Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 3

Group 11
Giorgia Lillo, Lamberto Ragnolini

Fall Semester, March 28, 2024

1 Files changed

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

2 Changes

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

- struct thread (*modified*) -> added the nice and recent_cpu fields

```
1 //Nice value of the thread for advanced scheduling
2 int nice;
3 //Fixed - real number used to determine how much cpu the thread has used
     recently
4 FPReal recent_cpu;
```

/pintos-env/pintos/threads/thread.c

- INITIAL SETUP:

  Our implementation is based on the ready_list being sorted by thread's priority meaning that at the beginning of the list there will be the ready thread with the highest priority. In order to do this initially we changed the function call to list_push_back inside the methods

thread_unblock and thread_yield to the function call list_insert_ordered defining also an auxiliary compare function explained later below.

- global variables added:

```
/* stores the system load (in number of ready+running threads per second in
   the last minute), initialized to 0*/
FPReal load_avg;

/*range of values for the nice var*/
int MIN_NICE = -20;
int MAX_NICE = 20;
```

- thread_init (*modified*)

NOT MODIFIED

```
void thread_init(void)
{
  ASSERT(intr_get_level() == INTR_OFF);

  lock_init(&tid_lock);
  list_init(&ready_list);
  list_init(&sleep_list);
  list_init(&all_list);
```

only added this initialization

```
  /*initialize load average to 0 and transform it into an FPReal*/
  load_avg = INT_TO_FPR(0);
```

NOT MODIFIED

```
  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread();
  init_thread(initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid();
}
```

- init_thread (*modified*)

NOT MODIFIED

```
/* Does basic initialization of T as a blocked thread named
   NAME. */
static void
init_thread(struct thread *t, const char *name, int priority)
{
  ASSERT(t != NULL);
  ASSERT(PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT(name != NULL);
  memset(t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy(t->name, name, sizeof t->name);
  t->stack = (uint8_t *)t + PGSIZE;
```

MODIFIED

```
  /*if we are not in advance scheduler*/
  if (!thread_mlfqs)
  {
    t->priority = priority;
  } else {
    /*if we are in advance scheduler*/
    t->recent_cpu = INT_TO_FPR(0);
    t->nice = 0;
  }
  t->magic = THREAD_MAGIC;
  list_push_back(&all_list, &t->allelem);
}
```

this function initializes a thread so whenever a thread is initialized, we check whether the priority scheduler or the advance scheduler is enabled, in order to do this we check the boolean variable thread_mlfqs, if the variable is set to false than we are using the priority scheduler so we just assign the priority to the thread's field, otherwise if the advance scheduler is enabled we set the recent_cpu value and the nice values of the thread to 0.

- check_priority_range (*added*)

```
/*auxiliary that checks that the priority lies in the correct range*/
void check_priority_range(struct thread *t, void *AUX UNUSED)
{
  if(t->priority > PRI_MAX) t->priority = PRI_MAX;
  else if(t->priority < PRI_MIN) t->priority = PRI_MIN;
}
```

This helper function simply checks if the priority of a thread lies in the correct range : 0 <= thread priority <= 63 otherwise whether the value is smaller than PRI_MIN which is the lower bound set to 0 we just clamp the priority to PRI_MIN and reassign it to the thread's priority, whether the value is higher than PRI_MAX which is the upper bound set to 63 we set the priority to PRI_MAX and reassign it to the thread's priority

- update_recent_cpu (*added*)

```
void update_recent_cpu(struct thread *t, void *aux UNUSED)
{
  /*assigning/reassiging the recent_cpu value to the thread based on the
     formula -> (2     load avg)/(2     load avg + 1)     recent cpu + nice
     */
  FPReal numerator = FPR_MUL_FPR(INT_TO_FPR(2), load_avg);
  FPReal denominator = FPR_ADD_INT(numerator, 1);
  FPReal division = FPR_DIV_FPR(numerator, denominator);
  t->recent_cpu = FPR_ADD_INT(FPR_MUL_FPR(division, t->recent_cpu), (t->nice
     ));
}
```

This helper function simply updates/calculates the recent_cpu value of a thread and assignes/reassignes it to it. In order to calculate this value we use the right formula provided by the assignment and the right FPReal functions provided by fpr_arith.h in order to compute the exact value.

- update_priority (*added*)

```
void update_priority(struct thread *t, void *aux UNUSED)
{
  /*save the current priority which will become the old since we are
     updating it (maybe)*/
  int old = t->priority;
  /*update it w.r.t to the formula -> PRI MAX     (recent cpu/4)     (nice
        2)*/
  t->priority = (PRI_MAX - FPR_TO_INT(FPR_DIV_INT(t->recent_cpu, 4)) - (t->
     nice * 2));
  /*check the correctness of the priority (it doesn't go out of bounds)*/
  check_priority_range(t, NULL);
  /*if priority was changed, the thread status is ready and the thread is
     not already the current one then we can re-insert it in the sorted
     list since it's position could be changed*/
  if(t->status != THREAD_RUNNING && t->status != THREAD_BLOCKED && t->status
      != THREAD_DYING && t != thread_current() && old != t->priority){
    list_remove(&(t->elem));
    list_insert_ordered(&ready_list, &t->elem, compare_threads, NULL);
  }
}
```

This is a helper function used to calculate/recalculate the priority of a thread. We first save the value of the priority of the thread before any computation, after this we set the new priority to the thread by calculating it using the right formula provided in the assignment and by using the right FPReal functions provided in the fpr_arith.h file, after being done with the calculation, we use a helper function that is check_priority_range already explained above to be sure that the priority lies in the right range : 0 <= priority <= 63. Once all of this is done, if the thread is in a THREAD_READY status, it is not the idle_thread and the its priority has actually changed then we need to reinsert it in the ready_list because maybe now it is not in the right sorted position anymore due to its updated priority. (the ready_list is sorted by priority)

- thread_tick (*modified*)

NOT MODIFIED

```
/* Called by the timer interrupt handler at each timer tick.
   Thus, this function runs in an external interrupt context. */
void thread_tick(void)
{
  struct thread *t = thread_current();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Enforce preemption. */
```

```
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return();

  check_for_sleeping_threads();
```

MODIFIED

```
  /*if advance scheduler is active*/
  if (thread_mlfqs)
  {
    /*check that the current thread is not idle in order to increment by 1
       its recent_cpu (we do it every interrrupt so everytime we are in
       this function)*/
    if (t != idle_thread)
    {
      FPR_INC(&(running_thread()->recent_cpu));
    }
    /*every fourth tick we update the priority of every thread*/
    if (timer_ticks() % 4 == 0)
    {
      thread_foreach(update_priority, NULL);
    }
    /*every second we update the load average w.r.t to the formula ->
       (59/60)     load avg + (1/60)     ready or running threads
      we update also the recent_cpu value to all the threads
    */
    if (timer_ticks() % TIMER_FREQ == 0)
    {
      /*count the number of ready and running threads by counting the
         elements in the ready_list adding also the current thread if it is
          not idle*/
      int ready_or_running_threads = list_size(&ready_list);
      if (thread_current() != idle_thread) {
        ready_or_running_threads++;
      }
      /*calculate load average and update it*/
      FPReal firstmul = FPR_MUL_FPR(FPR_DIV_FPR(INT_TO_FPR(59), INT_TO_FPR
         (60)), load_avg);
      FPReal secondmul = FPR_MUL_INT(FPR_DIV_FPR(INT_TO_FPR(1), INT_TO_FPR
         (60)), ready_or_running_threads);
      load_avg = FPR_ADD_FPR(firstmul, secondmul);
      // update
      thread_foreach(update_recent_cpu, NULL);
    }
  }
}
```

This function gets called every tick by the timer interrupt function. Only after the call to the function check_for_sleeping_threads we do some computation only if the advance scheduler is enabled, this is because we are checking if the variable thread_mlfqs is set to true. If then the advance scheduler is enabled we need to do multiple things : - every time the function gets called it means a timer tick happened so we need to increment the recent_cpu value of the current running thread by one. In order to do this we use the function FPP_INC provided by the file fpr_arith.h because we are incrementing a fixed-point real value. - every fourth tick we update the priority of each thread thanks to the help of the thread_foreach function that takes as argument our helper function that is update_priority already explained above. -

every second we need to do two things, the first that we do is the update of the load average global variable where we use the the right formula to calculate it which was provided in the assignment. just a quick point out for the variable ready_or_running_threads used in the formula of the update of the load average variable is that in order to calculate this we need to count all the threads that are in the ready_list plus the current thread if it not the idle thread. After having updated the load average we also need to recalculate the recent_cpu value for each thread and we do it to with the help of our helper function update_recent_cpu already explained above that is applied to every thread by passing it as argument to thread_foreach.

- thread_create (*modified*)

NOT MODIFIED

```
tid_t thread_create(const char *name, int priority,
                    thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  enum intr_level old_level;

  ASSERT(function != NULL);

  /* Allocate thread. */
  t = palloc_get_page(PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread(t, name, priority);
  tid = t->tid = allocate_tid();

  /* Prepare thread for first run by initializing its stack.
     Do this atomically so intermediate values for the 'stack'
     member cannot be observed. */
  old_level = intr_disable();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame(t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame(t, sizeof *ef);
  ef->eip = (void (*)(void))kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame(t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  intr_set_level(old_level);

```

```
  /* Add to run queue. */
  thread_unblock(t);
```

MODIFIED

```
  /*check if the priority of the just created thread is greater
    than the current running thread, if so, we yield the current running one
  */
  if (t->priority > thread_get_priority()) {
        thread_yield();
    }

  return tid;
}
```

This function creates a thread, this means that this thread could possibly have a priority higher than the current running thread so we need to check if this is the case. We added an if statement at the end of the function before the return statement that simply checks if the just created thread presents a higher priority than the current running thread (we retrieve the current running thread's priority with the function thread_get_priority that returns the current running thread priority), if it is the case we yield the current running priority.

- compare_threads (*added*)

```
bool compare_threads(const struct list_elem *a, const struct list_elem *b,
    void *aux UNUSED)
{
  struct thread *thea = list_entry(a, struct thread, elem);
  struct thread *theb = list_entry(b, struct thread, elem);
  return (thea->priority > theb->priority);
}
```

This function is simply the comparing priority function used by the method list_insert_ordered in order to provide a sorted by priority manner (highest first) in the ready_list so that we can be more efficient on computations and checks later.

- thread_set_priority (*modified*)

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void thread_set_priority(int new_priority)
{
  /*if we are not using the advance scheduler*/
  if (!thread_mlfqs)
  {
    /*retrieve current thread*/
    struct thread *curr = thread_current();
    /*if ready list is empty just let the current thread run*/
    if (list_empty(&ready_list))
    {
      /*just update the priority*/
      curr->priority = new_priority;
    }
    else
    {
```

```
      /*otherwise retrieve first element from ready list because it is
         sorted by priority*/
      struct list_elem *first = list_begin(&ready_list);
      struct thread *th = list_entry(first, struct thread, elem);
      /*assign new priority to current thread*/
      curr->priority = new_priority;
      /*if the current thread's priority is lower then the first in the
         queue which will be the thread with the highest priority, we yeld
         it*/
      if (curr->priority <= th->priority) thread_yield();
    }
  }
}
```

The body of the function executes only if the advanced scheduler is not enabled hence we are using the priority scheduler,this is because we are checking if the variable thread_mlfqs is set to false. If we are then using the priority scheduler, in the body We first retrieve the current thread, then if the ready_list is empty it means that there are no ready to run threads, so we simply set the new priority to the retrieved thread and we let it run, otherwise if the list contains some ready to run threads we check if the updated priority of the running thread is lower than the first element in the list, because the list is sorted by priority(highest first), if so we yield the current running thread because it has no more the highest priority.

- thread_get_priority (*NOT MODIFIED*)

```
/* Returns the current thread's priority. */
int thread_get_priority(void)
{
  return thread_current()->priority;
}
```

- thread_set_nice (*added*)

```

/* Sets the current thread's nice value to NICE. */
void thread_set_nice(int nice)
{
  if (thread_mlfqs)
  {
    /*assert nice value is in the range of possible nice values */
    ASSERT(MIN_NICE <= nice && nice <= MAX_NICE);
    struct thread *cur = thread_current();
    cur->nice = nice;
    /*adjust priority*/
    update_priority(cur,NULL);

    /*check priorities with the ready list if the list is not empty*/
    if (!list_empty(&ready_list))
    {
      /*retrieve first element from ready list which is sorted by priority*/
      struct list_elem *first = list_begin(&ready_list);
      struct thread *th = list_entry(first, struct thread, elem);
      /*if the current thread's priority is lower then the first in the
         queue which will be the highest, we yeld it */
```

```
      if (cur->priority <= th->priority) thread_yield();
    }
  }
}
```

The body of the function gets executed only if the advanced scheduler is enabled, this is because we are checking if the variable thread_mlfqs is set to true. In the body first of all we check with an assertion that the nice value is an acceptable value meaning that it lies in a range: -20 <= nice <= 20. After that we retrieve the current thread, set the new nice value to it, update its priority with an auxiliary function that is update_priority already explained above and after this if the ready_list is not empty we check if there is a thread in it that now has a higher priority, if so we yield the current thread.

- thread_get_nice (*implemented*)

```
/* Returns the current thread's nice value. */
int thread_get_nice(void)
{
  return thread_current()->nice;
}
```

We retrieve the nice value from the current thread and return it.

- thread_get_load_avg (*added*)

```
/* Returns 100 times the system load average. */
int thread_get_load_avg(void)
{
  return FPR_TO_INT(FPR_MUL_INT(load_avg, 100));
}
```

We retrieve the load average value, we multiply it by 100 with the help of the FPR_MUL_INT function provided in the fpr_arith.h file, then cast it with another function of the file to int and return it.

- thread_get_recent_cpu (*added*)

```
/* Returns 100 times the current thread's recent_cpu value. */
int thread_get_recent_cpu(void)
{
  return FPR_TO_INT(thread_current()->recent_cpu) * 100;

}
```

We retrieve the recent_cpu value from the current thread, then cast it with the function FPR_TO_INT provided by the file fpr_arith.h, we then multiply it by 100 and return it.

- next_thread_to_run (*NOT MODIFIED*)

```
/* Chooses and returns the next thread to be scheduled.  Should
   return a thread from the run queue, unless the run queue is
   empty.  (If the running thread can continue running, then it
   will be in the run queue.)  If the run queue is empty, return
```

```
   idle_thread. */
static struct thread *
next_thread_to_run(void)
{
  if (list_empty(&ready_list))
    return idle_thread;
  else
    return list_entry(list_pop_front(&ready_list), struct thread, elem);
}
```

We didn't need to modify it because the insertion of the elements in the ready_list is done in a sorted by priority manner. Since the method is popping the next thread from the front of the list, we are already sure that the popped thread will be the one with the highest priority.