Pintos1 Final report

20210084 김지민, 20210216 양준영

1. Alarm Clock

   A. Solution

      i. 기존 timer_sleep 함수에서는 x timer ticks이 지나기도 전에 while문으로 매시간을 확인하며 thread_yield() 함수를 호출하는 것이 문제였다. 그 결과 thread는 계속 ready list에 추가되어 다시 호출되는 busy wait을 하게 되었다. 따라서 thread를 x timer ticks이 지난 이후 ready list에 삽입되도록 함수를 수정해야 하는 것이 관건이다.

      ii. Thread를 ready 상태가 아닌 sleep (즉, block) 상태로 만든 후, 나중에 시간이 지나면 다시 깨우는 (즉, unblock) 형식으로 코드를 수정하였다.

      iii. 현재 thread.c에는 ready_list와 all_list만 존재하고 sleep한 thread를 따로 관리하는 list가 존재하지 않는다. 따라서 sleep 시킨 thread만 모아두는 'time_list'를 새로 정의하였다. 또한 int64_t time_to_wakeup 변수를 thread 구조체에 추가함으로써 thread가 깨어날 시간을 저장한다.

      iv. thread.c에 새로 추가된 함수

         1. bool compare_less_ticks (struct list_elem *s, struct list_elem *t, void* aux UNUSED): s와 t를 각각 요소로 가지는 thread들 간 time_to_wakeup 값을 비교하여 첫 번째 쓰레드가 더 작은 경우에 true를 반환하도록 한다. 이는 time_list를 오름차순으로 배열하도록 하는 함수이다.

         2. void thread_sleep (int64_t ticks): 쓰레드를 재우는 함수로, yield 함수와 기본적인 기능은 거의 동일하다. 다만 현재 쓰레드의 time_to_wakeup 변수를 입력받은 ticks로 저장하여 깰 시간을 저장하는 점, ready_list가 아닌 time_list에 깨어날 시간을 기준으로 오름차순으로 삽입하는 점, 현재 쓰레드의 status를 ready가 아닌 blocked 상태로 변경하는 점이 다르다.

3. void thread_wake (int64_t ticks): time_list를 돌며 time_to_wakeup이 입력 받은 ticks (현재 시간을 입력 받음)을 비교한다. 만약 쓰레드의 time_to_wakeup이 ticks보다 작거나 같다면 깨어날 시간이 되었다는 것이므로 time_list에서 해당 쓰레드 요소를 삭제하고 unblock 시킨다.

v. timer.c 변경사항

1. 기존 timer_sleep 함수에 있던 while문을 삭제한 후, if문으로 'timer_elapsed (start) < ticks'을 검사하여 만약 참이라면 thread_yield 함수가 아닌, thread_sleep 함수를 호출한다. 이때, 인수로 start+ticks를 넣음으로써 thread_sleep 함수에서 time_to_wakeup에 쓰레드가 일어날 시간을 저장할 수 있게끔 한다.

2. 나눠주신 Pintos Project1 문서를 보면 핀토스는 매 ticks마다 timer interrupt를 발생시킨다고 되어있다. Time_list를 수시로 보며 어떤 쓰레드를 깨워야 할 지 체크해야 하므로 timer_interrupt 함수에 thread_wake(ticks) (여기 ticks은 현재 ticks을 의미) 호출을 삽입함으로써 수시로 깨워야 할 쓰레드들을 체크하도록 한다.

B. Discussion

i. 기존 디자인과의 비교: Alarm clock의 경우, 뒤에 나올 Priority Scheduling과 Advanced Scheduling에 비해 비교적 구현이 간단한 편이어서 time_list를 오름차순으로 정렬하기 위한 compare 함수를 따로 만든 점 외에는 대부분 계획한대로 코드를 수정할 수 있었다.

ii. 어려웠던 점: Test를 계속 돌렸는데 계속 time out이 나오기에 알고리즘에 이상이 있는 줄 알고 몇 시간을 검사했었다. 그런데 알고 보니 time_list를 초기화시키지 않아 생긴 문제였다. 이를 계기로 뒤에 나올 기능을 구현할 때에는 새로운 리스트 등을 선언한 뒤에 바로 초기화부터 시켰다. 초기화의 중요성을 깨달았다.

2. Priority Scheduling

A. Solution

i. 우선순위 스케쥴링 구현은 크게 1) 기본 우선순위 스케쥴링 구현과 2)

Priority donation (우선순위 양도) 구현 이렇게 두 가지로 나눠서 생각할 수 있다.

ii. 1) 기본 우선순위 스케쥴링 구현 같은 경우, 기존에는 FIFO 방식으로 ready_list에서 쓰레드를 스케쥴링 했다면 이번엔 우선순위를 기준으로 스케쥴링해야 한다. 이는 lock, semaphore, condition variable를 기다리는 쓰레드들 리스트에서도 마찬가지다. 우선순위가 높은 쓰레드가 먼저 unblock되어 먼저 선점할 수 있도록 해야 한다. 그리고 만약 ready_list에 현재 쓰레드보다 우선순위가 높은 쓰레드가 추가되면 그 즉시 그 쓰레드에게 yield해야 한다.

iii. 2) Priority donation (우선순위 양도) 구현 같은 경우, Priority Inversion을 해결하기 위한 방법으로, multiple donation과 nested donation 두 가지를 고려해야 한다.

iv. thread 구조체 요소 추가: priority donation을 받은 쓰레드는 나중에 원래 자기의 우선순위로 돌아와야 한다. 현재 쓰레드 구조체에는 priority 변수가 하나밖에 없으므로 'init_prior' 변수를 추가해줌으로써 본래 우선순위를 저장하도록 한다. 또한 multiple donation의 경우 한 쓰레드는 여러 쓰레드로부터 우선순위를 양도받아야 할 수도 있다. 그러므로 'prior_donation' 리스트를 추가하여 자기에게 우선순위를 양도해준 쓰레드들을 저장한다. Wait_lock 변수는 현재 자신이 무슨 lock을 기다리고 있는지 저장하며, priority donation 구현에 필요하다. 마지막으로 'elem_d'는 'prior_donation' 리스트에 삽입되는 요소다.

v. thread.c에 새로 추가된 함수

1. bool compare_less (struct list_elem *s, struct list_elem *t, void* aux UNUSED): s와 t를 각각 요소로 가지는 thread들 간 priority 값을 비교하여 첫 번째 쓰레드가 더 큰 경우에 true를 반환하도록 한다. 이는 ready_list를 내림차순으로 배열하도록 하는 함수이다.

2. void check_prior(void): 이는 현재 실행 중인 쓰레드와 ready list에 있는 쓰레드의 우선순위를 비교하여 만약 ready_list의 맨 앞에 있는 쓰레드의 우선순위가 현재 쓰레드의 우선순위보다 크면 thread_yield 함수를 호출하도록 하는 함수이다. ready_list는 priority를 기준으로 내림차순 정렬이 되어있으므로 모든 요소가 아닌 맨

처음 요소와만 비교하면 된다.

vi. thread.c 변경사항

1. 우선 init_thread 함수에서 새로 선언한 init_prior, prior_donation, wait_lock 변수를 초기화한다.

2. 현재 thread_unblock과 thread_yield 함수를 보면 list_push_back 함수를 이용하여 ready_list 끝에서부터 쓰레드를 삽입하는 것을 알 수 있다. 우선순위를 기준으로 내림차순 정렬이 될 수 있도록 이를 'list_insert_orderd'로 변경한다. 이때, list_insert_orderd 함수 인자로 새로 정의한 'compare_less' 함수를 넘겨줌으로써 해당 쓰레드를 ready_list에 우선순위를 기준으로 내림차순 삽입하도록 한다.

3. 기존 thread_create 함수에서는 thread_unblock 함수를 호출함으로써 새로 생성된 쓰레드를 ready_list에 삽입했다. 이때, 새로 ready_list에 삽입된 쓰레드가 현재 쓰레드의 우선 순위보다 높다면 그 즉시 thread_yield 함수를 호출해야 하므로, 해당 기능이 구현된 'check_prior' 함수를 thread_unblock 함수 뒤에 호출한다.

4. 또한 thread는 어느 때에나 thread_set_priority 함수를 통해 우선순위를 변경할 수 있다. 이때에도 3.과 마찬가지로 우선순위 비교를 해서 thread_yield 함수를 호출할지 말지를 결정해야 한다. 따라서 이 함수에서도 check_prior 함수를 호출한다.

   ➔ 이렇게 ready_list에 쓰레드를 삽입되는 부분을 모두 내림차순 정렬이 될 수 있도록 수정함으로써 ready_list는 우선순위 기준 내림차순 상태가 유지된다. 또한 우선순위가 바뀌거나 새로 삽입된 쓰레드의 경우 우선순위 비교 후 작다면 즉시 yield하도록 하는 기능까지 구현되었다.

5. 기존 thread_set_priority 함수에서는 curr->priority를 new_prioritiy로 바꿨었는데, 만약 현재 priority 값이 양도받은 priority값이라면 문제가 발생할 수 있다. 따라서 curr-> init_prior 값을 new_priority 값으로 바꾼 다음, curr의 prior_donation의 맨 앞에 있는 쓰레드와 우선 순위를 비교해본다. 만약 현재 init_prior 값이 prior_donation의 맨 앞에 있는 쓰레드의 우선 순위보다 작다면 curr의 priority 값

을 prior_donation의 맨 앞에 있는 쓰레드의 우선 순위로 바꾸고, 아니라면 priority 값을 init_prior로 한다.

vii. synch.c에 새로 추가된 함수

1. bool compare_less_sema (struct list_elem *, struct list_elem *, void*): 이 함수는 condition 구조체의 waiters 리스트를 내림차순으로 정렬하기 위한 함수이다. condition의 waiters 리스트는 쓰레드가 아닌, semaphore들의 리스트이므로 각 세마포의 waiters 리스트의 맨 앞에 있는 쓰레드 간 우선순위를 비교해야 한다. 각 세마포의 waiters 리스트 또한 우선순위 기준 내림차순으로 정렬된 것이므로 맨 앞에 있는 쓰레드들만 고려하면 된다.

2. bool compare_less_donate (struct list_elem *, struct list_elem *, void*): thread.c에 구현된 compare_less와 기능은 동일하다. 다만, 이 함수는 elem_d를 요소로 가지는 prior_donation 리스트를 정렬할 때 쓰이기 때문에 synch.c에 따로 구현하였다.

viii. synch.c 변경사항

1. 기존 sema_down 함수에서는 sema의 waiter리스트에 뒤에서부터 삽입하는 형식이었다. 이것을 list_insert_orderd 함수로 바꾸어 쓰레드의 우선순위 기준으로 정렬될 수 있도록 한다. 이때, sema의 waiter리스트는 thread의 elem을 요소로 하기 때문에 thread.c에 있는 compare_less 함수를 재사용할 수 있다. 이 함수를 인수로 넣어준다.

2. sema_up 함수에서는 thread_unblock 함수를 호출하고 있는데, unblock 시키기 전에 list_sort 함수를 호출함으로써 sema의 waiters 리스트를 다시 내림차순 정렬해준다. 이는, priority donation 등으로 waiters 리스트 내에 있는 쓰레드들의 우선순위에 변화가 생긴 경우를 대비하기 위한 것이다. 또한 sema_up이 끝나기 전에 check_prior를 호출함으로써 thread_unblock으로 인해 새로 ready_list에 삽입된 쓰레드와 현재 쓰레드의 우선순위를 비교한다.

3. lock_acquire에서는 현재 쓰레드가 요청한 lock의 holder가 null이 아니라면 우선순위를 양도해주는 작업이 필요하다.

list_insert_ordered 함수를 호출하여 lock->holder의 prior_donation 리스트에 현재 쓰레드를 우선순위 내림차순으로 삽입한다. 이때, 이 리스트는 thread 구조체의 elem_d를 요소로 가지므로 compare_less_donate 함수를 인수로 넣어준다. 이 작업을 통해 현재 lock을 소유하고 있는 lock->holder의 prior_donation 리스트에는 현재 이 lock을 기다리고 있는 쓰레드들이 우선순위 대로 정렬되어 있을 것이다. 이는 multiple donation을 위한 것이다. 그리고 현재 쓰레드의 wait_lock을 현재 lock으로 저장한다.

이제 for문을 이용하여 nested donation을 구현한다. 현재 쓰레드의 wait_lock이 null이 아니라면, 그 lock을 소유한 holder와 현재 쓰레드와 우선순위를 비교한다. 만약 현재 우선순위가 더 크다면 holder의 우선순위를 현재 우선순위로 바꿔준다. 그리고 그 holder의 wait_lock이 null이 아니라면 이 작업을 반복한다. 이렇게 holder의 wait_lock을 따라 가면서 우선순위를 양도해주다가 wait_lock이 null이 되는 순간, nested donation이 완료될 것이다.

이후 sema_down를 실행한 뒤에는 lock을 점유하게 되므로 현재 쓰레드의 wait_lock을 null로 만든다.

4. 기존 lock_release 함수에서는 lock->holder를 null로 만든 후 sema_up를 호출하는 것이 끝이었는데, lock_acquire에서 multiple, nested priority donation을 구현했으므로 이에 대한 추가 구현이 필요하다. 우선 해당 lock을 점유한 holder의 prior_donation에서 이 lock을 기다리고 있는 쓰레드들을 삭제해야 불필요한 우선순위 양도를 막을 수 있다. for문을 이용하여 holder의 prior_donation을 돌면서 만약 이 lock을 기다리고 있는 쓰레드가 있다면 이 리스트에서 삭제한다. for문을 다 돌고 난 뒤에는 list_sort 함수를 호출함으로써 이 리스트를 재정렬한다.

   그리고 multiple donation인 경우, 원래 자신이 가지고 있던 priority(init_prior)보다 prior_donation 리스트 맨 앞 쓰레드의 우선순위가 더 크다면 priority를 그것으로 변경해야 한다. 따라서 prior_donation 리스트 맨 앞 쓰레드의 우선순위와 현재 lock->holder의 init_prior의 값을 비교하여 lock->holder의 priority 값을

변경해야 한다. 만약 prior_donation이 비어있다면, 본래 init_prior 값으로 priority를 변경하면 된다.

5. 기존 cond_wait 함수에서는 cond의 waiters 리스트에 끝에서부터 삽입했었다. list_insert_orderd 함수를 이용하여 이 리스트가 내림차순 정렬될 수 있도록 한다. 이때, compare_less_sema 함수를 인수로 넣어줌으로써 리스트 속 세마포들의 waiters 리스트의 맨 앞 쓰레드의 우선순위를 기준으로 정렬될 수 있도록 한다.

6. 마지막으로 cond_signal 함수에서 sema_up 함수를 호출하기 전에 list_sort 함수를 호출함으로써 cond의 waiters 리스트 속 세마포어들을 재정렬한다. 이는 priority donation 등으로 세마포어들의 waiters의 맨 앞 쓰레드가 변경된 경우를 대비하기 위한 것이다.

B. Discussion

i. 기존 디자인과의 비교: 2. Priority Scheduling을 구현하는 과정에서만 list 관련 함수를 호출할 때 필요한 비교함수를 3개 선언했다(thread.c에 1개, synch.c에 2개). 디자인 단계에서는 기능이 비슷하니 막연하게 1~2개 정도면 될 것이라 생각했는데, 실제로 구현해보니 데이터 타입이나 리스트 종류에 따라 각각 선언해줘야 했다.

ii. 어려웠던 점: 새로 선언한 함수들은 대부분 .c 파일의 앞 부분에서 정의하였는데, compare_less_donate 함수 정의를 넣었다하면 테스트에서 오류가 생겼다. 이유를 몰라 계속 헤매다가, 이 함수에 쓰이는 구조체 'semaphore_elem'가 synch.c 파일 중간에 선언되어 있어 그보다 앞에 이 함수를 정의하면 오류가 난다는 사실을 뒤늦게 깨달았다.

또한 donate 부분 test에서 fail이 많이 나오기에 lock_acquire와 lock_release 함수만 열심히 들여다 봤었는데, thread_set_priority 함수에서 우선순위가 바뀌는 부분을 고려하지 않았다는 사실을 깨달았다. 새로운 우선순위를 현재 쓰레드의 priority에 바로 저장하면 양도받은 우선순위가 사라질 수 있으니 init_priority에 우선 저장하고 이후 prior_donation 리스트와 비교하고 priority를 변경하는 방식으로 수정하니 test를 통과할 수 있었다.

3. Advanced Scheduling

A. Solution

i. Thread.c

1. 추가 변수

➔ 스레드 구조체에 nice, recent_cpu를 추가하여, 스레드의 nice, recent_cpu 값을 다룰 수 있도록 하였다. 이를 초기화하기 위해, thread_init에서 초기 스레드의 nice, recent_cpu값을 0으로 초기화하며, init_thread함수에서 기존의 스레드를 복사하므로, 현재 실행 중인 스레드의 nice, recent_cpu의 값을 복사하도록 구현하였다.

➔ Thread.c에서 계산에서 전체적으로 사용될 load_avg를 전역변수로 선언하여 recent_cpu, priority 계산에 사용될 수 있도록 하였다.

2. 기존 함수

➔ Thread_set_nice의 경우, 스레드의 nice를 설정하게 되는데, 이로 인해 thread의 priority값이 변할 수 있으므로, 현재 실행 중인 스레드의 우선순위를 mlfqs_priority_each_evaluate를 통해 재계산하도록 한다. 또한, 우선 순위가 바뀐 후에 우선 순위에 맞게 실행될 수 있도록 thread_yield를 통해 실행을 바꾸도록 구현하였다.

➔ Thread_get_nice의 경우, 스레드의 nice 값을 반환하도록 하였다.

➔ Load_avg, recent_cpu를 반환하는 경우, 현재 값에서 100을 곱해서 반환하도록 하였으며, 해당 값들은 자료형은 int이지만, 실수 계산을 지원하지 않아, int자료형으로 변환한 float형태이므로 float_to_int_to_nearest를 통해 가장 가까운 정수로 변환하여 반환한다

➔ Thread_set_priority 함수의 경우, 스레드의 우선 순위를 설정하지만, mlfqs에선 변경되어선 안 되므로, return으로 함수를 바로 종료하도록 하였다.

3. 추가 함수

➔ 첫 번째 함수로, 1 tick마다 현재 실행 중인 스레드의 recent_cpu를 증가시키는 mlfqs_recent_cpu_increase함수를 구현하였다. 하지만 현재 실행중인 스레드가 idle_thread인 경우 값을 올려선 안되기에 recent_cpu를 올리지 않도록 하였다.

➔ 다음으로, recent_cpu, load_avg, priority 각각을 재계산하는 함수들(mlfqs_recent_cpu_each_evaluate, mlfqs_load_avg_evaluate, mlfqs_priority_each_evaluate)을 구현하였다. 주어진 수식에 따라 값을 계산하도록 하였다. Recent_cpu와 priority의 경우 idle_thread에선 바꿔선 안 되므로, 조건문을 추가하여 값이 바뀌지 않도록 하였다.

➔ Timer.c에서 사용할 recent_cpu, priority를 재계산하는 함수인 mlfqs_all_evaulate함수를 만든다. Recent_cpu와 priority를 재계산하는 함수를 thread_for_each를 통해 전체에 실행시키도록 구현하였다.

ii. Timer.c

1. 변경점

➔ 1tick 마다 mlfqs값을 증가시켜야 하므로, timer_interrupt에서 mlfqs_recent_cpu_increase를 통해 현재 실행 중인 스레드의 recent_cpu값을 증가시킨다

➔ 4틱마다, 스레드의 우선 순위를 재계산하여야 하므로, mlfqs_priority_each_evaluate를 통해 스레드의 우선 순위를 재계산한다.

➔ 1초마다, load_avg, recent_cpu, priortity를 재계산하여야 하므로, tick % TIMER_FREQ==0일 때마다, mlfqs_all_evaluate를 실행시킨다.

iii. Synch.c

1. 변경점

➔ Mlfqs 방식에선 우선 lock_acquire, lock_release에서 일어나던 우선 순위 donation이 일어나면 안 된다. 따라서 mlfqs에서 작

동해야 할 최소 내용만을 조건문을 통해 실행되도록 구현하였다. (sema_down, sema_up과 lock_holder를 세팅하는 부분만이 실행되도록 구현하였다)

B. Discussion

i. 기존 디자인과의 비교: priority가 지정된 범위 밖으로 나갈 수 있었으므로 이를 확인해주는 check_range 함수를 구현하여 해당 값이 범위 안이 되도록 하는 함수를 새로 구현하였다. 이 외에는 priority_scheduling을 구현할 때 우선 순위에 따라 실행되도록 구현되도록 한 것, 우선 순위를 전달하는 부분을 막는 것은 기존 디자인대로 구현되었다. 지속적으로 priority_queue를 사용하지 못하여 우선 순위가 변동되어 정렬되느라 실행에 오랜 시간이 걸리는 것은 계획대로 구현하지 못한 점이다.

ii. 어려웠던 점: thread_get_nice-10 테스트를 실행할 때, thread0와 같이 초반 스레드에 tick이 많이 할당되어 실패하였었다. 이는 thread에서 load_avg, recent_cpu, priority계산의 순서가 잘못되어 발생한 문제로, 우선 순위를 계산하고 값을 바꿔서 우선 순위가 제대로 바뀌지 않아 문제가 발생하였던 것 같다. 이를 찾는데 상당한 시간이 소모되었으며, 이를 해결하면서 thread_for_each를 두 번 돌려 시간이 걸리던 것을 한 번만 돌도록 수정하면서 시간이 약간 단축되었던 것 같다.