intr\_gate (intr\_stubs[i], 0); //(2)

*/\* Load IDT register.\*/*

idtr\_operand = make\_idtr\_operand (**sizeof** idt - 1, idt); //(3)

**asm** **volatile** ("lidt %0" : : "m" (idtr\_operand)); //(3)

*/\* Initialize intr\_names. \*/*

**for** (i = 0; i < INTR\_CNT; i++) //(4)

intr\_names[i] = "unknown";

intr\_names[0] = "#DE Divide Error";

intr\_names[1] = "#DB Debug Exception";

intr\_names[2] = "NMI Interrupt";

intr\_names[3] = "#BP Breakpoint Exception";

intr\_names[4] = "#OF Overflow Exception";

intr\_names[5] = "#BR BOUND Range Exceeded Exception";

intr\_names[6] = "#UD Invalid Opcode Exception";

intr\_names[7] = "#NM Device Not Available Exception";

intr\_names[8] = "#DF Double Fault Exception";

intr\_names[9] = "Coprocessor Segment Overrun";

intr\_names[10] = "#TS Invalid TSS Exception";

intr\_names[11] = "#NP Segment Not Present";

intr\_names[12] = "#SS Stack Fault Exception";

intr\_names[13] = "#GP General Protection Exception";

intr\_names[14] = "#PF Page-Fault Exception";

intr\_names[16] = "#MF x87 FPU Floating-Point Error";

intr\_names[17] = "#AC Alignment Check Exception";

intr\_names[18] = "#MC Machine-Check Exception";

intr\_names[19] = "#XF SIMD Floating-Point Exception";

}

1. PIC(Programmable Interrupt Controller)를 초기화한다. intr\_init()이 CPU의 IDT를 설정할 때, 인터럽트 처리를 위한 PIC도 초기화된다.
2. IDT를 초기화 한다. 각 interrupt vector별 처리 함수의 시작주소로 값이 초기화 되어있는 배열인 intr\_stubs를 이용하여 IDT를 초기화 한다.
3. IDT register를 불러온다.
4. interrupt의 이름을 초기화 시킨다.

**7.[function] timer\_init()**

timer interrupt handling을 준비하는 함수이다. timer가 있어야 시간의 흐름을 알 수 있고 timesharing이 가능하게 된다.

**void**

timer\_init (**void**)

{

pit\_configure\_channel (0, 2, TIMER\_FREQ);

intr\_register\_ext (0x20, timer\_interrupt, "8254 Timer");//(1)

}

1. vector로 0x20이 부여된 외부 인터럽트가 발생할 때 호출할 handler로 timer\_interrupt를 등록하고, 디버깅을 위해서 8254 Timer 라는 이름을 지정한다. 그 결과 8254 Timer는 주기적으로 시스템에 interrupt를 발생시킬 수 있게 된다.

**8.[function] kbd\_init()**

keyboard interrupt handling을 준비하는 함수이다.

**void**

kbd\_init (**void**)

{

intr\_register\_ext (0x21, keyboard\_interrupt, "8042 Keyboard"); //(1)

}

1. vector로 0x21이 부여된 외부 인터럽트가 발생할 때 호출할 handler로 keyboard\_interrupt를 등록하고 디버깅을 위해서 8042 Keyboard 라는 이름을 지정한다.

**8-1.[function]intr\_register\_ext()**

디버깅을 위한 이름을 지정하고 그것을 호출하기 위해 외부 interrupt vector number를 등록하는 함수이다. 등록한 handler는 interrupt가 꺼진 상태에서 실행될 것이다.

1. 매개변수로 전달받은 vec\_no가 제대로 된 범위내에 있는지 확인한다.

**void**

intr\_register\_ext (uint8\_t vec\_no, intr\_handler\_func \*handler,

**const** **char** \*name)

{

ASSERT (vec\_no >= 0x20 && vec\_no <= 0x2f); //(1)

register\_handler (vec\_no, 0, INTR\_OFF, handler, name); //(2)

}

1. interrupt handler를 호출하기 위해 interrupt vector number를 지정하고 디버깅을 위한 이름을 부여한다.

**9.[function] input\_init()**

serial 및 키보드 입력을 하나의 스트림으로 병합하도록 설정하는 함수이다**.**

**void**

input\_init (**void**)

{

intq\_init (&buffer); // (1)

}

1. interrupt queue를 초기화 하는 함수이다. input buffer를 초기화 시킨다.

* **5,6,7,8 함수의 실행이 끝나면 전체적인 시스템의 interrupt가 동작하도록 interrupt 시스템이 초기화 된다.**

**10.[function] thread\_start()**

idle thread를 생성하고 interrupt가 발생하도록 만들어 스케쥴링을 시작할 수 있다.

**void**

thread\_start (**void**)

{

*/\* Create the idle thread. \*/*

**struct** semaphore idle\_started;

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

thread\_create ("idle", PRI\_MIN, idle, &idle\_started); // (1)

*/\* Start preemptive thread scheduling. \*/*

intr\_enable (); // (2)

*/\* Wait for the idle thread to initialize idle\_thread. \*/*

sema\_down (&idle\_started); // (3)

}

1. thread\_create()는 thread를 만드는 함수이다. 이 코드는 새로 생성될 thread에 “idle” 이라는 이름과 우선순위로는 PRI\_MIN을 부여하고, 새 thread가 수행할 함수인 idle과 그 함수에게 주는 parameter인 &idle\_started를 넘겨준다. 이 함수의 수행 결과로 만들어진 idle thread는 컴퓨터가 실행할 프로그램이 없을 때 실행되는 thread이다.
2. interrupt가 발생하도록 하는 함수이다.

**10-1.[function] 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)

*/\* Allocate thread. \*/*

t = palloc\_get\_page (PAL\_ZERO); // (2)

**if** (t == NULL) // (3)

**return** TID\_ERROR;

*/\* Initialize thread. \*/*

init\_thread (t, name, priority); //(4)

tid = t->tid = allocate\_tid (); //(5)

*/\* Prepare thread for first run by initializing its stack.*

*Do this atomically so intermediate values for the 'stack'*

*member cannot be observed. \*/*

old\_level = intr\_disable (); //(6)

*/\* Stack frame for kernel\_thread(). \*/* //(7)

kf = alloc\_frame (t, **sizeof** \*kf);

kf->eip = NULL;

kf->function = function;

kf->aux = aux;

*/\* Stack frame for switch\_entry(). \*/* //(8)

ef = alloc\_frame (t, **sizeof** \*ef);

ef->eip = (**void** (\*) (**void**)) kernel\_thread;

*/\* Stack frame for switch\_threads(). \*/* //(9)

sf = alloc\_frame (t, **sizeof** \*sf);

sf->eip = switch\_entry;

sf->ebp = 0;

1. 매개변수로 전달받은 thread가 실행할 함수가 NULL이 아닌지 확인한다.

intr\_set\_level (old\_level); // (10)

*/\* Add to run queue. \*/*

thread\_unblock (t); // (11)

**return** tid; // (12)

}

1. 새로운 TCB를 생성하기 위해 메모리를 할당하여 변수 t에 저장한다. t는 새로 생성될 thread의 TCB를 가리키는 포인터이다. palloc\_get\_page() 함수는 palloc\_init()으로 초기화된 동적 메모리할당 영역에서 메모리를 할당 받는데, 할당받을 수 없는 경우 NULL을 반환한다.
2. (2)에서 메모리를 할당 받을 수 없어 NULL값을 반환받은 경우 TID\_ERROR를 리턴한다.
3. 새로 할당받은 TCB에 이름과 우선순위를 부여하고 초기화한다. ([function] 1-1)
4. 새 thread에 allocate\_tid()함수를 호출하여 tid를 부여한다.
5. intr\_disable()함수를 호출하여 interrupt를 비활성화 시키고(발생하지 못하게 한다.) 이전 interrupt 상태를 반한하여 old\_level에 저장한다.
6. kernel\_thread()함수에서 사용되는 스택영역을 초기화한다.
7. switch\_entry()함수에서 사용되는 스택영역을 초기화한다.
8. switch\_threads()함수에서 사용되는 스택영역을 초기화한다.

* 새로 생성된 thread를 처음 실행하는 것은 예외적인 경우이다. 새로 생성되어 실행된적이 없는 thread와 실행중인 thread사이에서 문맥교환이 일어날 때 필요한 stack frame을 (7)~(9)의 과정을 통해 초기화 한다. 형식에만 맞춰서 초기화하기 때문에 fake stack frame을 생성한다고 한다.

1. intr\_set\_level()함수를 호출하여 old\_level에 따라 interrupt를 활성화하거나 비활성화한다. (6)에서interrupt를 비활성화하기 전의 상태로 돌려 놓는다.
2. thread\_unblock()함수를 호출하여 새로 생성된 thread를 ready queue에 연결하여 ready 상태로 들어 최초 실행 준비를 마친다.
3. 새로 생성된 thread의 tid를 반환한다.

**10-1-1. [function] thread\_unblock()**

blocked(waiting) 상태인 thread를 ready 상태로 바꾸는 함수이다.

10-1.[function]thread\_create()함수는 이 함수를 호출하여 새로 생성되는 thread를 ready상태로 바꾸고 ready queue에 연결하여 최초 실행 준비를 마친다.

**void**

thread\_unblock (**struct** thread \*t)

{

**enum** intr\_level old\_level;

ASSERT (is\_thread (t));

old\_level = intr\_disable (); //(1)

ASSERT (t->status == THREAD\_BLOCKED); //(2)

list\_push\_back (&ready\_list, &t->elem); //(3)

t->status = THREAD\_READY; //(4)

intr\_set\_level (old\_level); //(5)

}

1. intr\_disable()함수를 호출하여 interrupt를 비활성화 하고(발생하지 못하게 한다.) 이전 interrupt 상태를 반한하여 old\_level에 저장한다.
2. THREAD\_BLOCKED 상태인 thread의 상태를 바꾸는 함수이기 때문에, 매개변수로 전달받은 thread의 상태가 THREAD\_BLOCK인지 확인한다. 새로 생성되는 thread의 경우, thread\_create() 함수내에서 호출되는 init\_thread()를 통해 THREAD\_BLOCKED상태로 초기화된다.
3. 전달받은 thread를 TCB의 구조체 중 elem을 이용하여 ready\_list에 연결한다. 이코드를 통해 thread가 ready queue에 들어가게 된다.
4. thread의 상태를 THREAD\_READY로 바꾼다.
5. intr\_set\_level()함수를 호출하여 old\_level에 따라 interrupt를 활성화하거나 비활성화한다. (1)에서 interrupt를 비활성화시키기 전의 상태로 돌려 놓는다.

**11. [function]serial\_init\_queue()**

interrupt 기반의 입출력으로 serial port 장치를 초기화 한다.

**void**

serial\_init\_queue (**void**)

{

**enum** intr\_level old\_level;

**if** (mode == UNINIT) // (1)

init\_poll ();

ASSERT (mode == POLL); // (2)

intr\_register\_ext (0x20 + 4, serial\_interrupt, "serial"); // (3)

mode = QUEUE;

old\_level = intr\_disable (); // (4)

write\_ier (); // (5)

intr\_set\_level (old\_level); // (6)

}

1. mode가 UNINT이면 init\_poll() 함수를 호출하여 serial port 장치를 초기화 한다. init\_poll() 함수 내부에서 mode가 POLL로 바뀐다.
2. mode가 POLL인지 확인한다.
3. vector로 0x20 + 4가 부여된 외부 인터럽트가 발생할 때 호출할 handler로 serial\_interrupt를 등록하고 디버깅을 위해서 “serial” 라는 이름을 지정한다.([function]8-1)
4. intr\_disable()함수를 호출하여 interrupt를 비활성화 하고 이전 interrupt 상태를 반한하여 old\_level에 저장한다.
5. interrupt enable register를 업데이트 한다.
6. intr\_set\_level()함수를 호출하여 old\_level에 따라 interrupt를 활성화하거나 비활성화한다. (1)에서 interrupt를 비활성화시키기 전의 상태로 돌려 놓는다.

**12. [function]timer\_calibrate()**

짧은 딜레이를 구현하는 데 사용되는 loops\_per\_tick를 보정한다.

**void**

timer\_calibrate (**void**)

{

**unsigned** high\_bit, test\_bit;

ASSERT (intr\_get\_level () == INTR\_ON);

printf ("Calibrating timer... ");

*/\* Approximate loops\_per\_tick as the largest power-of-two*

*still less than one timer tick. \*/*

loops\_per\_tick = 1u << 10;

**while** (!too\_many\_loops (loops\_per\_tick << 1))

{

loops\_per\_tick <<= 1;

ASSERT (loops\_per\_tick != 0);

}

*/\* Refine the next 8 bits of loops\_per\_tick. \*/*

high\_bit = loops\_per\_tick;

**for** (test\_bit = high\_bit >> 1; test\_bit != high\_bit >> 10; test\_bit >>= 1)

**if** (!too\_many\_loops (high\_bit | test\_bit))

loops\_per\_tick |= test\_bit;

printf ("%'"PRIu64" loops/s.\n", (uint64\_t) loops\_per\_tick \* TIMER\_FREQ);

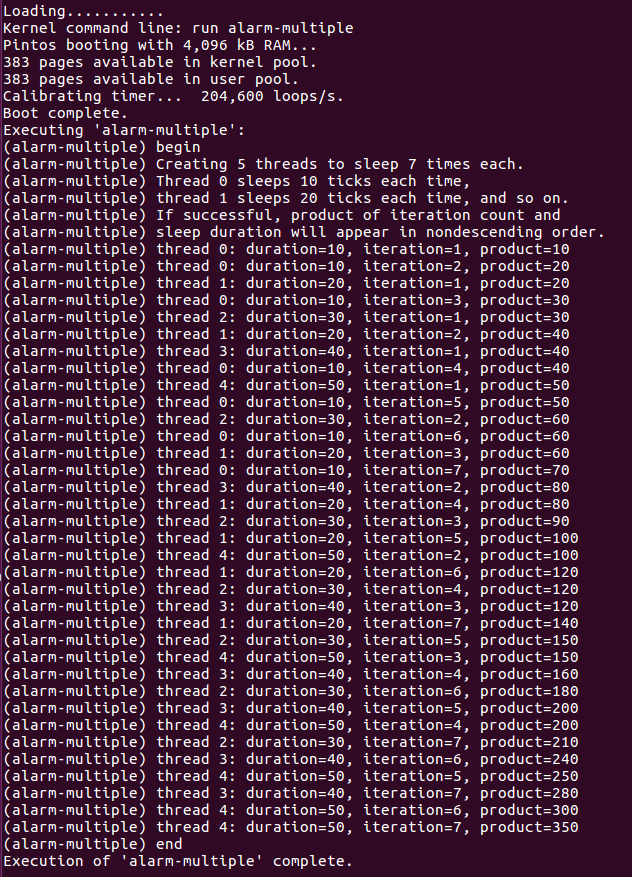
}

**13. printf ("Boot complete.\n");**

12까지 완료되면 부팅이 완료되고 부팅이 완료됐다는 메시지를 출력한다.

**14. [function]run\_actions()**

command line으로 입력된 test를 수행하는 함수이다. alarm-multiple을 전달받았으므로 alarm-multiple test를 수행하게 된다. 다음은 alarm-multiple test의 실행화면이다.



Loading..........

Kernel command line: run alarm-multiple // (1)

Pintos booting with 4,096 kB RAM... // (2)

383 pages available in kernel pool. // (3)

383 pages available in user pool. // (4)

Calibrating timer…204,600 loops/s. // (5)

Boot complete. // (6)

Executing ‘alarm-multiple’: // (7)

(alarm-multiple) begin

…

(alarm-multiple) end

Execution of ‘alarm-multiple’ complete.

1. read\_commad\_line()([function]1)에서 출력된다.
2. main함수에서 출력된다.
3. palloc\_init()([function]3) 함수 내부의 (7)에서 호출되는 init\_pool() 함수에서 출력된다. kernel pool에서 사용가능한 페이지의 수를 출력한다.
4. palloc\_init()([function]3) 함수 내부의 (8)에서 호출되는 init\_pool() 함수에서 출력된다. user pool에서 사용가능한 페이지의 수룰 출력한다.
5. timer\_calibrate()([function]12)에서 출력된다.
6. main()함수에서 출력되는 부팅이 완료되었다는 메시지다.
7. alarm-multiple test가 수행된다.