Blocking, Non-Blocking방식을 활용한 Alarm clock의 개선

배수한

3학년 201910899, 융합전자공학전공 전기전자공학부

201910899@sangmyung.kr

요약

기존 Pintos 커널에서 time_sleep()함수는 정상적으로 작동하지만, Busy waiting방식으로 구현되어 있습니다. 이는 CPU를 낭비하므로 비효율을 초래합니다. 특히 타이머가 긴 시간 동안 대기해야 하는 경우에는 매 타이머 틱마다 체크하므로, CPU를 낭비하고 다른 프로세스나 스레드의 실행을 방해하며, 전력 소비도 증가시킵니다.

이러한 문제를 해결하기 위해 pintos 커널에 있는 timer.c의 timer_sleep()과 timer_interrupt()를 개선하는 방법을 사용합니다. alarm 구조체를 사용한 alarms를 더블 링크드 리스트 형태로 구현하고, 이를 활용하여 코드를 개선할 수 있습니다. 또한 alarm_compare함수를 추가로 구현하여 리스트의 순서를 항상 만료시간 기준 오름차순이 될 수 있도록 만들어줍니다. 이후 pintos에서 제공하는 test들을 통해 개선된 Alarm Clock이 잘 작동되는지 확인합니다.

이와 같은 과정을 통해, Alarm clock을 No Busy-wait형태로 구현할 수 있습니다.

Introduction – 기존 pintos 커널에선 thread_yield함수를 활용한 busy waiting기법을 사용하고 있습니다. 이는 반복문을 사용하여 특정 시간 동안 매 틱마다 체크하는 방식으로, 특정 시간이 경과할 때까지 계속 프로세스를 실행시키는 방법입니다. 이는 CPU를 낭비하고, 전력소비를 증가시키는 문제가 발생합니다.

따라서 기존의 busy waiting문제를 개선하기 위해 리스트를 활용한 blocking, non-blocking방식을 사용합니다. 본 레포트에서는 timer.c의 timer_sleep과 timer_interrupt를 수정하고, alarms리스트와 alarm_compare함수를 구현하여 busy-waiting문제를 해결하는 방법에 대해 자세히 다루고 있습니다.

The proposed method – 수정된 timer_sleep()에서는 지정된 시간이 되면 스레드를 block하고, 이를 alarm_list에서 대기시킵니다. timer_interrupt()에서는 매 틱마다 깨워야 할 스레드가 있는지 확인하고, 만약 깨워야 할 스레드가 있다면 alarms에서 pop 시킨 후 해당 스레드를 unblock 합니다. 이를 위해 alarm구조체와 alarm_list를 구현합니다.

alarm구조체는 알람 만료 시간(expiration)과 알람을 요청한 스레드에 대한 정보(th), 알람 리스트에 대한 연결 필드 정보를 담고 있습니다. 이를 활용하여 alarms라는 이름의 Linked List를 구현합니다. Linked List는 여러 개의 Node로 이루어진 자료구조로, 각 Node는 데이터와 다음 노드를 가리키는 포인터로 이루어져 있습니다. alarms는 timer_init함수에서 타이머와 함께 초기화합니다. timer.c에서 이를 구현하기 위해 수정한 코드는 [그림 1]과 같습니다.

```
static struct list alarms;
struct alarm {
  int64_t expiration;
  struct thread *th;
  struct list_elem elem;
};


void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
  list_init (&alarms);
}
```

[그림 1] timer.c의 수정

수정된 timer_sleep함수는 인자로 ticks 값을 받아 현재 실행 중인 스레드(cur)와 알람 만료 시간(expiration)을 알람 구조체에 할당하고, 이를 alarms에 메단 후에 해당 스레드를 block하는 역할을 합니다.

우선, intr_get_level 함수로 현재 인터럽트 상태가 INTR_ON인지 확인합니다. 만약 비활성화 상태라면 에러를 발생시킵니다. 또한 인자로 받은 ticks 값이 0 이하라면, 함수를 종료합니다. 스레드를 잠들일 필요가 없기 때문입니다. 이후 인터럽트 상태를 INTR_OFF로 변경합니다. 이는 이후 코드에서 alarms에 접근하는 도중 인터럽트가 발생하는 것을 방지하기 위함입니다.

alarm의 크기만큼 동적으로 메모리를 할당하여 new_alarm 포인터에 저장합니다. 이후 알람 만료 시간과 현재 실행중인 스레드를 구조체에 할당합니다. 이때, 알람 만료 시간은 expiration 변수에 현재 시간(start)에 ticks를 더한 값을 할당합니다.

리스트를 항상 만료시간 기준 오름차순으로 정렬하기 위해 list_insert_ordered 함수를 사용하여 alarms에 새로운 알람을 삽입합니다. 이 함수는 연결 리스트의 처음부터 끝까지 순회하면서 alarm_compare 함수를 사용해 new_alarm이 들어갈 위치를 결정한 후 해당 위치에 삽입합니다. 이후 스레드를 block 상태로 만들고, 인터럽트 상태를 이전 상태로 되돌립니다. 코드는 [그림 2]와 같습니다.

```
void
timer_sleep (int64_t ticks)
{
  struct thread *cur = thread_current ();

  ASSERT (intr_get_level () == INTR_ON);

  if (ticks <= 0)
    return;

  int64_t start = timer_ticks ();

  enum intr_level old_level = intr_disable ();

  struct alarm *new_alarm = malloc (sizeof (struct alarm));
  new_alarm->expiration = start + ticks;
  new_alarm->th = cur;
  list_insert_ordered (&alarms, &new_alarm->elem, alarm_compare, NULL);

  thread_block ();

  intr_set_level (old_level);
}
```

[그림 2] timer_sleep함수 수정

list_insert_ordered 함수에서 사용되는 alarm_compare 함수를 구현합니다. 이 함수는 두개의 요소를 비교하여 첫번째 요소가 작으면 ture를 반환하고, 그렇지 않으면 false를 반환합니다. 우선 list_entry 함수를 이용하여 a 구조체의 포인터를 a_comp 포인터 변수에 할당합니다. 따라서 a_comp는 해당 요소를 가리킵니다. 같은 방법으로 b_comp도 할당 후, a_comp와 b_comp의 알람 만료 시간을 비교하고, 이에 따른 값을 반환합니다. 코드는 [그림 3]과 같습니다.

```
static bool
alarm_compare (const struct list_elem *a, const struct list_elem *b)
{
  const struct alarm *a_comp = list_entry (a, struct alarm, elem);
  const struct alarm *b_comp = list_entry (b, struct alarm, elem);

  return a_comp->expiration < b_comp->expiration;
}
```

[그림 3] alarm_compare함수 구현

수정된 timer_interrupt함수는 매 틱마다 깨워야 할 스레드가 있는지 확인하고, 만약 깨워야 할 스레드가 있다면 alarms에서 해당 alarm을 pop 시킨 후 스레드를 unblock 합니다. 따라서 while문을 통해 알람 리스트가 비어있지 않을 때까지 깨워야 할 스레드를 탐색하고 깨웁니다. 이전 과정들을 통해 alarms는 만료 시간 기준 오름차순으로 미리 정렬되어 있으므로, alarms의 첫번째 요소를 가져온 후에 알람 리스트에서 알람을 제거하고 알람을 요청한 스레드를 깨우는 작업을 반복합니다. 만약 알람의 만료 시간이 현재 시간보다 커진다면 반복문을 종료하고, thread_tick함수를 호출하여 스레드 스케줄링을 수행합니다. 이를 통해 해당 시간에 깨워야 할 스레드들을 깨울 수 있습니다. 또한 이전에 구조체에 할당했던 메모리는 list_pop_front함수를 통해 자동으로 해제됩니다. 코드는 [그림 4]와 같습니다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  while (!list_empty (&alarms))
    {
      struct list_elem *e = list_front (&alarms);
      struct alarm *a = list_entry (e, struct alarm, elem);

      if (a->expiration > ticks)
        break;

      list_pop_front (&alarms);
      thread_unblock (a->th);
    }

  thread_tick ();
}
```

[그림 4] timer_interrput함수 수정

위와 같은 과정들을 통해 busy-waiting방식을 block, non-block방식으로 변경함으로써 Alarm Clock을 개선할 수 있습니다.

Simulation results – 코드를 수정한 후 pintos가 제공하는 test를 통해 개선된 Alarm Clock이 잘 작동함을 확인할 수 있습니다. [그림 5]는 test의 수행 결과입니다.

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
```

[그림 5] test 수행결과

Conclusions – 앞서 pintos의 timer.c 소스코드를 수정하여 비효율적인 busy-waiting방식을 blocking, non-blocking방식으로 개선했습니다. 기존 busy-waiting 방식은 계속해서 CPU를 사용하여 대기하면서 매 틱마다 확인하는 작업을 반복합니다. 이는 CPU 자원을 낭비하고 전력 소모를 증가시키는 문제를 일으킵니다.

반면, blocking, non-blocking 방식은 대기 상태에 들어간다면 해당 시간까지 실행이 중지되므로 CPU를 다른 작업에 할당할 수 있습니다. 이를 통해 CPU자원의 효율적인 활용과 전력 소모 감소를 통해 시스템의 성능을 개선할 수 있습니다.