운영체제 소스코드 분석 보고서

핀토스1번과제

201511449

컴퓨터과학과

김동현

1. 메인함수에 대한 간략한 설명

|  |
| --- |
| 1. int 2. main (void) 3. { 4. char \*\*argv; 5. bss\_init (); 초기화되지 않은 전역변수를 담은 bss영역을 초기화하는 작업을 수행합니다. 6. 부팅 중 커널에게 주어지는 argument 7. argv = read\_command\_line (); 8. argv = parse\_options (argv); 10. thread\_init (); 커널 자기자신을 thread화 합니다. 11. console\_init (); 12. intr\_init (); 인터럽트 영역을 초기화합니다. 13. timer\_init (); 타이머 인터럽트(주기적으로 time 신호를 cpu에게 준다) 14. kbd\_init (); 키보드 인터럽트 15. input\_init (); 16. thread\_start (); 스레드 시작, timer 가동, 멀티프로그래밍 시작 17. serial\_init\_queue (); 18. timer\_calibrate (); 19. ide\_init (); 20. locate\_block\_devices (); 21. filesys\_init (format\_filesys); 22. #endif 23. printf ("Boot complete.\n"); 24. run\_actions (argv); 테스트 프로그램 시작 25. shutdown (); 26. thread\_exit (); |

(2)함수 분석

1. The initial thread

|  |
| --- |
| 1. 1-1 2. void thread\_init (void) 3. { 4. ASSERT (intr\_get\_level () == INTR\_OFF); 5. lock\_init (&tid\_lock); 6. list\_init (&ready\_list); 7. list\_init (&all\_list); 8. initial\_thread = running\_thread (); 9. init\_thread (initial\_thread, "main", PRI\_DEFAULT); 10. initial\_thread->status = THREAD\_RUNNING; 11. initial\_thread->tid = allocate\_tid (); 12. } |

1. ASSERT(intr\_get\_level()==INTR\_OFF) : kernel코드에서 확인사항 생길 때 ASSERT함수 사용

intr\_get\_level()==INTR\_OFF

인터럽트 off유무 확인!!

1. list\_init(&ready\_list)

ready queue를 초기화

1. initial\_thread = running\_thread ();

tcb의 포인터(시작주소를 가리킨다.)

1. init\_thread (initial\_thread, "main", PRI\_DEFAULT);

tcb의 초기화

1. initial\_thread->status = THREAD\_RUNNING;

thread를 running상태로 초기화합니다.

1. initial\_thread->tid = allocate\_tid ();

새로운 tid할당🡺기존 tid+1

2)메모리 초기화

|  |
| --- |
| 2-1 void palloc\_init (size\_t user\_page\_limit)  {  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;  init\_pool (&kernel\_pool, free\_start, kernel\_pages, "kernel pool");  init\_pool (&user\_pool, free\_start + kernel\_pages \* PGSIZE,  user\_pages, "user pool");  } |

1. uint8\_t \*free\_start = ptov (1024 \* 1024);

핀토스의 linkscript 인 kernel.lds.s에 정의된 끝 주소를 참조하여 핀토스가 사용할

핀토스가 사용할 메모리 영역의 포인터를 변수 free\_start에 저장합니다.

1. uint8\_t \*free\_end = ptov (init\_ram\_pages \* PGSIZE);

전체 영역의 크기를 free\_end에 저장합니다.

1. size\_t free\_pages = (free\_end - free\_start) / PGSIZE;

사용 가능한 전체 메모리 영역의 페이지 개수를 free\_pages에 저장합니다.

1. size\_t user\_pages = free\_pages / 2;

전체 페이지 개수중 절반을 user\_pages에 할당합니다.

1. kernel\_pages = free\_pages - user\_pages;

남은 것은 kernel\_pages에 저장합니다.

6. init\_pool (&kernel\_pool, free\_start, kernel\_pages, "kernel pool");

(이름,시작 주소, 페이지 개수,메모리 풀)생성합니다.

7. init\_pool (&user\_pool, free\_start + kernel\_pages \* PGSIZE,

user\_pages, "user pool");

(이름,시작 주소, 페이지 개수,메모리 풀)생성합니다.

|  |
| --- |
| 2-2 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);  }  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init\_page\_dir)));  } |

가상 메모리를 운영하는 방법: paging기법

paging\_init();

페이징을 초기화합니다.

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

가상메모리를 동적으로 할당 합니다.

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

Page dir 생성하기(pd[pde\_idx] == 0 아직 만들지 않았으면)

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

Kernel page table생성

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

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

페이지 디렉토리와 페이지 테이블 인덱시 번호 부여하는 동작입니다.

3)인터럽트 핸들링 초기화

|  |
| --- |
| 3-1 void intr\_init (void)  {  uint64\_t idtr\_operand;  int i;  pic\_init ();  for (i = 0; i < INTR\_CNT; i++)  idt[i] = make\_intr\_gate (intr\_stubs[i], 0);  idtr\_operand = make\_idtr\_operand (sizeof idt - 1, idt);  asm volatile ("lidt %0" : : "m" (idtr\_operand));  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";  } |

1. pic\_init(); 인터럽트 제어기를 초기화합니다.:
2. for (i = 0; i < INTR\_CNT; i++)

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

IDT에 대한 초기화

|  |
| --- |
| 3-2 void intr\_register\_ext (uint8\_t vec\_no, intr\_handler\_func \*handler,  const char \*name)  {  ASSERT (vec\_no >= 0x20 && vec\_no <= 0x2f);  register\_handler (vec\_no, 0, INTR\_OFF, handler, name);  } |

Timer 인터럽트 처리함수와 keyboard 인터럽트 처리함수를 실행합니다.

|  |
| --- |
| 3-3 void timer\_init(void)  {  uint16\_t count=(1193180+ TIMER\_FREQ/2) / TIMER\_FREQ;  outb(0x43, 0x34)  outb(0x40, count& 0xff)  outb(0x40, cout>>8)  intr\_register\_ext(0x20, timer\_interrupt, “8254 Timer”);  } |

1. TIMER\_FREQ: 주기 100(핀토스내)==1초에 100번 인터럽트
2. intr\_register\_ext(0x20, timer\_interrupt, “8254 Timer”);

0x20==인터럽트 번호(32번): 인터럽트 번호

timer\_interrupt: timer 인터럽트 처리함수

8254 Timer==PIT(programmable interval timer)

주기적으로 cpu에게 인터럽트 요청

|  |
| --- |
| 3-4 void kbd\_init(void){  intr\_register\_ext(0x21, keyboard\_interrupt, “8042 keyboard”);  } |

4) thread scheduler 시작

|  |
| --- |
| 4-1 void intr\_handler (struct intr\_frame \*frame)  {  bool external;  intr\_handler\_func \*handler;  external = frame->vec\_no >= 0x20 && frame->vec\_no < 0x30;  if (external)  {  ASSERT (intr\_get\_level () == INTR\_OFF);  ASSERT (!intr\_context ());  in\_external\_intr = true;  yield\_on\_return = false;  }  handler = intr\_handlers[frame->vec\_no];  if (handler != NULL)  handler (frame);  else if (frame->vec\_no == 0x27 || frame->vec\_no == 0x2f)  {    }  else  unexpected\_interrupt (frame);  if (external)  {  ASSERT (intr\_get\_level () == INTR\_OFF);  ASSERT (!intr\_context ());  in\_external\_intr = true;  yield\_on\_return = false;  }  handler = intr\_handlers[frame->vec\_no];  if (handler != NULL)  handler (frame);  else if (frame->vec\_no == 0x27 || frame->vec\_no == 0x2f)  {    }  else  unexpected\_interrupt (frame);  if (external)  {  ASSERT (intr\_get\_level () == INTR\_OFF);  ASSERT (intr\_context ());  in\_external\_intr = false;  pic\_end\_of\_interrupt (frame->vec\_no);  if (yield\_on\_return)  thread\_yield ();  }  } |

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

interrupt가 off인지 확인

1. ASSERT (!intr\_context ());

인터럽트 context가 아닌지 확인

3. in\_external\_intr 외부 인터럽트 실행 flag

4. yield\_on\_return

timesharing에서 cpu스틸링 할지 말지를 결정

5. pic\_end\_of\_interrupt

waiting상태를 끝낼 수 있게 답장을 보냅니다.

1. thread\_yield ();

running thread를 ready로 만듦

|  |
| --- |
| 4-2 void thread\_start (void)  {  struct semaphore idle\_started;  sema\_init (&idle\_started, 0);  thread\_create ("idle", PRI\_MIN, idle, &idle\_started);  intr\_enable ();  sema\_down (&idle\_started);  } |

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

새로운 쓰레드를 생성합니다.(이름,우선순위, 수행할 함수,함수에게 줄 파라미터)

1. intr\_enable():

인터럽트 시작(타이머 인터럽트 동작)

|  |
| --- |
| 4-3 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);  t = palloc\_get\_page (PAL\_ZERO);  if (t == NULL)  return TID\_ERROR;  init\_thread (t, name, priority);  tid = t->tid = allocate\_tid ();  old\_level = intr\_disable ();  kf = alloc\_frame (t, sizeof \*kf);  kf->eip = NULL;  kf->function = function;  kf->aux = aux;  ef = alloc\_frame (t, sizeof \*ef);  ef->eip = (void (\*) (void)) kernel\_thread;  sf = alloc\_frame (t, sizeof \*sf);  sf->eip = switch\_entry;  sf->ebp = 0;  intr\_set\_level (old\_level);  thread\_unblock (t);  return tid;  } |

palloc\_get\_page: 4kb 동적메모리 할당

///////

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

kf->eip = NULL;

kf->function = function;

kf->aux = aux;

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

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

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

sf->eip = switch\_entry;

///////가짜 스택 프레임 만드는 과정입니다.

thread\_unblock :스레드를 깨우는 함수(waiting->ready)

1. list\_push\_back (&ready\_list, &cur->elem);

현재 쓰레드 구조체를 ready queue의 맨끝에 푸쉬합니다.

1. cur->status = THREAD\_READY;

현재상태를 running에서 ready로 변경합니다.

1. schedule ();

다음번 실행할 스레드를 선택하고 그 쓰레드로 문맥교환합니다.

|  |
| --- |
| 4-4 void thread\_yield (void)  {  struct thread \*cur = thread\_current ();  enum intr\_level old\_level;  ASSERT (!intr\_context ());  old\_level = intr\_disable ();  if (cur != idle\_thread)  list\_push\_back (&ready\_list, &cur->elem);  cur->status = THREAD\_READY;  schedule ();  intr\_set\_level (old\_level);  } |

|  |
| --- |
| 4-5 void thread\_block (void)  {  ASSERT (!intr\_context ());  ASSERT (intr\_get\_level () == INTR\_OFF);  thread\_current ()->status = THREAD\_BLOCKED;  schedule ();  } |

1.thread\_current ()->status = THREAD\_BLOCKED;

현재 스레드상태를 running에서 waiting으로 변경합니다.

Cf) Waiting==BLOCKED==sleep

|  |
| --- |
| 4-6 void thread\_exit (void)  {  ASSERT (!intr\_context ());  #ifdef USERPROG  process\_exit ();  #endif  intr\_disable ();  list\_remove (&thread\_current()->allelem);  thread\_current ()->status = THREAD\_DYING;  schedule ();  NOT\_REACHED ();  } |

1.thread\_current ()->status = THREAD\_DYING;

현재 실행 상태를 running에서 terminated로 변경합니다.

|  |
| --- |
| 4-7  static void schedule (void)  {  struct thread \*cur = running\_thread ();  struct thread \*next = next\_thread\_to\_run ();  struct thread \*prev = NULL;  ASSERT (intr\_get\_level () == INTR\_OFF);  ASSERT (cur->status != THREAD\_RUNNING);  ASSERT (is\_thread (next));  if (cur != next)  prev = switch\_threads (cur, next);  thread\_schedule\_tail (prev);  } |

1.struct thread \*cur = running\_thread ();

running\_thread의 tcb

2.struct thread \*next = next\_thread\_to\_run ();

다음 실행할 쓰레드를 선택합니다.

3.ASSERT (is\_thread (next));

유효한 스레드인지 확인합니다.

if (cur != next)

prev = switch\_threads (cur, next);

현재 스레드와 다음 실행할 스레드가 다를 경우에만 문맥교환합니다. 따라서 prev에는 cur의 tcb포인터가 들어가게됩니다.

thread\_schedule\_tail (prev);

종료된 쓰레드는 할당해제 하고 타임 슬라이스은 0으로 돌립니다.

|  |
| --- |
| 4-8 switch\_threads  .globl switch\_threads  .func switch\_threads  switch\_threads:    pushl %ebx  pushl %ebp  pushl %esi  pushl %edi    .globl thread\_stack\_ofs  mov thread\_stack\_ofs, %edx  movl SWITCH\_CUR(%esp), %eax  movl %esp, (%eax,%edx,1)  movl SWITCH\_NEXT(%esp), %ecx  movl (%ecx,%edx,1), %esp  popl %edi  popl %esi  popl %ebp  popl %ebx  ret  .endfunc |

1. pushl %ebx

pushl %ebp

pushl %esi

pushl %edi

stack에 push(running 스레드가 문맥을 저장합니다.)

1. movl SWITCH\_CUR(%esp), %eax

movl %esp, (%eax,%edx,1)

movl SWITCH\_NEXT(%esp), %ecx

movl (%ecx,%edx,1), %esp

cur->next로 실행의 전환이 발생합니다.

3. popl %edi

popl %esi

popl %ebp

popl %ebx

next의 문맥이 복원됩니다.

1. ret : eax값을 반환합니다.

따라서 cur의 시작주소를 반환합니다.

|  |
| --- |
| 4-9 void thread\_schedule\_tail (struct thread \*prev)  {  struct thread \*cur = running\_thread ();  ASSERT (intr\_get\_level () == INTR\_OFF);  cur->status = THREAD\_RUNNING;  thread\_ticks = 0;  #ifdef USERPROG  process\_activate ();  #endif  if (prev != NULL && prev->status == THREAD\_DYING && prev != initial\_thread)  {  ASSERT (prev != cur);  palloc\_free\_page (prev);  }  } |

1.thread\_ticks = 0;

thread\_ticks를 4->0으로 만들어서 새로운 time slice를 시작합니다.

2. palloc\_free\_page (prev);

현재 스레드 상태가 종료상태라면 동적할당된 4kb 스택이자 tcb를 할당 해제합니다.

|  |
| --- |
| 4-10  .globl switch\_entry  .func switch\_entry  switch\_entry:  addl $8, %esp  pushl %eax  .globl thread\_schedule\_tail  call thread\_schedule\_tail  addl $4, %esp  ret  .endfunc |

1. addl $8, %esp

스택의 esp를 cur->eip로 옮김

cur와 next가 stack에서 제거되는 효과가 발생됩니다.

1. ret

kernel\_thread 함수의 시작주소를 반환합니다.

|  |
| --- |
| 4-11 static void kernel\_thread (thread\_func \*function, void \*aux)  {  ASSERT (function != NULL);  intr\_enable ();  function (aux);  thread\_exit ();  } |

1.intr\_enable ();

인터럽트 시작, 타이머를 가동하여 40ms까지 진행합니다.

2.function (aux);

스레드 함수 실행

3.thread\_exit ();

함수가 return을 시행하면 스레드를 종료시킵니다.

부록

|  |
| --- |
| int64\_t  timer\_elapsed (int64\_t then)  {  return timer\_ticks () - then;  } |

타임을 체크 하는 함수

|  |
| --- |
| void  timer\_sleep (int64\_t ticks)  {  int64\_t start = timer\_ticks ();  ASSERT (intr\_get\_level () == INTR\_ON);  while (timer\_elapsed (start) < ticks)  thread\_yield ();  } |

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

인터럽트가 켜져 있는지 확인 합니다.

2.while (timer\_elapsed (start) < ticks)

thread\_yield ();

time slice가 timer\_elapsed(start)보다 크면 running->ready로 변경합니다.

////테스트 프로그램 수행 함수

|  |
| --- |
| static void  run\_actions (char \*\*argv)  {  struct action  {  char \*name; /\* Action name. \*/  int argc; /\* # of args, including action name. \*/  void (\*function) (char \*\*argv); /\* Function to execute action. \*/  };  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;  for (a = actions; ; a++)  if (a->name == NULL)  PANIC ("unknown action `%s' (use -h for help)", \*argv);  else if (!strcmp (\*argv, a->name))  break;  for (i = 1; i < a->argc; i++)  if (argv[i] == NULL)  PANIC ("action: `%s' requires %d argument(s)", \*argv, a->argc - 1);    a->function (argv);  argv += a->argc;  }  } |

(1).a->function (argv);

argv += a->argc;

테스트 할 함수를 실행시킵니다.

(2)자료구조 분석

|  |
| --- |
| struct thread  {  tid\_t tid;  enum thread\_status status;  char name[16];  uint8\_t \*stack;  int priority;  struct list\_elem elem;  #ifdef USERPROG  uint32\_t \*pagedir;  #endif  unsigned magic;  }; |

1. .tid

Thread의 고유번호를 저장한다.

1. .status

Thread

Thread의 현재 상태 정보를 저장하는 곳

실행 중인 thread는 THREAD\_RUNNING, cpu할당을 기다리는 thread는 THREAD\_READY, 수행되고 나서 종료되는 Thread는 THREAD\_DYING, I/O 또는 event를 기다리는 thread는 THREAD\_BLOCKED

1. .name

Thread의 이름을 저장한다

1. .stack

Thread가 사용하는 스택의 포인터를 저장합니다.

1. .priority

Thread의 우선순위를 저장한다. PRI\_MIN:0~PRI\_MAX:63

기본값은 PRI\_DEFAULT는 32, 숫자가 높을수록 높은 우선순위를 갖습니다.

1. .elem

Thread의 구조체에 리스트를 구성하기 위해 이전 thread 구조체와 다음 thread 구조체의 포인터를 저장합니다.

1. .magic

스택오버플로우를 탐지하는 기능을 합니다.

스크린샷이(가) 표시된 사진

매우 높은 신뢰도로 생성된 설명

1. .bss

초기화 되어 있지 않은 전역변수를 저장합니다.

1. .data

초기화된 전역변수를 저장합니다.

1. .rodata

kernel source code 중 읽기전용 data를 저장합니다.

1. .text

kernel source code 저장

밑에 1MB만큼 띄우고

LOADER\_PHYS\_BASE(0xc0000000)에 main함수를 위치시킵니다.

스크린샷이(가) 표시된 사진

매우 높은 신뢰도로 생성된 설명

1. User pool, kernel pool:

동적 메모리 할당시 사용되는 저장공간 입니다.(4KB단위)

함수 palloc\_get\_page()에 의해서 할당됩니다.

지도, 텍스트이(가) 표시된 사진

매우 높은 신뢰도로 생성된 설명

next: next’s stack의 시작주소를 저장합니다.

cur: cur’s stack의 시작주소를 저장합니다.

stack: 현재 struct thread, stack의 상대적위치(시작점~stack구조체 필드)

(3)프로그램 실행 경로 분석

1. 스레드 초기화 경로

main()->thread\_init()

1. 메모리 초기화 경로

palloc\_init()->paging\_init()

1. 인터럽트 핸들링 초기화

intr\_init()->timer\_init(), kbd\_init()

1. Thread 스케줄링 시작

Thread\_start()

Timer 가동, 멀티 프로그래밍 시작

///부팅 끝

1. Run\_actions(argv);

테스트 프로그램시작

1. 스레드 초기화

Kernel 자기자신을 thread로 만드는 작업입니다.

따라서 4KB tcb(thread control block)이 생성되게 됩니다.

1. 메모리 초기화

palloc\_init()로 동적 메모리 할당 영역을 초기화 하고

palloc\_get\_page()함수를 통해 동적 메모리 공간을 스택 내에 할당 합니다.

인텔cpu에서는 응용프로세스마다 4GB가상메모리 할당

3GB/user가 사용하고 1GB는 kernel이 사용합니다.

paging\_init()에 의해서 page dir와 page table을 생성하고 index부여합니다.

1. Interrupt handling 초기화
2. 인텔 cpu에 의해서 h/w적 스레드 stack에 push가 일어난다.
3. push $0x##NUMBER; intrNN\_stub()에 의해 일어나는 스레드의 stack에 push
4. pushl %esp esp레지스터 값을 stack에 push
5. .globl intr\_handler

call intr\_handler

intr\_handler 호출하여 통합 인터럽트 처리

|  |
| --- |
| .func intr\_entry  intr\_entry:  pushl %ds  pushl %es  pushl %fs  pushl %gs  pushal  cld  mov $SEL\_KDSEG, %eax  mov %eax, %ds  mov %eax, %es  leal 56(%esp), %ebp    pushl %esp  .globl intr\_handler  call intr\_handler  addl $4, %esp  .endfunc |

|  |
| --- |
| .globl intr\_exit  .func intr\_exit  intr\_exit:    popal  popl %gs  popl %fs  popl %es  popl %ds      addl $12, %esp    iret  .endfunc |

|  |
| --- |
| .data  .globl intr\_stubs  intr\_stubs:  #define zero  pushl %ebp;  pushl $0  #define REAL  pushl (%esp);  movl %ebp, 4(%esp)  #define STUB(NUMBER, TYPE)  .text;  .func intr##NUMBER##\_stub;  intr##NUMBER##\_stub:  TYPE;  push $0x##NUMBER;  jmp intr\_entry;  .endfunc; |

4)Thread scheduler 시작

thread\_start()에 의해서 thread\_create()와 intr\_enable() 스레드 생성과 interrupt시작

thread\_yield(): running->ready 상태전이

thread\_block(): running->waiting 상태전이

thread\_exit(): running->terminated 상태전이

결국 thread\_yield(),thread\_block(),thread\_exit()에 의해서 스레드 문맥교환이 일어납니다.

1. Running test

pintos -v –-run alarm-multiple에 의해서 1~4의 수행과정을 alarm기능을 통해 테스트 해본다.

*소감문: 기존의 수업시간에 배웠던 핀토스 소스코드를 분석과제를 통해 다시 한번 소스코드에 대한 내용을 정리하고 머리속에 넣음으로서 전체적인 핀토스 내부 코드에 대한 이해가 정리되었고 앞으로의 과제 수행내용에 대해 도움이 될 것 같습니다.*

*특히 수업시간에 다뤘던 함수 이외의 함수들의 코드를 분석하며 프로그램의 경로 분석 및 메모리할당에 대해 조금 더 깊게 이해 할 수 있었습니다. 그리고 하나하나 문장들을 분석해보면서*

*앞으로의 소스코드과제들을 어떻게 수행해야 할지에 대한 방향도 어느 정도 설정 할 수 있게 되었습니다.*

*항상 좋은 강의를 진행해주시는 손성훈 교수님 고맙습니다.*