2019320097_조이강

Q1. timer_sleep() function

- Q1-1) Describe what the “timer_tick()” does.
  - timer_ticks()는 64비트 정수인 ticks을 반환하는 함수입니다.
  - ticks는 **static int64_t** ticks; 이 코드로 전역 변수로 선언되어있으므로, critical section(cs) 데이터가 됩니다.
  - 따라서 ticks에 접근하기 전에 **enum** intr_level old_level **=** intr_disable (); 에서 인터럽트를 비활성화 합니다.
  - **int64_t** t **=** ticks; 를 통해 t에 ticks 값을 저장합니다.
  - intr_set_level (old_level); 더 이상 ticks에 접근하지 않으므로 인터럽트를 다시 활성화 합니다.
  - **return** t; 최종적으로 2번째 라인을 실행했던 시점의 ticks를 t로 반환합니다.
- Q1-2) Describe how often interrupts occur in Pintos and which functions are called due to the interrupt.
  - timer.h 파일에 **#define** TIMER_FREQ 100로 1 초에 100번의 빈도로 인터럽트가 호출되게끔 정의 되어있습니다.
  - timer_interrupt() 함수가 인터럽트를 담당하고 있습니다.
  - timer_interrupt()에서 ticks의 값을 증가시키고, thread_tick() 함수를 호출합니다.
  - 이 호출은 1초에 100번의 빈도로 일어나게 됩니다.
- Q1-3) Describe how the ”ticks” variable changes when an interrupt occurs.
  - ticks는 timer_interrupt() 함수에 의해 변화합니다.
  - timer_interrupt() 함수는 한 번 호출될 때 마다 ticks++; 를 통해 ticks 값을 1 증가시킵니다.
  - timer_interrupt() 함수는 1초에 100회 호출되므로 ticks도 1초에 100씩 증가하게 됩니다.

- Q1-4) Describe what the ”timer elapsed()” does.
  - ticks는 os가 부팅된 시점부터 1초에 100씩 증가합니다.
  - timer_elapsed()는 int64_t then이라는 변수를 통해 then 변수의 시점부터 얼마나 많은 틱이 경과했는지를 반환합니다.
  - timer_ticks () 이 라인을 통해 현재 시점에 ticks의 값을 timer_ticks()에게서 받아옵니다.
  - 받아온 현재의 ticks에 then을 뺌으로써 then 시점에서 얼마나 ticks가 경과했는지를 반환합니다.
- Q1-5) Describe what the ”timer sleep()” does in detail.
  - timer_sleep()은 함수를 호출한 스레드가 특정 시간동안 CPU를 계속해서 포기하게끔 하는 함수입니다.
  - 만약 인터럽트가 비활성화되어있다면 스레드가 cs에 접근한 상태이므로 스레드는 sleep할 수 없습니다.
  - ASSERT (intr_get_level () **==** INTR_ON); 이 코드를 통해 인터럽트가 활성화 되어있는지 검사합니다.
  - 포기하게 하는 시간을 ticks를 통해 입력받습니다.
  - **int64_t** start **=** timer_ticks (); 이 코드를 통해 함수를 호출한 시점에서의 ticks 값을 start에 저장합니다.
  - **while** (timer_elapsed (start) **<** ticks) , timer_elapse(start)함수 호출을 통해서 start 시점부터 현재까지 얼마나 많은 ticks가 경과했는지를 지속적으로 검사하고, 경과 시간을 입력받은 ticks 값과 비교합니다.
  - 아직 시간이 입력받은 ticks만큼 경과하지 않았다면, thread_yield()함수를 계속해서 호출하여 스레드가 cpu를 계속해서 포기하게 만듭니다.

Q2) thread yield() function

- Q2-1) Describe each member field in the ”struct thread”.
  - **typedef int** tid_t로 정의되어있는 tid를 선언합니다. tid는 각 스레드를 구분하기 위한 id입니다.
  - enum thread_status 열거식 자료인 status를 선언합니다.

- thread_status는 THREAD_RUNNING, THREAD_READY, THREAD_BLOCKED, THREAD_DYING의 네 가지 상태를 가집니다.
- 총 16글자의 char name[16]을 디버깅 목적으로 가지게됩니다.
- 스레드의 스택 위치를 가리키는 uint8_t *stack을 가집니다.
- 스레드의 우선순위인 int priority를 가집니다
- 우선순위는 PRI_MIN, PRI_DEFAULT, PRI_MAX 세 가지로, 각각 0, 31, 63의 값을 가지고 있습니다.
- 존재하는 모든 스레드를 위한 리스트인 **struct** list_elem allelem을 가집니다
- thread_yield() 같은 함수에서 준비 대기열과 같은 다른 목록에 추가하기 위한 **struct** list_elem elem 리스트를 가집니다.
- USERPROG가 정의되었을 때만 페이지의 주소인 **uint32_t** *pagedir 을 가집니다.
- 스택 오버플로우를 감지하기 위한 unsigned magic을 가집니다.
- 만약 스택 오버플로우가 발생하면 magic의 값이 변화하므로 이를 감지할 수 있습니다.

- Q2-2) Describe what the ”running thread()” does.
  - running_thread(0는 지금 cpu에서 실행 중인 스레드가 어떤 스레드인지를 파악하여 반환하는 함수입니다.
  - **uint32_t** *esp 현재 스레드의 메모리 주소를 파악하기 위한 esp 포인터를 선언합니다.
  - **asm** ("mov %%esp, %0" : "=g" (esp)) 어셈블리어로 현재 cpu의 스택 포인터 값을 esp에 할당합니다.
  - 스레드 구조체는 페이지의 첫 부분에 위치하고, 스택 포인트는 페이지의 중간 어딘가에 위치하고 있으므로 스택 포인터의 값을 반올림하면 현재 실행 중인 스레드의 주소가 나오게 됩니다.
  - **return** pg_round_down (esp) esp의 주소를 반올림하여 찾아낸 스레드를 반환합니다.
- Q2-3) Describe the four statuses that a thread can have during its life cycle.
  - enum thread_status에 총 4가지의 스레드의 상태가 정의되어있습니다.
  - THREAD_RUNNING = 실행 중인 상태를 의미합니다.
  - THREAD_READY = 실행 중이 아니며, CPU를 대기하고 있는 상태입니다.

    - THREAD_BLOCKED = 특정한 이벤트를 기다리고 있는 상태입니다
    - THREAD_DYING = 곧 사라지게 될 스레드의 상태입니다.
- Q2-4) Describe what the ”list push back()” does.
    - 또 다른 리스트인 *list를 입력으로 받아, *list의 맨 끝에 *elem을 삽입하는 함수입니다.
    - list_insert (list_end (list), elem)을 호출하여 리스트의 맨 끝에 elem을 삽입합니다.
    - 리스트의 맨 끝 주소를 알아내기 위해 list_end()라는 함수를 호출하여 반환 받은 주소로 삽입을 진행합니다.
- Q2-5) Describe why the ”struct list read list” variable is needed for scheduling.
    - ready_list 에는 현재 실행 중이 아니지만 대기 상태인 스레드들이 저장되어 있습니다.
    - thread_yield() 나 thread_unblock()된 스레드들이 이 ready_list에 저장되게 됩니다.
    - next_thread_to_run() 함수에서 ready_list의 스레드가 앞에서 부터 실행되기 시작합니다.
    - 따라서 ready_list는 실행이 끝나 대기하는 스레드들을 모아 다음 실행할 스레드를 뽑는 스케줄링에 있어 필수적이라고 할 수 있습니다.
- Q2-6) Describe what the ”thread yield()” does in detail.
    - thread_yield() 함수는 현재 실행 중인 스레드가 cpu를 포기하게 만들고, 스레드를 대기 스레드 리스트에 넣습니다.
    - **struct** thread ***cur = thread_current ();** 라인은 현재 실행 중인 스레드의 주소를 cur에 할당합니다.
    - ASSERT (**!**intr_context ()); 이 코드를 통해 외부 인터럽트를 처리하고 있는지를 검사합니다. 인터럽트를 처리하고 있다면 오류가 발생합니다.
    - old_level **=** intr_disable (); 이 코드를 통해 스레드를 대기 리스트에 넣기 전 인터럽트를 비활성화 합니다.
    - 실행 중인 스레드가 idle_thread가 아니라면 list_push_back (**&**ready_list, **&**cur→elem); 이 코드를 통해 현재 스레드를 대기 스레드들의 리스트 마지막에 추가합니다.

- cur→status = THREAD_READY; 스레드의 상태를 THREAD_RUNNING에서 THREAD_READY로 바꿉니다.
- schedule (); 인터럽트가 비활성화 되어있고, 현재 cpu에 실행 중인 스레드가 없으므로 새롭게 실행할 스레드를 스케줄링합니다.
- intr_set_level (old_level); 이 코드를 통해 다시 인터럽트를 이전 상태로 되돌립니다.

Q3) schedule() function

- Q3-1) Describe what the ”next thread to run()” and ”list entry()” does, respectively.
  - next_thread_to_run() 함수는 다음 실행할 스레드를 반환하는 역할을 하고 있습니다.
  - **if** (list_empty (**&**ready_list)) 만약 대기 상태의 스레드가 아무것도 없다면
  - **return** idle_thread; idle_thread가 실행될 수 있도록 반환합니다.
  - **else return** list_entry (list_pop_front (**&**ready_list), **struct** thread, elem); 그렇지 않다면 대기 중 리스트에서 가장 앞에 있는 스레드를 반환합니다.
  - list_entry() 함수는 list_pop_front() 함수로 뽑은 elem 리스트 주소를 그 주소가 포함되어있는 구조체인 thread를 가리키도록 변환해주는 역할을 맡습니다.
- Q3-2) Describe what the ”switch threads()” does.
  - switch_threads()는 현재 스레드 cur과 다음 실행될 스레드 next 사이에서 컨텍스트 스위칭을 해주는 역할을 맡고 있습니다.
  - 스위칭을 하는 함수는 switch.S에 구현되어 있으며, 호출한 스레드(cur)의 레지스터 상태와 스택 포인터를 기록하고, 다음 스레드(next)의 상태를 불러옵니다.
  - prev = switch_threads (cur, next); cur 에서 next로의 컨텍스트 스위칭이 일어나게 됩니다. 그 후, prev에 cur을 저장하게 됩니다.
  - thread_schedule_tail (prev); 이 코드를 통해 컨텍스트 스위칭을 완료하고, 만약 prev 가 THREAD_DYING 상태이면 없앱니다.
- Q3-3) Describe what the ”switch schedule tail()” does.
  - thread_schedule_tail()은 스케줄링을 마무리 짓는 역할을 하고 있습니다.
  - 컨텍스트 스위칭 자체는 switch_threads()에서 완료되어 새로운 스레드가 실행되고 있으나, 이전 스레드의 처리와 새로운 스레드의 상태 등을 기록하는 역할을 합니

다.

- thread_schedule_tail()은 이전 스레드인 prev를 입력받고, **struct** thread ***cur = running_thread (); 이 코드를 통해 현재 실행 중인 스레드를 cur에 저장합니다.
- cur→status = THREAD_RUNNING; 실행 중인 스레드 cur의 상태를 RUNNING으로 바꿔줍니다.
- thread_ticks = 0; 스레드가 특정한 time slice 동안 실행될 수 있게 thread_tick를 초기화하여 시간을 기록합니다.
- **if** (prev **!= NULL &&** prev→status **==** THREAD_DYING **&&** prev **!=** initial_thread), initial thread를 제외한 스레드가 DYING상태이고, ASSERT (prev **!=** cur); 현재 스레드와 과거 스레드의 주소가 같지 않는 경우
- palloc_free_page (prev); 과거 스레드의 메모리를 free시켜 스레드를 제거합니다.

- Q3-4) Describe what the ”schedule()” does in detail.
  - schedule() 함수는 스케줄링, 현재 스레드에서 다음 스레드로 전환하는 작업을 합니다.
  - **struct** thread *cur = running_thread (); cur에는 현재 실행 중인 스레드를 저장합니다.
  - **struct** thread *next = next_thread_to_run (); next에 다음 실행될 스레드를 불러옵니다.
  - **struct** thread *prev = **NULL**; 스케줄링 후에 이전 스레드를 저장할 prev를 선언합니다.
  - ASSERT (intr_get_level () == INTR_OFF); 로 인터럽트가 비활성화되어있음을 확인합니다.
  - ASSERT (cur→status **!=** THREAD_RUNNING); 현재 스레드 cur이 실행중이 아님 확인합니다.
  - ASSERT (is_thread (next)); next에 쓰레기 값이 아닌 실제 스레드의 주소가 저장되어있는지 확인합니다.
  - **if** (cur **!=** next) 만약 현재 스레드와 다음 스레드가 동일하지 않다면
  - prev = switch_threads (cur, next); switch_threads를 이용하여 컨텍스트를 스위칭하고, 과거 스레드인 cur을 prev에 저장합니다.
  - thread_schedule_tail (prev); prev의 상태에 따른 작업을 수행하고, 스케줄링을 마무리 합니다.

- 최종적으로 cur → next로 스레드의 전환이 이루어지며, next의 상태가 RUNNING이 되고 주어진 time_slice 동안 실행됩니다.
- prev는 DYING 상태일 경우 제거되고, 다른 경우에는 READY, BLOCKED 상태가 됩니다.

Q4) thread block() & thread unblock() function

- Q4-1) Describe what the ″thread current()″ does.
  - thread_current()함수는 running_thread()와 비슷하게 현재 실행 중인 스레드를 반환하지만, 스레드가 온전한지에 대한 검사를 진행합니다.
  - **struct** thread ***t =** running_thread (); 이 코드로 t 에 현재 실행 중인 스레드의 주소를 할당합니다.
  - t에 저장되어있는 주소가 정상적인 스레드인지 확인합니다.
  - ASSERT (is_thread (t)); 이 코드는 t가 스레드 구조체임을 확인합니다.
  - ASSERT (t→status **==** THREAD_RUNNING); 이 코드는 스레드가 실제로 실행 중인지를 검사합니다.
  - **return** t; 모든 검사를 마친 확실히 실행 중인 스레드를 반환합니다.
- Q4-2) Describe what the ″is thread()″ does.
  - is_thread() 함수는 입력받은 스레드 *t가 정상적인 스레드인지를 검사해 T/F로 반환하는 함수입니다.
  - 먼저 t 가 NULL이 아닌지 검사합니다.
  - t→magic 값이 초기 값인 THREAD_MAGIC에서 변화하지 않았는지 검사합니다. 만약 변화했다면, 스택 오버플로우가 발생했음을 알 수 있습니다.
  - **return** t **!= NULL &&** t→magic **==** THREAD_MAGIC; 최종적으로 정상 스레드면 True를, 그렇지 않은 경우 False를 반환합니다.
- Q4-3) Describe what the ″thread block()″ and ″thread unblock()″ does in detail.
  - thread_block()은 호출한 스레드를 sleep하게 만듭니다. 이 thread_unblock() 함수로 깨워지기 전까지는 스케줄링 되지 않습니다.

- 이 과정은 인터럽트가 비활성화된 상태에서 진행되어야합니다.
- ASSERT (**!**intr_context ()); , ASSERT (intr_get_level () **==** INTR_OFF); 이 두 코드로 외부 인터럽트를 처리하지 않고, 인터럽트과 비활성화되어있음을 확인합니다.
- thread_current ()→status **=** THREAD_BLOCKED; 호출한 스레드의 상태를 BLOCKED로 바꿉니다.
- schedule (); 그 다음 스레드로 전환하기 위해 schedule()을 호출합니다.
- thread_unblock()은 BLOCKED된 입력 스레드 t 를 READY 상태로 만들고, 스케줄링을 위해 ready_list에 추가하는 작업을 진행합니다.
- ASSERT (is_thread (t)); 이 코드로 t가 정상 스레드임을 검사합니다.
- old_level **=** intr_disable (); 인터럽트를 비활성화 합니다.
- ASSERT (t→status **==** THREAD_BLOCKED); t의 상태가 BLOCKED인지를 검사합니다.
- list_push_back (**&**ready_list, **&**t→elem); ready_list의 가장 뒷 부분에 스레드 t의 elem을 추가합니다.
- t→status **=** THREAD_READY; t의 상태를 READY로 만듭니다.
- intr_set_level (old_level); 인터럽트를 이전 상태로 되돌립니다.

Q5) init thread() function and getting/setting priority of a thread

- Q5-1) Describe a range of priority a thread can have and the value of ”PRI MIN”, ”PRI MAX”, and ”PRI DEFAULT”.
  - 스레드의 우선순위(priority)는 thread.h에 총 세 가지로 정의되어 있습니다.
  - **#define** PRI_MIN 0 / 가장 낮은 우선순위 는 PRI_MIN이고 0을 가집니다.
  - **#define** PRI_DEFAULT 31 / 기본 우선순위는 PRI_DEFAULT고 31을 가집니다.
  - **#define** PRI_MAX 63 / 가장 높은 우선순위는 PRI_MAX고 63을 가집니다.
  - 따라서 스레드는 0 부터 63까지의 우선순위 값을 가질 수 있습니다.
- Q5-2) Describe what the ”init thtread()” does in detail.
  - init_thread()는 create_thread()에 의해 호출되어 입력 받은 t에 메모리를 할당하며, BLOCKED 상태로 만들고, 우선순위를 부여하는 등 스레드를 초기화하는 역할

을 합니다.

- init_thread (**struct** thread ***t, const char ***name, **int** priority) / init_thread는 새로 만들어진 스레드 t 와 이름 name, 그리고 우선순위 priority를 입력받습니다.
- ASSERT (t **!= NULL**); 스레드가 정상적인 스레드인지 확인합니다.
- ASSERT (PRI_MIN **<=** priority **&&** priority **<=** PRI_MAX); 스레드에 입력될 우선순위 값이 정상 범위에 있는지 확인합니다.
- ASSERT (name **!= NULL**); 스레드에 입력될 이름이 정상인지 확인합니다.
- memset (t, 0, **sizeof** *t); 스레드의 모든 바이트를 0으로 설정합니다.
- t→status **=** THREAD_BLOCKED; 새 스레드의 상태를 BLOCKED로 설정합니다.
- strlcpy (t→name, name, **sizeof** t→name); 스레드의 이름을 입력받은 name으로 설정합니다.
- t→stack **=** (**uint8_t** *) t **+** PGSIZE; 스레드의 스택 포인터 위치를 t + PGSIZE로 설정합니다.
- t→priority **=** priority; 스레드의 우선순위를 입력받은 priority 값으로 설정합니다.
- t→magic **=** THREAD_MAGIC; 스레드에서 스택 오버플로우를 감지하는 THREAD_MAGIC을 기본 값으로 설정합니다.
- list_push_back (**&**all_list, **&**t→allelem); 새롭게 초기화된 스레드를 모든 스레드들의 리스트인 all_list에 추가합니다.
- 이 과정을 통해 새로운 스레드가 초기화됩니다.

- Q5-3) Describe what the ”thread get priority()” and ”thread set priority” does
  - thread_get_priority() 함수는 현재 실행 중인 스레드의 우선순위를 반환합니다.
  - **return** thread_current ()→priority; thread_current()를 통해 현재 스레드를 찾고, 우선순위를 반환합니다.
  - thread_set_priority() 함수는 현재 실행 중인 스레드의 우선순위를 입력받은 new_priority의 값으로 변경합니다.
  - thread_current ()→priority **=** new_priority; thread_current()를 통해 현재 스레드를 찾고, 우선순위를 new_priority로 변경합니다.