Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project

Group 11

Giorgia Lillo, Lamberto Ragnolini

Fall Semester, March 6, 2024

Report Instructions

- Please report **all** changes, even if minor, that you did to complete the project.
- You have to list all files that have been modified, and for each of them, list all functions/structs that have been modified or added (clearly stating "modified"/"added"). Then add a brief explanation or motivation for all those changes.
- A single report is required for each group. For the first individual project, each student submits a report together with the source code files that were changed.

1 Files changed

- /pintos-env/pintos/devices/timer.c
- /pintos-env/pintos/threads/thread.h

2 Changes

/pintos-env/pintos/devices/timer.c

- **struct list holder;** (*added*): -> list of threads declaration

  Declaration of a list that will be used to store the blocked threads.

- **timer_init(void)** (*modified*):
  - pit_configure_channel (0, 2, TIMER_FREQ); (not changed) - **intr_register_ext (0x20, timer_interrupt, "8254 Timer");** (not changed)

- **list_init(&holder);** (added) -> Initialization of the thread list declared previously.

We initialize the list as an empty list as soon as the timer is initialized. We use the function list_init(struct list *) provided by the interface list.h.

- **timer_sleep (int64_t ticks)** (*modified*) -> modified version to avoid busy waiting:

The function avoids busy waiting by blocking the threads that needs to sleep for a certain amount of time, placing them into a list sorted by wake up time (field added in struct thread, watch thread.h modification below). The implementation disables interrupts in the critical section and re-enables them as soon as the critical section is over.

- **int64_t start = timer_ticks ();** (not changed)
- **ASSERT (intr_get_level () == INTR_ON);** (not changed)

- **struct thread * th = thread_current();** (added) -> Retrieving current thread that needs to sleep.

Retrieving thread that needs to sleep thanks to the "thread_current" method provided by thread.h interface. Retrieving the current running thread is needed later in order to block it and placing it in the list.

- **int64_t wk = start + ticks;** (added) -> Exact tick number on which the thread will be released

We compute the wake up time stored in variable wk thanks to the sum of the current tick number (start variable) and the amount of ticks the current thread should sleep(ticks, parameter passed to the method).

- **th->wakeup = wk;** (added) -> Assignment of wake up time to its proper fields in the current thread struct.

Simply assigning variable wk calculated previously to the field "wakeup" added by us in struct thread(watch thread.h modification below).

- **intr_disable();** (added) -> Disabling interrupts

Thanks to this function provided by interrupt.h interface we can disable interrupts in order to preserve ourselves from bad inconveniences due to the following incoming critical section.

- **list_insert_ordered(&holder,&th->elem,comparing,NULL);** (added) -> Current thread insertion in blocking list sorted by wake up time

This function is inserting the current thread in the blocking queue in a sorted (by wake up time) manner, using the comparing function(explained later) . It is useful to provide an efficient iteration later when the threads will need to be unblocked.

- **thread_block();** (added) -> Blocking the current thread

Blocking the thread once added to the list. This gives us the great possibility of avoiding busy waiting by avoiding a continuous check of the thread elapsing wake up time that is preventing the CPU from making progress, wasting CPU cycles. Thread block method is asserting that interrupts are off in order to be successful, so we need to block the thread before enabling the interrupts.

- **intr_enable();** (added) -> Enabling the interrupts

The critical section is now over, so we can re-enable interrupts.

- **comparing(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED);** (*added*) -> Comparison function used to compare wake up time of threads:

  This function is passed as reference parameter in the above described "list_insert_ordered" function in "timer_sleep" function. It is used to compare the wake up times of threads in order to provide a valid sorting (in our case by wakeup time) in the blocking list.

  - **struct thread thea = list_entry(a, struct thread, elem);** (added)
  - **struct thread theb = list_entry(b, struct thread, elem);** (added)
  - **return thea->wakeup < theb->wakeup;** (added)

  Retrieving the struct thread items in the list and comparing them in order to be able to place the incoming thread in right position in the list. The actual work is done in the "list_insert_ordered" function that works by using the "comparing" function multiple times in a loop in order to compare the wake up times.

- **timer_interrupt (struct intr_frame *args UNUSED)** (*modified*) -> modified version of "timer_interrupt" to check possible unblocking threads:

  When this function gets called we iterate over the list of blocked threads in order to check if some thread wake up time is greater or equal to the tick number in the moment this function gets called. In case this condition is satisfied we will be able to unblock the threads satisfying it.
  NOTE: Remember that our list is sorted by wake up time so in order to be efficient we can break the iteration as soon as the condition is not satisfied by the current thread in the iteration.

  - **ticks++;** (not changed)
  - **thread_tick ();** (not changed)

  - **struct list_elem * pos;** (added) -> Declaration of a pointer for the iteration
  This pointer is used to iterate over the elements of the list.

    - **while (!list_empty(&holder)) {** (added) -> Iteration over the list of blocked threads

    Initialization of a loop that in the worst case iterates over all the elements of the list of blocked threads (worst case complexity: O(n)) or exits immediately (best case complexity: O(1)). The body of the loop will check the wake up times of the threads in the list.

- **pos = list_front(&holder);** (added) -> Assigning current thread in iteration to the iteration pointer

We will always retrieve the first element of the list everytime we enter the loop due to the implementation of the body of the while loop (below here). The implementation, given that the list is sorted, will always pop the first element if satisfying the condition below, otherwise it will break out of the loop, so if we are iterating we will always pop the first element and access the new one which will be now in the first position in the list.

- **struct thread i;** (added)
- **i = list_entry(pos, struct thread, elem);** (added)

Retrieving the thread currently iterated on in order to be able to unblock it. List entry is giving us the outer struct (in our case struct thread) on which the iterated element is in. Due to the list implementation the iteration element is inside the item in the list and the actual item in the list (struct thread in our case) is the outer struct containing it. We need then to retrieve the actual thread struct by referencing to the outer struct which is different from what we are referencing while iterating on them. We will unblock the thread only if the below condition will be satisfied.

- **if (timer_ticks() >= i->wakeup) {** (added) -> Condition checking if the current iterated thread can be unblocked

The condition checks if the current tick number is greater or equal than the wake up number, in case this is satisfied we will be able to remove the thread from the list and unblock it.

- **list_pop_front(&holder);** (added) -> Removes the thread from the list (the first in the list)

This function will remove the first element in the list due to the explanation already explained above.

- **thread_unblock(i);** (added) -> unblocks the thread

We now can unblock the thread because the sleeping time has elapsed

- **} else { break; } }** (added) -> breaks the loop (exits the loop)

In case the condition provided by the if statement explained above is not satisfied, we will directly exit the loop in order to be efficient. As already explained there is no need to iterate over the list if we encounter a thread that has not reached its elapsed time due to the fact that the threads are ordered by wake up time in the list.

/pintos-env/pintos/threads/thread.h

- struct thread (*modified*):
  - **int64_t wakeup;** (added) -> field added to the struct

This field was added in order to store the exact tick number on which the thread will need to be unblocked. It is computed by the sum of the ticks number retrieved at the moment of the blocking of the thread and the actual amount of ticks that the thread needs to be blocked.