OS Project 1 - Team 4

20185083 서영석, 20185141 이준명, 20185010 권강민

1. Introduction

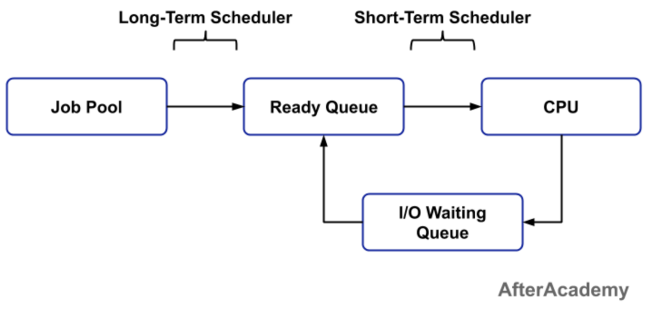
This assignment was conducted to improve the efficiency of the `timer.c/timer_sleep` function implemented on the existing PintOS platform. Our team modified the existing function that performed busy waiting to operate using thread waiting queue, enabling thread waiting even in idle states. As a result, we were able to solve the significant problem-waiting threads continuously occupying resources. Additionally, we discussed the limitations of the re-implemented code.

2. Overview

A. Problem - busy waiting

The main problem with the existing code is busy waiting in the `sleep_timer` function. When the target thread calls `thread_yield` function repeatedly to check for ticks in a while loop, the thread is continuously put into the ready queue. As a result, the target thread running `sleep_timer` continues to occupy resources. Busy waiting involves executing function calls and context switching continuously, leading to excessive use of computing resources, which can create several issues such as preventing the execution of other jobs or causing the CPU to overheat.

B. Problem solving - waiting queue



[Figure 1] Diagram of waiting queue (I/O waiting queue), (from afteracademy.com)

In general, operating systems use multiple queues during scheduling, among which the ready queue and waiting queue are the most important. The ready queue is a queue containing threads that are 'ready' to be allocated system resources, but the waiting queue consists of threads that have requested tasks such as I/O and are 'waiting' for the results of those tasks. Threads in the waiting queue return to the ready queue and start functioning after receiving the results of their requested tasks.

However, in Project 1 of PintOS, only ready_queue exists. It is defined in the `thread.c` file as a variable called as `ready_list`. And, the `next_thread_to_run` function, which is called during scheduling, refers to `ready_list` to return the next thread to run.

- `threads/thread.c`

```
void
thread_yield (void)
{
...
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
...
```

Originally, in the `thread_yield` function, busy waiting is implemented by continuously inserting the current thread into the ready list. Therefore, our team re-implemented `sleep_timer` to prevent busy waiting by defining an additional waiting queue. When calling `sleep_timer`, we block the target thread and add it to the waiting queue. Then, we call the function to check the waiting condition(tick limit) of threads in the waiting queue when scheduling starts. Threads whose waiting condition is met are unblocked and inserted into the Ready Queue.

3. Code-Level development

The expiration of the waiting condition should be checked even when the target thread is blocked. Therefore, we removed the tick comparison statement performed in the while loop on `timer.c`. Instead, we defined the `thread_sleep` function in `thread.c`, for checking ticks of the idle thread by calling it from `timer.c`

- `devices/timer.c`

```
void timer_sleep(int64_t ticks){
  ASSERT (intr_get_level() == INTR_ON);
  thread_sleep(ticks);
}
```

We added `block_list` and `blockelem` to `thread.h` and `thread.c` to handle the role of the waiting queue. Additionally, we defined `wait_start_time` and `wait_ticks` in the thread structure to store the tick information that the thread needs to wait for(waiting condition).

- `threads/thread.c`

```
static struct list ready_list;
static struct list all_list;
static struct list block_list;//added
```

- `threads/thread.h`

```
struct thread
  {
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */
    struct list_elem elem;              /* List element. */
    struct list_elem blockelem;//added
    int64_t wait_start_time;//added
    int64_t wait_ticks;//added

#ifdef USERPROG
    uint32_t *pagedir;                  /* Page directory. */
#endif
    unsigned magic;                     /* Detects stack overflow. */
  }
```

When the `thread_sleep` function is called, the kernel stores the waiting information of the corresponding thread and saves it in the `block_list`. Then, the thread is blocked. However, if the thread receives an interrupt during this process, the `thread_block` may not be called after the thread is added to the `block_list`.

To solve this problem, we use the `intr_disable` function to make the operation of adding a thread to the waiting queue an atomic operation. Additionally, to shorten the time it takes to check the waiting condition when inserting a thread into the Block_list, we created a sorted list based on the remaining waiting time.

- `threads/thread.c`

```
void
thread_sleep(int64_t ticks){
  enum intr_level old_level = intr_disable();
  struct thread * cur_thread = thread_current();
  cur_thread->wait_start_time = timer_ticks();
  cur_thread->wait_ticks = ticks;
  list_insert_ordered(&block_list, &(cur_thread->blockelem),compare_wait_left_tick,NULL);
  thread_block();
  intr_set_level (old_level);
}

...
// function for compare waiting time
bool compare_wait_left_tick(const struct list_elem *a, const struct list_elem *b, void * aux UNUSED){
  struct thread * a_thread = list_entry(a,struct thread,blockelem);
  struct thread * b_thread = list_entry(b,struct thread,blockelem);
  int64_t a_leftTick = get_remain_time(a_thread);
  int64_t b_leftTick = get_remain_time(b_thread);
  bool isSmaller =  a_leftTick <= b_leftTick;
  return isSmaller;
}
int64_t get_remain_time(struct thread * thread){
  return (thread->wait_ticks) - timer_elapsed(thread->wait_start_time);
}
```

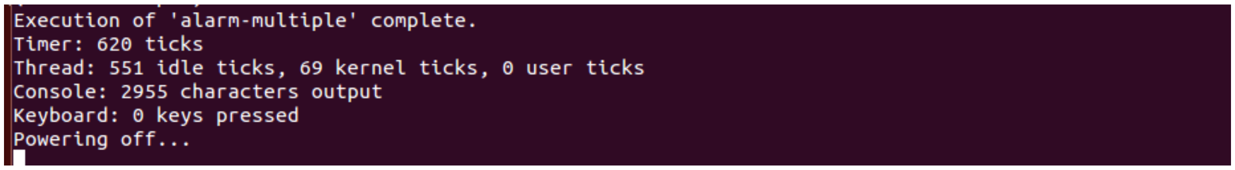
During scheduling, we check if the block condition has expired. Since the `block_list` is a sorted list, the first element of list is the thread with the smallest waiting time. Therefore, we unblock the first element of the `block_list` until it becomes an element that has not expired. When `thread_unblock` function is called, the thread inserted to the ready queue and wait the resources (CPU).

- `threads/thread.c`

```
// inspect waiting queue and check block condition
void check_wait_threads(){
  struct list_elem * a0;
  for(; !list_empty(&block_list);){
    a0 = list_begin(&block_list);
    struct thread * least_wait_left_thread = list_entry(a0, struct thread, blockelem);
    if(get_remain_time(least_wait_left_thread) < 0){
      list_remove(a0);
      thread_unblock(least_wait_left_thread);
    }
    else{
      break;
    }
  }
}

...
schedule (void)
{
  check_wait_threads();
  //other scheduling codes.
  ...
}
```

4. Result



```
Execution of 'alarm-multiple' complete.
Timer: 620 ticks
Thread: 551 idle ticks, 69 kernel ticks, 0 user ticks
Console: 2955 characters output
Keyboard: 0 keys pressed
Powering off...
```

[Figure 2] Execution result for our code

	idle ticks	kernel ticks	user ticks
Ours	551	69	0
Given example	550	382	0

[Table 1] Compare with our result and given example

Our experimental results are well demonstrated in Figure 2 and Table 1. Compared to the example results, we can see that idle tick increased similarly. However, the kernel tick showed a significant difference.

5. Discussion.

A. complexity

1. When `thread_sleep` function inserts an element to the `block_list` using a sorting algorithm, it has a time complexity of $O(N)$. And, storing the thread waiting information in the thread structure takes $O(1)$ time. Therefore, thread_sleep has a time complexity of $O(N)$.
2. When checking the block condition, its maximum time complexity is $O(N)$. However, in reality, it is rare for many threads to simultaneously expire the block condition, resulting in an average time complexity of $O(1)$.

B. flexibility

- The biggest problem in our implementation is its lack of flexibility. Time waiting is not the only type of various waiting types; There are various types of waiting such as I/O waiting(keyboard, HDD, and network), and users should be able to handle them through system calls. However, if we implement all waiting status by storing block conditions within the thread struct, the thread struct becomes bloated. This makes it difficult to manage the code after implementation and increases thread initialization costs and unnecessary memory usage.

Solutions

1. The aforementioned problem can be solved by introducing a new `block_information` structure. The `block_information` structure includes a thread pointer, a `block_condition_check` function pointer, and an auxiliary argument pointer, separating the implementation of `block_information` from the thread structure. This allows the definition of separate exception handlers for different types of waiting and the addition of waiting conditions without modifying the thread structure.
2. However, in this case, it is not possible to compare the precedence between `block_information`, so the `block_list` cannot be made into a sorted list. Therefore, the time complexity of condition checking increases to $O(N)$. Therefore, it is difficult to solve both flexibility and computational cost with the current method of the kernel checking all waiting conditions at schedule time.
3. Therefore, there is a need to expand the `block_information` to allow waiting conditions to be checked in a different execution flow using interrupts rather than scheduling.