Operating System Project 1

Alarm Clock

20195008 곽병혁, 20195052 김희수, 20195182 최승훈

March 21, 2024

1 Objectives

1. Configure your local system to develop the PintOS project (2 pts)
2. Implement Alarm Clock in the PintOS project (2 pts)
3. Write down a report for explaining policy and mechanism of implemented alarm clock (1 pt)

2 Definition

As shown in **Figure 1**, a thread life cycle in PintOS starts at THREAD_READY state when a program is loaded on the main memory. Then such thread runs on the processor if it is available, THREAD_RUNNING, and then can be THREAD_BLOCKED while it waits for other data such as IO response or system call. Lastly, the thread reaches THREAD_DYING state after finishing running.

This project focuses on a mechanism called "busy waiting" which is implemented in PintOS. Busy waiting occurs when a certain process repeatedly checks its conditions by occupying the processor's resources. This busy waiting approach is easy to understand and implement compared to other methods such as utilizing interrupts or signals. However, the process monopolizes the processor so other processes need to wait even though they are high-priority tasks than the process that occupies a CPU doing unnecessary calculations. Thus, we aim to improve the system in PintOS that wastes the processor's resources due to busy waiting.

3 Policy and Algorithm Design

3.1 Policy

The busy waiting approach in the original PintOS makes the thread stuck in the THREAD_READY state and THREAD_RUNNING state loop. In order to prevent this symptom, each thread should turn into THREAD_BLOCKED state at *time_sleep* step, not into THREAD_READY. This solution enables threads to use a CPU only when they need it and resolve resource waste problems. Threads should be *blocked* for *ticks* times and *unblocked* after *ticks* by modifying *timer_sleep()* function.

3.2 Algorithm

1. Call *timer_sleep()*
   timer_sleep() use *ticks* as a parameter to sleep a thread for certain ticks.
2. Block thread
   timer_sleep() blocks the thread for certain ticks using *MACRO_BLOCK_THREAD_UNTIL_TICK*. The macro decides how long the thread will be blocked by adding *start* and *ticks*.

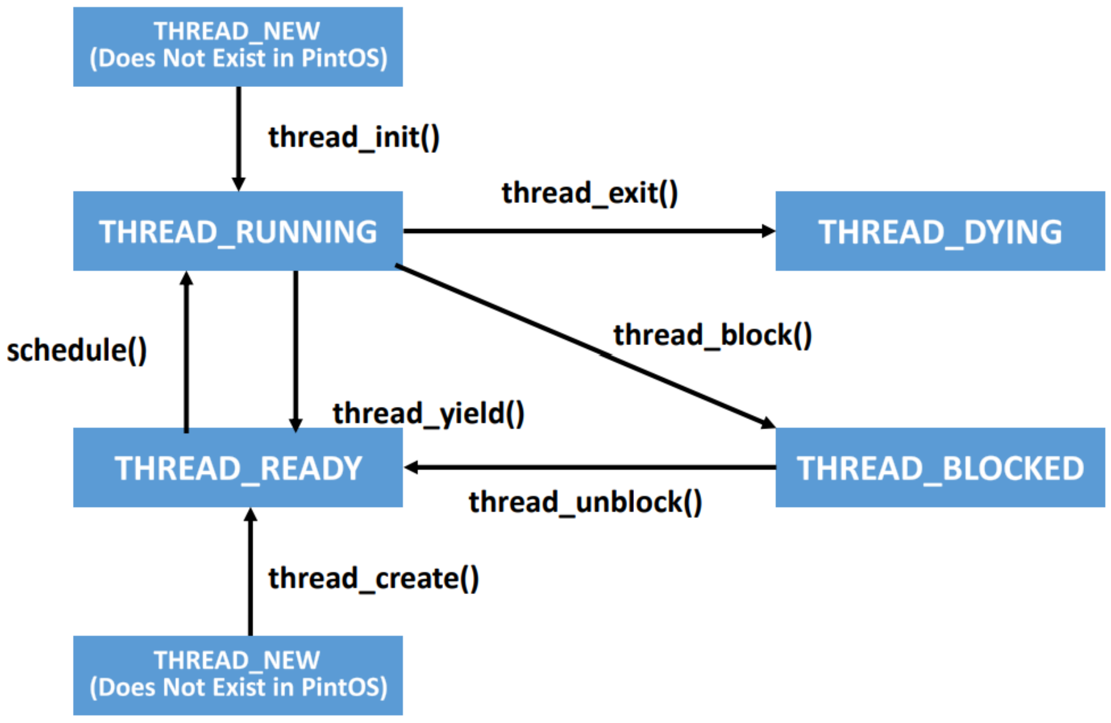


Figure 1: Process life cycle in PintOS.

3. Call *block_thread_until_tick()* to block the thread
   block_thread_until_tick() sets the current thread's *sleep_tick* in the thread structure type, adds such thread in *blocked_list* and then converts the thread into THREAD_BLOCK state.

4. Control interrupt level for safety
   The system prevents unexpected stops by interrupts by adjusting the interrupt level. *unblock_thread_after_tick()* call the *intr_disable()* before changing the thread's state. This ensures the thread's state-changing operations can be operated safely and without any interrupts. After changing the state, *SET_INTR* macro calls *intr_set_level()* to restore the previous level.

5. Call *unblock_thread_after_tick()* to unblock thread
   When it is time to wake up the process, *unblock_thread_after_tick()* enumerates *blocked_list*, check a condition that whether current *ticks* is larger than *sleep_tick*, remove the thread from the list and unblock it if such condition satisfies.

4 Mechanism

Four things should be clarified before explaining the mechanisms below for the algorithm implementation.

1. *block_thread_until_tick()*
   A function that converts threads into block state instead of busy waiting in *timer_sleep()*

2. *blocked_list*
   Double-linked struct list that contains blocked threads

3. *unblock_thread_after_tick()*
   A function that regularly searches threads that should be unblocked among blocked ones through *timer_interrupt()*

4. *sleep_tick*
   New int64 member variable in thread structure that records each tick when itself will wake up

First of all, *block_thread_until_tick()* receives a ticks gap from the time OS booted to the goal sleep duration as a parameter. The ticks gap parameter initializes the *sleep_tick* in the struct of thread which is executing *time_sleep()*. The thread in which the goal tick sets is pushed at the tail of *blocked_list* by *list_push_back()* in list.c. Next, the thread state turns blocked and then the function terminates. We convert interrupt flags to INTR_OFF since interrupt can disrupt the internal process in the function and defined *SET_INTR* macro function which contains *intr_set_level()* for the consistent interrupt flag. Furthermore, a global variable *first_sleep_tick* is declared in thread.c. This variable saves the value that blocked thread's *sleep_tick* whereby it uses for the condition in *timer_interrupt()* that executes *unblock_thread_after_tick()* when current tick passes *first_time_tick*. Note that a new blocked thread is pushed at the tail of *blocked_list*. *unblock_thread_after_tick()* utilizes the *blocked_list* when it looks for threads that need to be unblocked by while loop. In fact, specifically, *blocked_list* has the same form with predefined *ready_list*, *head* and *tail* list_elem exists in the struct. In addition, pointers that point to another list_elem, *prev* and *next*, are in the struct. Hence, these constructs *blocked_list* in the form of the double-linked list.

Code 1 illustrates the way to use *block_thread_until_tick()* function without using *thread_yield()* that keeps threads scheduling on the ready state.

```c
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);

  // added
  int64_t *ptrs=&start, *ptrtick=&ticks;
#define MACRO_BLOCK_THREAD_UNTIL_TICK(start, ticks) block_thread_until_tick(*(int64_t*)(
    start) + *(int64_t*)(ticks))
  MACRO_BLOCK_THREAD_UNTIL_TICK(ptrs, ptrtick);
  /*
  Previous code that causes busy waiting

  while (timer_elapsed (start) < ticks)
    thread_yield ();
  */
#undef MACRO_BLOCK_THREAD_UNTIL_TICK*/
}
```

Code 1: While loop using thread_yield is deprecated and the new MACRO function block_thread_until_is defined above.

Blocking threads is simple as shown in **Code 1**, whereas unblocking threads at a desired time requires continuous condition validations. As *timer_interrupt()* is a predefined function, it yields interrupts regularly in a certain period based on a constant variable, *TIMER_FREQ*. Therefore, This regularly executing *timer_interrupt()* is suitable to handle threads to be wakened up by checking time conditions repeatedly as shown in **Code 2** below.

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

  // added
  if(first_sleep_tick <= ticks)
  {
    unblock_thread_after_tick(ticks);
  }
  //
}
```

Code 2: When total executed time (ticks) overcomes first_sleep_tick unblocking is executed.

Ticks, the global variable in timer.c compares the value with the other global variable, *first_sleep_tick*, in thread.h every execution of *timer_interrupt()*. A thread to be unblocked exists when *first_sleep_tick* is smaller than *ticks* so *unblock_thread_after_tick()* executes to unblock proper threads.

```c
void
unblock_thread_after_tick (int64_t ticks)
{
#define MOVE(elem) list_next(elem)
#define UNBLOCK_THREAD(e, pntr, tick) \
  if ((tick)<list_entry(e, struct thread, elem)->sleep_tick) { \
    e = MOVE(e); \
  } else { \
    e = list_remove(&pntr->elem); \
    thread_unblock(pntr); \
  }
#define SET_TICK_CONDITION(tick, cond) \
  if(cond) { \
    set_first_sleep_tick(tick); \
  }

  struct list_elem *elem_pntr;
  elem_pntr = list_begin(&blocked_list);

  while(elem_pntr != list_end(&blocked_list))
  {
    struct thread* tmp_thrd_pntr = list_entry(elem_pntr,
                                               struct thread,
                                               elem);

    SET_TICK_CONDITION(tmp_thrd_pntr->sleep_tick, ticks < tmp_thrd_pntr->sleep_tick);
    UNBLOCK_THREAD(elem_pntr, tmp_thrd_pntr, ticks);
  }
#undef MOVE
#undef UNBLOCK_THREAD
#undef SET_TICK_CONDITION
}
```

Code 3: While traversing blocked_list unblock threads only which expired the sleep_tick.

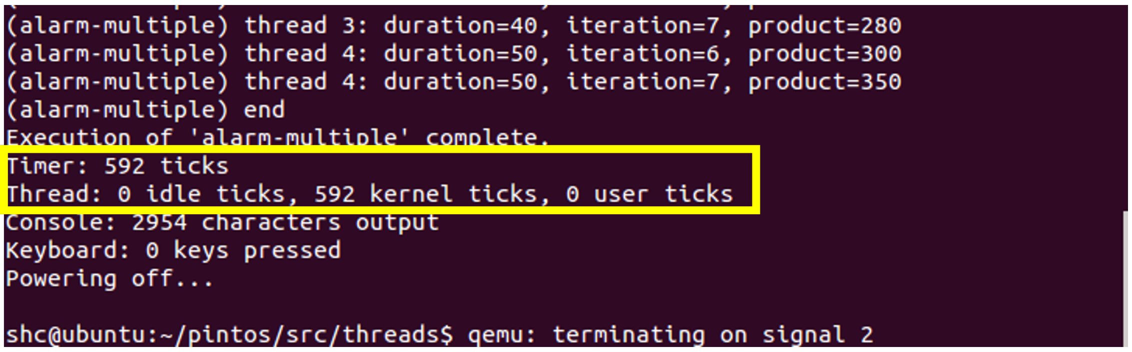
unblock_thread_after_tick() defined in thread.c (**Code 3**) retrieves whole list_elem in *blocked_list* from head to tail using *list_begin()*. A while loop iteration separates threads whether they will be unblocked or not. If *sleep_tick* that pointer points is larger than *ticks*, new defined *SET_TICK_CONDITION* sets *sleep_tick* to *first_sleep_tick*.

After the pointer used in *UNBLOCK_THREAD*, it points the next list_elem in *blocked_list* by *MOVE*. Otherwise, a list_elem the pointer points at from *blocked_list* is removed by *list_remove()*. At last, *thread_unblock()* is executed.

5 Result

An idle thread is a running thread when the CPU has no task to perform. the code in PintOS was implemented with the busy-waiting method which resulted in 0 idle ticks changed to 550 idle ticks. *timer_sleep()* occupies the processor when PintOS adopts a busy waiting approach so that idle threads cannot be executed. Through this project, however, blocked threads did not occupy the resources anymore and idle threads executed. In conclusion, system resources are able to be consumed properly.

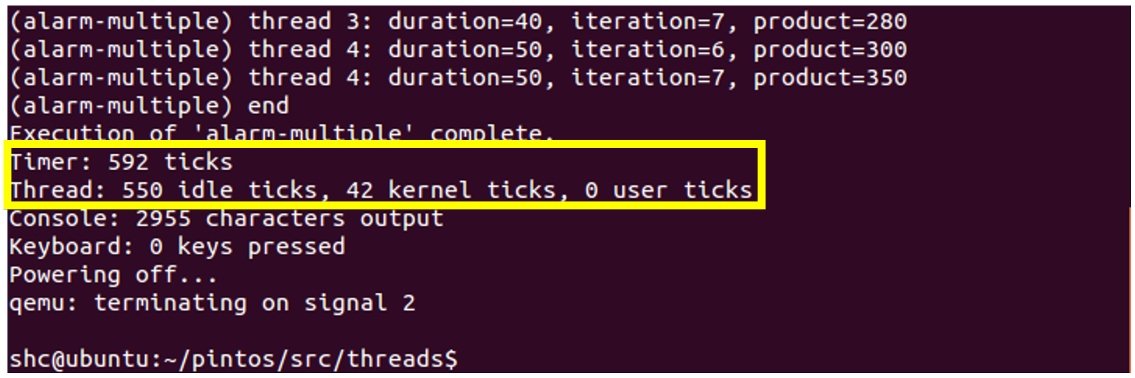
Acknowledgment All source codes are available in https://github.com/ddeunghoon/PintOS_team10.



```
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 592 ticks
Thread: 0 idle ticks, 592 kernel ticks, 0 user ticks
Console: 2954 characters output
Keyboard: 0 keys pressed
Powering off...

shc@ubuntu:~/pintos/src/threads$ qemu: terminating on signal 2
```

(a)



```
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 592 ticks
Thread: 550 idle ticks, 42 kernel ticks, 0 user ticks
Console: 2955 characters output
Keyboard: 0 keys pressed
Powering off...
qemu: terminating on signal 2

shc@ubuntu:~/pintos/src/threads$
```

(b)

Figure 2: Results of **pintos -q run alarm-multiple** command. **2a** demonstrates ticks without any modification in PintOS that no idle ticks exist. The improved processor usage is shown in **2b**.