운영체제

Pintos Project #1 – Pintos 환경 구축

****

컴퓨터과학전공

운영체제 2분반

201311154 이재민

**목차**

**Ⅰ. 명령어 “pintos -v -- run alarm-multiple” 실행**

**Ⅱ. 프로그램 수행 경로에 따른 주요 함수, 자료구조 분석**

**1. main() 코드 분석**

**2. main()에서 호출하는 주요 함수 분석**

(a) bss.init()

(b) thread\_init()

(b). 1. lock\_init()

(b). 2. init\_thread()

(c) palloc\_init() – palloc\_init(user\_page\_limit)

(d) malloc\_init()

(e) paging\_init()

(f) intr\_init()

(g) thread\_start()

(g). 1. idle()

(g). 2. thread\_create()

(g). 3. thread\_unblock()

(g). 4. thread\_block()

(h) run\_action(argv)

**3. main()에서 사용되는 자료구조 분석**

(a) thread 자료구조

(b) lock 자료구조

(c) semaphore 자료구조

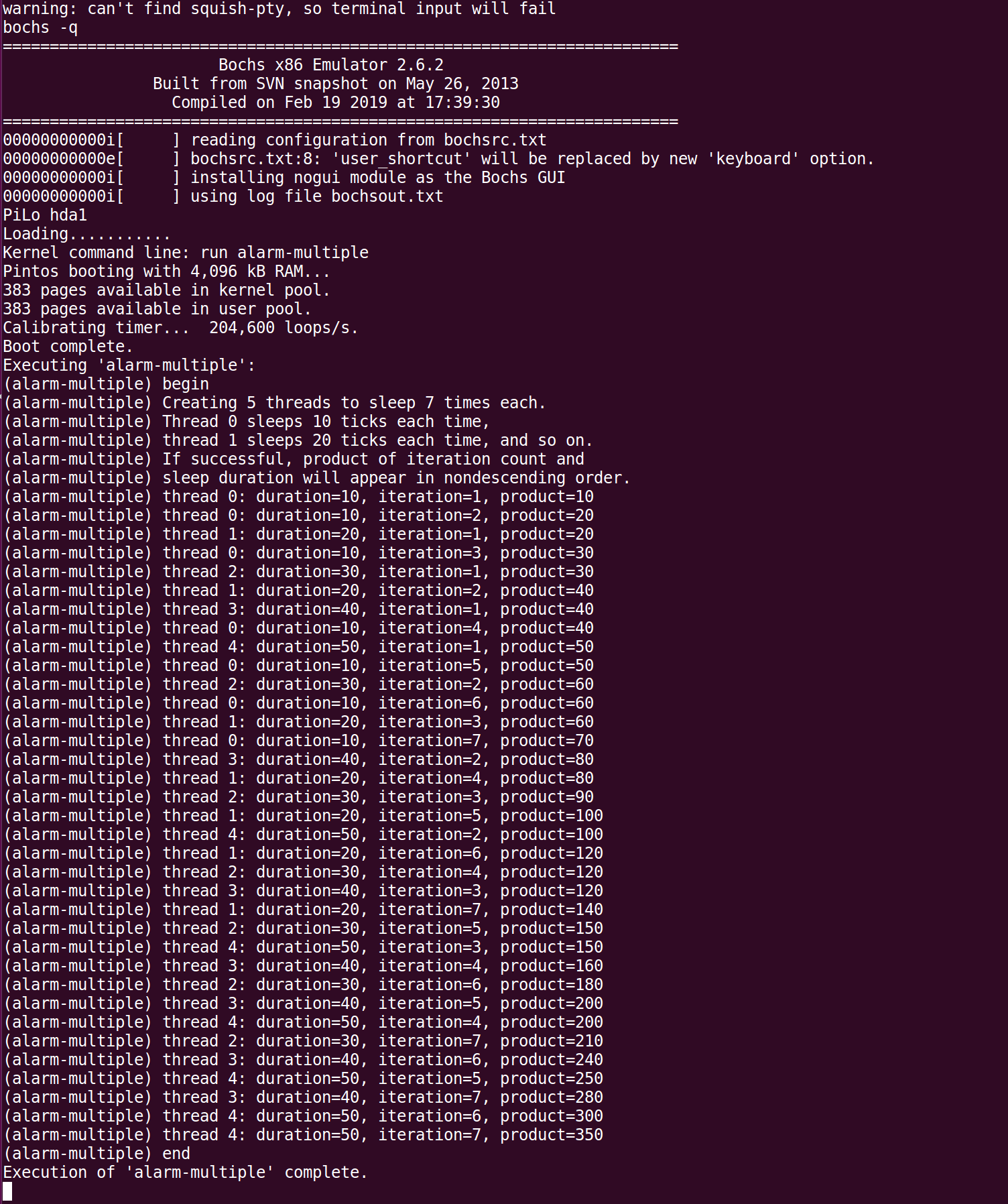
(d) action 자료구조

(e) intr\_frame 자료구조

(f) pool 자료구조

**4. main()의 실행 경로 분석**

Ⅰ. 명령어 “pintos -v -- run alarm-multiple” 실행



🡪 결과 : Pintos 정상 작동하는 것을 확인

**pintos –v -- run alarm-multiple**

- -v는 bochs로 전달 되는 옵션

- run alarm-multiple은 pintos kernel에 전달되는 옵션

- /pintos/src/threads/init.c 의 main이 실행되어 핀토스 실행

- main의 수행으로 bss\_init 으로 초기화되고 run\_action() 함수를 호출

- run alarm-multiple은 run과 alarm-multiple을 인자로 전달하게 되고 run이므로 run\_task()가 호출

- run\_task의 process\_wait(process\_excute(task)); 호출 🡪 task는 run으로 넘겨준 인자

- process\_excute에서 page 설정, thread 생성, thread\_create 호출

- run\_test 함수로 alarm-multiple을 인자로 수행 🡪 struct test의 일치하는 부분이 있는 {"alarm-multiple", test\_alarm\_multiple} 에서 test\_alarm\_multiple함수의 주소를 호출

**creating 5 threads to sleep 7 times each.**

- creating 5는 생성할 thread의 개수

- 7 times는 반복할 수

**thread 0 sheeps 10 tisks each time**

**thread 1 sleeps 20 ticks each time, and so on**

- 0번 thread는 10 ticks 간격

- 1번 thread는 20 ticks 간격

- 2번 thread는 30 ticks 간격

- 3번 thread는 40 ticks 간격

- 4번 thread는 50 ticks 간격

**thread n : duration = n , iteration = n , product = n ….**

- 이후의 실행은 thread 들이 실행된 결과를 보여주는 것

- n번 thread가 n ticks의 간격이라는 것과 n번의 누적 횟수를 가짐

- 실행 시간은 n ticks에서 실행됨

**end**

- run\_test 함수에서의 프로그램 종료 메시지

**Execution of ‘alarm-multiple’ complete.**

- run\_task에서 프로그램 완료 메시지

Ⅱ. 프로그램 수행 경로에 따른 주요 함수, 자료구조 분석

1. main() 코드 분석

|  |
| --- |
| /\* Pintos main program. \*/  int  main(void)  {  **char \*\*argv; // (1)**  /\* Clear BSS. \*/  **bss\_init (); // (2)**  /\* Break command line into arguments and parse options. \*/  **argv = read\_command\_line (); // (3)**  **argv = parse\_options (argv);**  /\* Initialize ourselves as a thread so we can use locks,  then enable console locking. \*/  **thread\_init (); // (4)**  **console\_init (); // (5)**  /\* Greet user. \*/  printf ("Pintos booting with %'"PRIu32" kB RAM...\n",  **init\_ram\_pages \* PGSIZE / 1024); // (6)**  /\* Initialize memory system. \*/  **palloc\_init (user\_page\_limit); // (7)**  **malloc\_init (); // (8)**  **paging\_init (); // (9)**  /\* Segmentation. \*/  #ifdefUSERPROG  tss\_init ();  gdt\_init ();  #endif  /\* Initialize interrupt handlers. \*/  **intr\_init (); // (10)**  **timer\_init (); // (11)**  **kbd\_init (); // (12)**  **input\_init (); // (13)**  #ifdefUSERPROG  exception\_init ();  syscall\_init ();  #endif  /\* Start thread scheduler and enable interrupts. \*/  **thread\_start (); // (14)**  **serial\_init\_queue (); // (15)**  **timer\_calibrate (); // (16)**  #ifdefFILESYS  /\* Initialize file system. \*/  ide\_init ();  locate\_block\_devices ();  filesys\_init (format\_filesys);  #endif  **printf ("Boot complete.\n"); // (17)**    /\* Run actions specified on kernel command line. \*/  **run\_actions (argv); // (18)**  /\* Finish up. \*/  **shutdown (); // (19)**  **thread\_exit (); // (20)**  } |

**<main – pintos/src/threads/init.c>**

(1) argument 값의 포인터 선언을 의미한다.

(2) kernel RAM을 초기화하기 위해 커널의 “BSS” segment를 초기화한다.

\* “BSS” segment는 초기화 되지 않는 전역 변수의 메모리 영역이다.

(3) argv = read\_command\_line (); → argument 값을 분석한다. (배열에 저장)

argv = parse\_options (argv); → Kernel을 실행했을 때 옵션을 가져온다.

(4) 최초의 thread를 생성한다.

(5) console에 메시지를 초기화하기 위해 console을 초기화한다.

(6) 사용자의 인사 메시지로, RAM 값을 초기화함과 동시에 저장한다.

(7) 동적 메모리 할당 영역을 초기화하는 동작으로 그 크기에 제한을 두었다.

(8) 동적 메모리 할당 영역을 초기화하는 동작이다.

(9) 페이징 시스템을 초기화하는 동작이다.

(10) 인터럽트를 초기화한다.

(11) 타이머를 초기화한다.

(12) 키보드를 초기화한다.

(13) 입력 버퍼를 초기화한다.

(14) idle thread를 생성한다.

(15) 시리얼 포트로 오는 I/O 인터럽트를 초기화한다.

(16) 타이머를 설정한다.

(17) 부트가 완료되었다는 메시지를 출력한다. (부팅완료)

(18) 시험 프로그램을 실행시킨다.

(19) windows를 예약 종료를 요청한다.

(20) 현재 thread를 종료한다.

2. main()에서 호출하는 주요 함수 분석

(a) bss.init()

|  |
| --- |
| static void  bss\_init (void)  {  extern char \_start\_bss, \_end\_bss;  memset (&\_start\_bss, 0, &\_end\_bss - &\_start\_bss);  } |

**<bss\_init – pintos/src/threads/init.c>**

- BSS 영역을 초기화한다.

- BSS : 초기화되지 않은 전역 변수의 메모리 영역이다.

pintos/src/threads/kernel.ld.s에 정의된 \_start\_bss에서 \_end\_bss까지의 메모리 영역을 초기화한다.

(b) thread\_init()

|  |
| --- |
| void  thread\_init (void)  {  **ASSERT (intr\_get\_level () == INTR\_OFF); // (1)**  **lock\_init (&tid\_lock); // (2)**  **list\_init (&ready\_list); // (3)**  list\_init (&all\_list);  /\* Set up a thread structure for the running thread. \*/  **initial\_thread = running\_thread (); // (4)**  **init\_thread (initial\_thread, "main", PRI\_DEFAULT); // (5)**  **initial\_thread->status = THREAD\_RUNNING; // (6)**  **initial\_thread->tid = allocate\_tid (); // (7)**  } |

**<thread\_init() – pintos/src/threads/thread.c>**

(1) 현재의 인터럽트 상태가 OFF면 계속 진행하고 아니면 에러메세지 출력 후 종료한다.

\* 이 명령어는 현재 상태에서 꼭 만족되어야 하는 조건을 가지고 있는 명령어로, 그것을 확인하는 차원에서 사용되어진다.

⇒ 현재 인터럽트가 꺼져 있는지 확인한다.

(2) tid\_lock을 초기화한다.

(3) ready\_list를 초기화한다.

\* ready\_list는 pintos에서의 ready queue이기에 ready queue를 초기화하는 동작이다.

(4) stack pointer로부터 현제 thread의 주소를 얻는다.

(5) 현재 thread를 초기화한다.

(6) 현재 thread의 상태를 running 상태로 만들어준다.

(7) 현재 thread에 id를 부여한다.

(b). 1. lock\_init()

|  |
| --- |
| void  lock\_init (struct lock \*lock)  {  ASSERT (lock != NULL);  lock->holder = NULL;  **sema\_init (&lock->semaphore, 1); // (1)**  } |

**<lock\_init – pintos/src/threads/synch.c>**

- thread 동기화에 사용되는 lock을 초기화하는 함수이다.

- Binary semaphore이기 때문에 0과 1을 사용하고, 1일 때 접근할 수 있다.

(1) lock에 사용하는 semaphore를 초기화한다.

(b). 2. init\_thread()

|  |
| --- |
| 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);  } |

**<init\_thread – pintos/src/threads/thread.c>**

(1) thread를 실행시키기 위해 일단 THREAD\_BLOCKED 처리한다.

(2) thread의 이름을 설정한다.

(3) thread의 stack pointer 값을 저장한다.

(4) thread의 우선순위 값을 저장한다.

(5) thread의 magic 값을 저장한다.

(c) palloc\_init() – palloc\_init(user\_page\_limit)

|  |
| --- |
| 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);  uint8\_t \*free\_end = ptov (init\_ram\_pages \* PGSIZE);  size\_t free\_pages = (free\_end - free\_start) / PGSIZE;  size\_t user\_pages = free\_pages / 2;  size\_t kernel\_pages;  if (user\_pages > user\_page\_limit)  user\_pages = user\_page\_limit;  **kernel\_pages = free\_pages - user\_pages; // (1)**  /\* Give half of memory to kernel, half to user. \*/  **init\_pool (&kernel\_pool, free\_start, kernel\_pages, "kernel pool"); // (2)**  init\_pool (&user\_pool, free\_start + kernel\_pages \* PGSIZE,  **user\_pages, "user pool"); // (3)**  } |

**<palloc\_init – pintos/src/threads/palloc.c>**

(1) cpu 내부에 page를 allocate한다.

- 위의 코드 9에서 주석 처리로 Free memory starts at 1 MB and runs to the end of RAM. 라고 되어 있기 때문에 시작 부분을 1MB로 설정해주고 kernel pool과 user pool로 나눈다.

(2) memory의 빈 영역 중 앞의 절반을 kernel pool로 할당한다.

(3) memory 영역의 나머지 영역을 user pool로 할당한다.

(d) malloc\_init()

|  |
| --- |
| /\* Initializes the malloc() descriptors. \*/  void  malloc\_init (void)  {  size\_t block\_size;  **for (block\_size = 16; block\_size < PGSIZE / 2; block\_size \*= 2) // (1)**  **{**  **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); // (2)**  **}**  } |

**<malloc\_init – pintos/src/threads/malloc.c>**

- Memory descriptor인 malloc을 초기화한다.

(1) 블록의 사이즈를 16으로 초기화하고 반복문 내부로 들어갈 수 있을 때까지 반복한다.

(2) 조건이 만족돼 반복문 내부로 들어가는게 성공하면, 빈 공간을 얻는다.

(e) paging\_init()

|  |
| --- |
| 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); // (1)**  pt = NULL;  **for (page = 0; page < init\_ram\_pages; page++) // (2)**  **{**  **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**  **}**  /\* Store the physical address of the page directory into CR3  aka PDBR (page directory base register). This activates our  new page tables immediately. See [IA32-v2a] "MOV--Move  to/from Control Registers" and [IA32-v3a] 3.7.5 "Base Address  of the Page Directory". \*/  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init\_page\_dir)));  } |

**<paging\_init – pintos/src/threads/init.c>**

- Paging 시스템을 초기화한다.

(1) palloc\_get\_page 함수를 이용하여 페이지를 할당 받아 base\_page\_dir에 저장한다.

(2) 동작해야 할 page들이 virtual address space와 physical address space 사이를 움직이며 필요한 data들을 생성한다.

(f) intr\_init()

|  |
| --- |
| /\* Initializes the interrupt system. \*/  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.  See [IA32-v2a] "LIDT" and [IA32-v3a] 5.10 "Interrupt  Descriptor Table (IDT)". \*/  idtr\_operand = make\_idtr\_operand (sizeof idt - 1, idt);  **asm volatile ("lidt %0" : : "m" (idtr\_operand)); // (3)**  /\* Initialize intr\_names. \*/  for (i = 0; i < INTR\_CNT; i++)  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"; // (4)**  } |

**<intr\_init – pintos/src/threads/interrupt.c>**

- Interrupt system을 Interrupt Descriptor Table(IDT)를 이용해 초기화한다.

(1) 인터럽트 컨트롤러인 PIC(Process Interrupt Controller)을 초기화한다.

(2) IDT(Interrupt Descriptor Table)를 초기화하기 위해 idt 배열에 descriptor를 copy한다.

\*IDT는 pintos가 인터럽트를 초기화하기 위해 cpu가 만든 자료구조이다.

(3) idtr이 IDT의 시작 위치를 가리키게 한다.

(4) intr\_names 배열을 초기화한다.

(g) thread\_start()

|  |
| --- |
| /\* Starts preemptive thread scheduling by enabling interrupts.  Also creates the idle thread. \*/  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)**  } |

**<thread\_start – pintos/src/threads/thread.c>**

(1) idle thread를 생성한다.

(2) preemptive thread scheduling을 시작한다.

(3) idle thread의 초기화를 기다린다.

(g). 1. idle()

|  |
| --- |
| static void  idle (void \*idle\_started\_ UNUSED)  {  struct semaphore \*idle\_started = idle\_started\_;  idle\_thread = thread\_current ();  sema\_up (idle\_started);  for (;;)  {  /\* Let someone else run. \*/  intr\_disable ();  thread\_block ();  asm volatile ("sti; hlt" : : : "memory");  }  } |

**<idle – pintos/src/threads/thread.c>**

- 새로 만들어진 idle thread가 실행할 코드이다.

(g). 2. 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);  /\* Allocate thread. \*/  t = palloc\_get\_page (PAL\_ZERO);  if (t == NULL)  return TID\_ERROR;  /\* Initialize thread. \*/  init\_thread (t, name, priority);  **tid = t->tid = allocate\_tid (); // (1)**  /\* 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 ();  /\* Stack frame for kernel\_thread(). \*/  kf = alloc\_frame (t, sizeof \*kf);  kf->eip = NULL;  kf->function = function;  **kf->aux = aux; // (2)**  /\* Stack frame for switch\_entry(). \*/  ef = alloc\_frame (t, sizeof \*ef);  **ef->eip = (void (\*) (void)) kernel\_thread; // (3)**  /\* Stack frame for switch\_threads(). \*/  sf = alloc\_frame (t, sizeof \*sf);  **sf->eip = switch\_entry; // (4)**  sf->ebp = 0;  intr\_set\_level (old\_level);  /\* Add to run queue. \*/  **thread\_unblock (t); // (5)**  return tid;  } |

**<thread\_create – pintos/src/threads/thread.c>**

(1) 새로 만들어질 thread의 4kb의 stack을 만들어 메모리를 할당하고 초기화한다.

(2) thread의 스택에서 kernel 부분을 초기화한다.

(3) thread의 스택에서 switch\_entry() 부분을 초기화한다.

(4) thread의 스택에서 switch\_threads() 부분을 초기화한다.

(5) thread 실행을 위해 ready queue에 넣어 ready 상태로 만들어 준다.

(g). 3. thread\_unblock()

|  |
| --- |
| void  thread\_unblock (struct thread \*t)  {  enum intr\_level old\_level;  ASSERT (is\_thread (t));  old\_level = intr\_disable ();  ASSERT (t->status == THREAD\_BLOCKED);  list\_push\_back (&ready\_list, &t->elem);  t->status = THREAD\_READY;  intr\_set\_level (old\_level);  } |

**<thread\_unblock – pintos/src/threads/thread.c>**

- thread를 실행할 수 있는 상태로 만들어 준다.

(g). 4. thread\_block()

|  |
| --- |
| void  thread\_block (void)  {  ASSERT (!intr\_context ());  ASSERT (intr\_get\_level () == INTR\_OFF);  thread\_current ()->status = THREAD\_BLOCKED;  schedule ();  } |

**<thread\_block – pintos/src/threads/thread.c>**

- thread\_unblock()에 의해 호출될 때까지 현재의 thread를 잠재운다.

(h) run\_action(argv)

|  |
| --- |
| /\* Executes all of the actions specified in ARGV[]  up to the null pointer sentinel. \*/  static void  run\_actions (char \*\*argv)  {  /\* An action. \*/  struct action  {  char \*name; /\* Action name. \*/  int argc; /\* # of args, including action name. \*/  void (\*function) (char \*\*argv); /\* Function to execute action. \*/  };  /\* Table of supported actions. \*/  static const struct action actions[] =  {  {"run", 2, run\_task},  #ifdef FILESYS  {"ls", 1, fsutil\_ls},  {"cat", 2, fsutil\_cat},  {"rm", 2, fsutil\_rm},  {"extract", 1, fsutil\_extract},  {"append", 2, fsutil\_append},  #endif  {NULL, 0, NULL},  };  while (\*argv != NULL)  {  const struct action \*a;  int i;  /\* Find action name. \*/  for (a = actions; ; a++)  if (a->name == NULL)  PANIC ("unknown action `%s' (use -h for help)", \*argv);  else if (!strcmp (\*argv, a->name))  break;  /\* Check for required arguments. \*/  for (i = 1; i < a->argc; i++)  if (argv[i] == NULL)  PANIC ("action `%s' requires %d argument(s)", \*argv, a->argc - 1);  /\* Invoke action and advance. \*/  a->function (argv);  argv += a->argc;  }  } |

**<run\_actions(argv) – pintos/src/threads/init.c>**

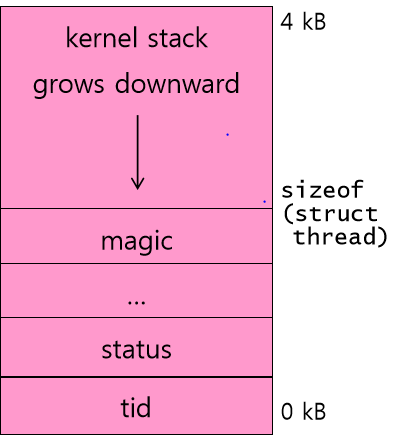
- null pointer가 끝날 때까지 argv[]에 지정된 모든 작업을 실행한다.

3. main()에서 사용되는 자료구조 분석

(a) thread 자료구조

|  |
| --- |
| struct thread  {  /\* Owned by thread.c. \*/  tid\_t tid; /\* Thread identifier. \*/  enum thread\_status status; /\* Thread state. \*/  char name[16]; /\* Name (for debugging purposes). \*/  uint8\_t \*stack; /\* Saved stack pointer. \*/  int priority; /\* Priority. \*/  struct list\_elem allelem; /\* List element for all threads list. \*/  /\* Shared between thread.c and synch.c. \*/  struct list\_elem elem; /\* List element. \*/  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir; /\* Page directory. \*/  #endif  /\* Owned by thread.c. \*/  unsigned magic; /\* Detects stack overflow. \*/  }; |

**<thread 자료구조(TCB) – pintos/src/threads/thread.h>**



**<thread 구조체 그림>**

**- tid**

: thread를 구분하는 고유의 id를 저장하는 영역이다.

**- status**

: thread의 현재 상태 정보를 저장하는 영역이다.

**- name**

: thread의 이름을 저장하는 영역이다.

**- stack**

: thread가 사용하는 stack의 sp를 저장하는 영역이다.

**- priority**

: thread의 우선순위를 저장하는 영역이다.

**- elem**

: thread 구조체 사이에 리스트 구성을 위해 이전 thread 구조체와 다음 thread 구조체의 포인터를 저장하는 영역이다.

**- magic**

: magic number는 thread 정보와 stack 영역 사이에 위치한다. stack overflow가 발생할 경우, magic number가 변경되기 때문에 에러를 검출하는데 사용된다.

(b) lock 자료구조

|  |
| --- |
| /\* Lock. \*/  struct lock  {  struct thread \*holder; /\* Thread holding lock (for debugging). \*/  struct semaphore semaphore; /\* Binary semaphore controlling access. \*/  }; |

**<lock 자료구조 – pintos/src/threads/synch.h>**

**- semaphore**

: lock에서 사용하는 semaphore이다.

(c) semaphore 자료구조

|  |
| --- |
| /\* A counting semaphore. \*/  struct semaphore  {  **unsigned value; /\* Current value. \*/ // (1)**  struct list waiters; /\* List of waiting threads. \*/  }; |

**<semaphore 자료구조 – pintos/src/threads/synch.h>**

(1) 부호가 없는 현재 값을 저장한다.

**- list waiters**

: waiting하고 있는 thread들의 리스트이다.

(d) action 자료구조

|  |
| --- |
| /\* An action. \*/  struct action  {  char \*name; /\* Action name. \*/  int argc; /\* # of args, including action name. \*/  void (\*function) (char \*\*argv); /\* Function to execute action. \*/  }; |

**<action 자료구조 – pintos/src/threads/init.c>**

**- name**

: action의 name

**- argc**

: action name을 포함한 args의 개수

**- function**

: action을 실행시킬 함수

(e) intr\_frame 자료구조

|  |
| --- |
| struct intr\_frame  {  /\* Pushed by intr\_entry in intr-stubs.S.  These are the interrupted task's saved registers. \*/  uint32\_t edi; /\* Saved EDI. \*/  uint32\_t esi; /\* Saved ESI. \*/  uint32\_t ebp; /\* Saved EBP. \*/  uint32\_t esp\_dummy; /\* Not used. \*/  uint32\_t ebx; /\* Saved EBX. \*/  uint32\_t edx; /\* Saved EDX. \*/  uint32\_t ecx; /\* Saved ECX. \*/  uint32\_t eax; /\* Saved EAX. \*/  uint16\_t gs, :16; /\* Saved GS segment register. \*/  uint16\_t fs, :16; /\* Saved FS segment register. \*/  uint16\_t es, :16; /\* Saved ES segment register. \*/  uint16\_t ds, :16; /\* Saved DS segment register. \*/  /\* Pushed by intrNN\_stub in intr-stubs.S. \*/  uint32\_t vec\_no; /\* Interrupt vector number. \*/  /\* Sometimes pushed by the CPU,  otherwise for consistency pushed as 0 by intrNN\_stub.  The CPU puts it just under `eip', but we move it here. \*/  uint32\_t error\_code; /\* Error code. \*/  /\* Pushed by intrNN\_stub in intr-stubs.S.  This frame pointer eases interpretation of backtraces. \*/  void \*frame\_pointer; /\* Saved EBP (frame pointer). \*/  /\* Pushed by the CPU.  These are the interrupted task's saved registers. \*/  void (\*eip) (void); /\* Next instruction to execute. \*/  uint16\_t cs, :16; /\* Code segment for eip. \*/  uint32\_t eflags; /\* Saved CPU flags. \*/  void \*esp; /\* Saved stack pointer. \*/  uint16\_t ss, :16; /\* Data segment for esp. \*/  }; |

**<intr\_frame 자료구조 – pintos/src/threads/interrupt.h>**

**- intr\_frame**

: 레지스터에 저장한 task의 interrupt를 cpu에 push한다.

(f) pool 자료구조

|  |
| --- |
| /\* A memory pool. \*/  struct pool  {  struct lock lock; /\* Mutual exclusion. \*/  struct bitmap \*used\_map; /\* Bitmap of free pages. \*/  uint8\_t \*base; /\* Base of pool. \*/  }; |

**<pool 자료구조 – pintos/src/threads/palloc.c>**

**- pool**

: 메모리의 저장 공간으로, kernel pool과 user pool이 있다.

**- lock**

: 상호 배제하는 thread들을 고정한다.

**- used\_map**

: 프리 페이지로 비트맵을 말한다.

**- base**

: 저장 공간의 base가 되는 곳이다.

4. main()의 실행 경로 분석

\* 위 🡪 아래 방향으로 순차 실행됨

|  |
| --- |
| **bss\_init ()** |
| **thread\_init ()** |
| **lock\_init()** |
| **sema\_init()** |
| **list\_init()** |
| **init\_thread()** |
| **console\_init()** |
| **lock\_init()** |
| **palloc\_init()** |
| **init\_pool()** |
| **lock\_init()** |
| **sema\_init()** |
| **malloc\_init()** |
| **paging\_init()** |
| **intr\_init()** |
| **pic\_init()** |
| **make\_intr\_gate()** |
| **make\_gate()** |
| **timer\_init()** |
| **kbd\_init()** |
| **intr\_register\_init()** |
| **input\_init()** |
| **intq\_init()** |
| **thread\_start()** |
| **idle()** |
| **thread\_create()** |
| **thread\_unblock()** |
| **thread\_block()** |
| **serial\_init\_queue()** |
| **init\_poll()** |
| **intr\_register\_ext()** |
| **intr\_disable()** |
| **write\_ier()** |
| **intr\_get\_level()** |
| **intr\_set\_level()** |
| **timer\_calibrate()** |
| **loops\_per\_tick()** |
| **run\_actions()** |
| **run\_task()** |
| **run\_test()** |
| **test\_alarm\_multiple()** |
| **test\_sleep()** |