Operating System Project #1

Replace Busy Waiting / Priority Schedule / Priority Donation

Team 8
2009-11841 최영진
2011-11749 윤태훈
2013-13436 김시준

목 차

1. Wait Queue Design

Pintos의 원래 구조는 sleep 함수가 시간이 지날 때까지 무한정 loop를 도는 방식으로 구현되어 있다. 이를 busy waiting이라 하는데, 해결하기 위해서 Wait Queue를 구현하였다. 설계 및 구현 사항은 다음과 같다.

- timer.c의 timer_sleep() 함수에서, 계속 loop를 돌면서 시간이 완료되지 않았을 경우 yield하는 부분 대신 새로 추가된 thread_sleep()함수를 호출하도록 변경
- thread.c 에 wait_list 추가, 이는 sleep하고 있는 thread들을 저장하는 역할
- thread.h / thread.c 에 thread_sleep() 함수 추가, sleep한 thread는 block시키고 wait_list에 삽입하되, sleep이 빨리 끝나는 thread가 앞에 오도록
- thread.c next_thread_to_run()의 수정, wait_list에 sleep 완료된 thread가 있을 경우 unblock 호출(ready_list로 가서 다음 스케쥴링을 따르게 됨)

결과적으로 여러 thread가 sleep 시 idle tick을 확인할 수 있었다. 정확한 결과는 5. Testing result of 'alarm-multiple'에 서술할 것이다.

2. Priority Scheduling Design

ready_queue에 thread를 삽입할 때, thread의 priority 순서대로 삽입하여 우선 순위가 높은 thread가 먼저 실행되도록 수정하였다. 이 때 pintos kernel library에서 제공하는 list 관련 함수들을 사용하였는데, list_insert_ordered, list_sort 등에 비교 함수를 넣어줌으로써 순서를 유지할 수 있다. 비교 함수는 thread.c의 priority_more() 이다.

또한 thread_set_priority 함수로 우선 순위가 바뀌거나, 새로운 thread가 생기거나, 실행되던 thread가 terminate 되는 경우 우선 순위를 체크하여 가장 높은 thread가 실행 중이 아닐 경우 즉시 바꿔주도록 해야 한다. 이 역할을 구현한 함수는 thread_check_ready이다. 다만 pintos 구조상 thread가 죽을 경우 자동으로 다음 thread schedule을 진행하므로 terminate 시에는 굳이 호출하지 않았다.

추후 priority donation 등을 진행할 때 확인해보니, 세마포어의 대기 리스트나 Condition Variable의 대기 리스트에서도 우선 순위대로 thread를 깨울 필요성이 있었다. 따라서 synch.c 의 sema / cond 관련 함수들에서도 priority 순으로 삽입하고 정렬하였다.

3. Priority Donation Design

thread의 동기화를 위해 locking mechanism을 사용할 때, 우선 순위가 낮은 thread가 lock을 잡고 있는 경우 이 thread는 실행될 수 없으므로 lock이 풀리지 않는다. 이 문제를 해

결하는 방법 중 하나가 priority donation으로, lock을 기다리는 우선 순위 높은 thread가 lock을 건 thread에게 priority를 'donate' 하게 된다.

기본 개념은 간단하지만 lock이 순차적으로 걸리는 경우, lock이 chain을 형성하는 경우, lock-unlock 중에 priority가 바뀌는 경우 등 신경써야 할 부분이 많다. 8조는 이를 다음과 같은 방식으로 설계하고 구현하였다.

- thread.h 의 thread 구조체에 변수 추가
  - original_priority : donate 전의 원래 우선 순위를 저장
  - waiting_lock : 현재 기다리고 있는 lock 을 저장
  - donator : priority donate를 한 thread들을 갖고 있는 list (이는 여러 thread가 donate했을 때, priority를 순차적으로 계산하기 위해 필요하다.)
    - donator_elem : 위 donator list의 elem
- thread.h / thread.c, synch.c 수정
  - synch.c lock_acquire() : 어떤 thread에서 lock을 요청했는데 아직 걸려있다면, lock과 donate한 thread들을 저장한 뒤 priority donation을 시도한다.
  - synch.c lock_release() : 해당 lock을 기다리는 thread들의 기록을 삭제하고, priority를 원래대로 돌린다.
  - thread.c donate_priority() : lock holder가 더 낮은 priority를 가졌을 경우 현재 thread의 priority로 (임시로) 바꿔준다.
  - thread.c clear_waiting() : 해당 lock을 wait하던 thread들 (이 thread에 donate했을 가능성이 있음)을 삭제한다.
  - thread.c restore_priority() : donate로 바뀌었을 가능성이 있는 priority를 원래대로 돌려준다. 다만 donator list를 확인하여 아직 donate한 thread가 있을 경우 그 최대 priority 로 바꿔준다. 이는 하나의 thread에 대해 각각 다른 여러 개의 donation이 들어왔을 수 있기 때문이다.

이후 priority-donate-lower 테스트가 실패하는 문제가 있었는데, 이는 thread_set_priority() 함수에서 즉시 우선 순위를 설정하지 않고 donator list를 확인함으로써 해결하였다.

4. Implementing Issues

구현 도중 여러 문제가 발생하였는데 그 중 대표적인 이슈와 해결 방법을 정리하면 다음과 같다.

1. alarm-simultaneous test fail

wait_list에서 완료된 thread들을 뺄 때 첫 하나에 대해서만 검사하여 일어난 문제였다. 이 때문에 여러 thread가 한꺼번에 깨지 않고 1, 2tick 차이가 발생하였다. while loop로 조건을 만족하는 모든 thread를 추출하자 해결되었다.

2. priority-sema, priority-condvar test fail

위에 잠시 서술하였으나, priority scheduling만 한 상태로는 세마포어나 컨디션 변수 사용 시 제대로 스케쥴링이 되지 않았다. 이는 lock을 기다리던 thread들 중 priority가 가장 높은 것부터 깨어나야 하는 로직이 없기 때문이었다. 따라서 각 개체들의 wait list 관리 시 priority 순으로 정렬하는 로직을 추가하였다.

이 때, Condition Variable의 경우는 thread가 아니라 thread를 가지는 semaphore_elem 이라는 구조체를 저장하기에, 이에 대한 비교 함수 priority_more_semaphore_elem 을 새로 정의하였다.

3. priority donation 구현 중 page fault

코드 상으로는 문제점 파악이 어려워 gdb를 사용하여 해당 로직까지 쫓아가보았다. 메모리 값들을 찍어 본 결과 사용자 변수에 0x0, 0x8 등 이상한 주소가 들어가있었는데, 원인은 list 구조를 잘못 이해하여 list_elem을 제대로 사용하지 못한 탓이었다. 이를 수정하니 잘 동작하였다.

5. Testing result of ‘alarm-multiple’

1. 구현 전

… 생략 ...

Execution of 'alarm-multiple' complete.

Timer: 867 ticks

Thread: 0 idle ticks, 870 kernel ticks, 0 user ticks

=> idle tick 없이 kernel tick만 존재하며 busy waiting을 하고 있음을 알 수 있다.

2. 구현 후

… 생략 ...

Execution of 'alarm-multiple' complete.

Timer: 869 ticks

Thread: 549 idle ticks, 322 kernel ticks, 0 user ticks

=> idle tick이 생겼으며, busy waiting 문제는 해결되었다.

6. Testing result of ‘alarm-priority’

yeongjin@ubuntu:~/OS/prj0/src/threads$ pintos -- -q run alarm-priority
… 생략 ...
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
Timer: 597 ticks
Thread: 469 idle ticks, 130 kernel ticks, 0 user ticks

=> priority 순으로 깨어나고 실행되었음을 확인할 수 있다.

7. Testing result of All, including ‘priority-donate-*’

yeongjin@ubuntu:~/OS/prj0/src/threads$ make check
cd build && make check
make[1]: Entering directory `/home/yeongjin/OS/prj0/src/threads/build'
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple

pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
All 18 tests passed.

=> pintos의 multi thread에 관련된 alarm, priority 등의 테스트를 모두 성공적으로 통과하였다. 특히 priority-donate-로 시작하는 테스트들은 중첩, 사이클 등 donation에 관련된 여러 상황들을 모델링한 것들이다. 본 프로젝트의 목표인 donation 관련 테스트 역시 통과하였음을 확인할 수 있다.