1. Цель работы

Изучение системы управления процессами, а также механизма работы системного таймера в ОС Pintos, анализ его недостатков и модификация его алгоритма.

2. Ход работы

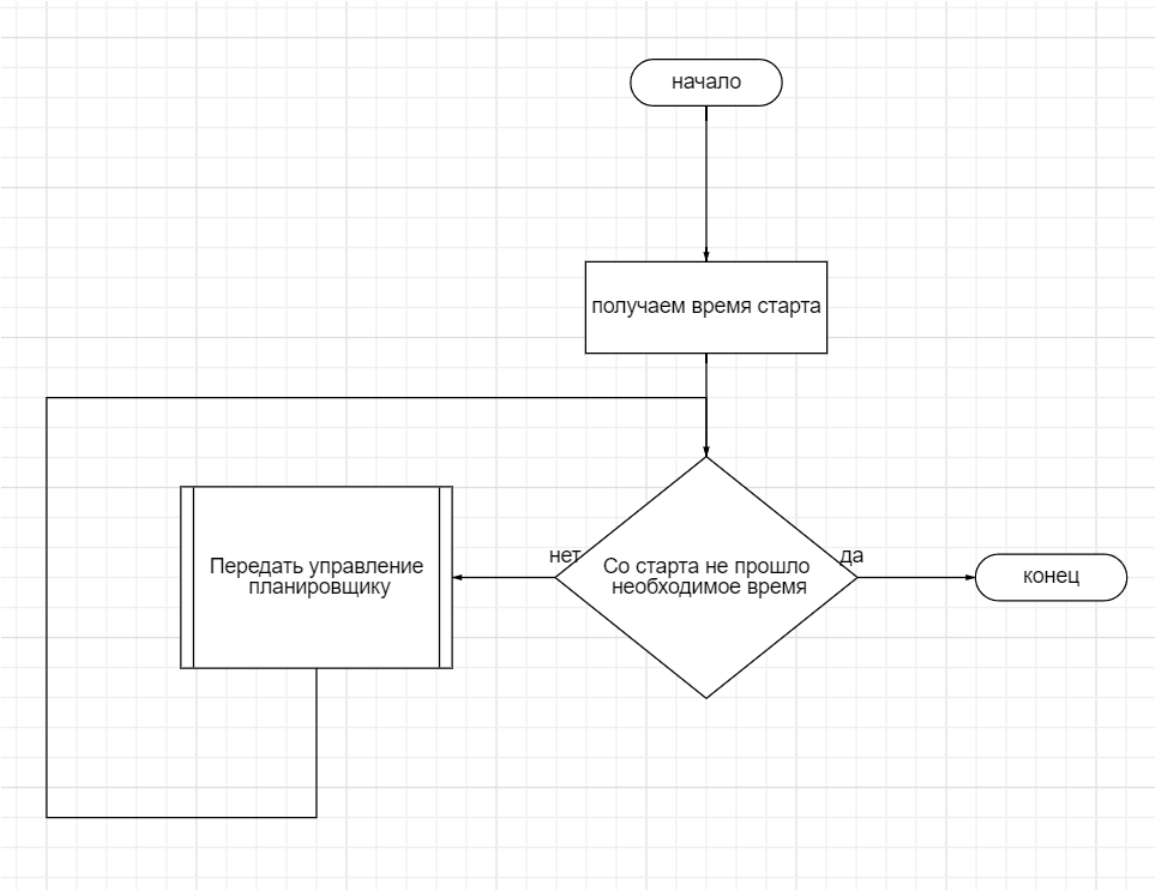
Ознакомившись с исходными кодами таймера, представленных в файлах timer.h, timer.c, а также с функциями работы с процессами, которые описаны в thread.h и thread.c, составим таблицу функций системного таймера с описанием аргументов, которые принимают эти функции, и действий, которые эти функции выполняют (Таблица 1).

Таблица 1 – Описание функций timer.h

Функция	**Действие**
void timer_init (void)	Устанавливает таймер на прерывание **TIMER_FREQ** раз в секунду и регистрирует соответствующее прерывание.
void timer_calibrate (void)	Калибрует **loops_per_tick**, используемый для реализации коротких задержек.
int64_t timer_ticks (void)	Возвращает количество тиков таймера, прошедших с момента загрузки ОС.
int64_t timer_elapsed (int64_t then)	Возвращает количество тактов таймера, прошедших с значения **THEN**, возвращенным функцией **timer_ticks ()**.
void timer_sleep (int64_t ticks)	«Засыпает» на **ticks** тактов. Прерывания должны быть включены.
void timer_msleep (int64_t ms)	«Засыпает» на **ms** миллисекунд. Прерывания должны быть включены.
void timer_usleep (int64_t us)	«Засыпает» на **us** микросекунд. Прерывания должны быть включены.

Функция	Действие
void timer_nsleep (int64_t ns)	«Засыпает» на **ns** наносекунд. Прерывания должны быть включены.
void timer_mdelay (int64_t ms)	Активное ожидание **ms** миллисекунд. Прерывания должны быть отключены. Активное ожидание тратит циклы процессора, а активное ожидание с выключенным прерыванием между тиками таймера или дольше приведет к потере тиков. Таким образом, нужно использовать **timer_msleep()**, если прерывания включены.
void timer_udelay (int64_t us)	Активное ожидание примерно **us** миллисекунд. Прерывания должны быть отключены. Активное ожидание тратит циклы процессора, а активное ожидание с выключенным прерыванием между тиками таймера или дольше приведет к потере тиков. Таким образом, нужно использовать timer_usleep(), если прерывания включены.
void timer_ndelay (int64_t ns)	Активное ожидание примерно **ns** наносекунд. Прерывания должны быть отключены. Активное ожидание тратит циклы процессора, а активное ожидание с выключенным прерыванием между тиками таймера или дольше приведет к потере тиков. Таким образом, нужно использовать timer_nsleep(), если прерывания включены.
void timer_print_stats (void)	Выводит статистику таймера с момента загрузки ОС.

Алгоритм активного ожидания, реализованный в timer_sleep() (Рисунок 1):



(Рисунок 1 – блок схема реализации алгоритма функции timer_sleep())

Базовая реализация системного таймера содержит в себе цикл активного ожидания. Данная реализация имеет огромные недостатки, такие, как бесполезное расходование процессорного времени и расход системных ресурсов. Следовательно, необходимо оптимизировать данный алгоритм следующим образом:

a) Если система находится в режиме ожидания (нет других запущенных потоков), то вызвавший функцию timer_sleep() процесс должен быть разбужен точно по истечении заданного времени;

b) Процесс не должен мешать другим процессам, которые исполняются в момент его пробуждения. Для решения этой проблемы необходимо каким-то образом хранить спящие процессы. На выбор предоставлены массив и список, так как мой вариант работы предусматривает реализацию через список, то им и воспользуемся

В следствие чего были введены следующие изменения:

1) Изменения в файле thread.h:

   Мною было принято решение дополнить структуру thread дополнительным свойством, а именно время сна для него:

```
struct thread_status
  {
    /* Owned by thread.c. */
    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. */
    int wake_up;
    int time_running;
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

2) Изменения в файле timer.h:

   Объявил в заголовочном файле timer.h новую структуру данных, массив спящих процессов, а в timer.c создадим глобальный указатель на начало нашего массива

3) Далее необходимо изменить саму функцию timer_sleep и избавиться от цикла while{}. Был реализован метод вставки в массив спящих процессов с сортировкой по убыванию (рисунок 4):

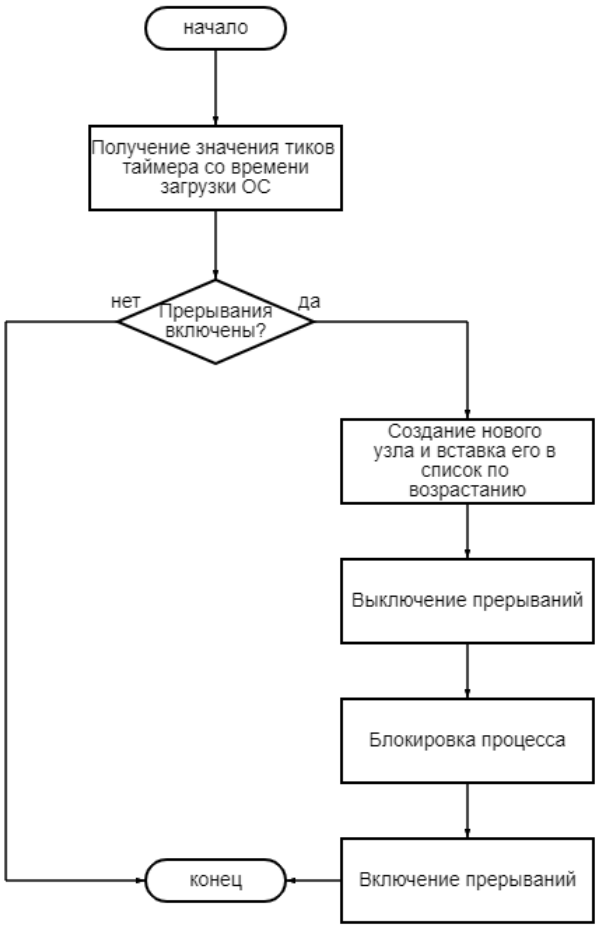
```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  int new_time = start + ticks;
  if (ticks < 0) return;

  struct thread* current = thread_current();
  //msg("--- %s\n", current->name);
  current->wake_up = start + ticks;
  struct thread* temp_1;
  struct thread* temp_2;
  int flag = 0;

  // if (global_numbers_sl_list >= max_cnt - 10){
  //    max_cnt = max_cnt * 2;
  //    sleep_list = (struct thread**)realloc(sleep_list, max_cnt);
  // }

  if (global_numbers_sl_list == -1){
    sleep_list = (struct thread**)calloc(50, sizeof(struct thread*));
    sleep_list[0] = current;
    global_numbers_sl_list += 2;
  }


  else if(global_numbers_sl_list == 0){
    sleep_list[0] = current;
    global_numbers_sl_list++;
  }

  else {
    for (int i = 0; i < global_numbers_sl_list; i++){
      if (current->wake_up < (sleep_list[i]->wake_up)) {
        temp_1 = sleep_list[i];
        sleep_list[i] = current;
        i++;
        global_numbers_sl_list++;
        //sleep_array[count] = temp;
        while (i < global_numbers_sl_list) {
          temp_2 = sleep_list[i];
          sleep_list[i] = temp_1;
          temp_1 = temp_2;
          i++;
        }
        flag = 1;
        break;
      }
    }

    if (flag == 0){
      sleep_list[global_numbers_sl_list] = current;
      global_numbers_sl_list++;
    }
  }
  intr_disable();
  thread_block();
  intr_enable();
```

(Рисунок 4 – файл timer.c)

4) Заключительное изменение было сделано в функции timer_interrupt(). Необходимо было добавить unblocked процесса, и удаление его из списка спящих процессов, а так же смещение остальных элементов (Рисунок 5 – файл timer.c)

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  if (global_numbers_sl_list > 0) {
    while ((sleep_list[0] != 0) && (timer_ticks() >= sleep_list[0]->wake_up)) {
      thread_unblock(sleep_list[0]);
      //global_numbers_sl_list--;
      for (int i = 0; i < global_numbers_sl_list; i++) {
        sleep_list[i] = sleep_list[i + 1];
      }
      global_numbers_sl_list--;
    }
  }
}

/* Returns true if LOOPS iterations waits for more than one timer
   tick, otherwise false. */
```

Блок схема нового алгоритма (рисунок 6):



(Рисунок 6 – файл timer.c)

Дополнительные задания

Задание 3 (Реализовать тесты "max-mem-malloc", "max-mem-calloc", "max-mem-palloc" в файле tests/threads/max-mem.c, измеряющий количество памяти, которая доступна в куче ядра)

Для реализации теста было написано три простых функции, схожие по работе, отличаясь способом выделения памяти.

1) Для написания теста max-mem-malloc введем пустой указатель, выделив ему память в размере 256 бит динамическим способом, а после зациклим присвоение, выбрав условием указатель типа !NULL (Указателю будет присвоен NULL, если не удалось выделить необходимое количество памяти)

```
void test_max_mem_malloc(void)
{
  //msg("Not implemented.");

  void *vrm = malloc(256);
  int max_mem_malloc = 0;
  while (vrm != NULL){
    vrm = malloc(256);
    max_mem_malloc++;
  }
  msg("WARNING: max_mem_malloc (x256) = %d", max_mem_malloc);
  msg("WARNING: max_mem_malloc (bytes) = %d", max_mem_malloc * 256);


}
```

2) Реализация теста max-mem-calloc полностью аналогичен malloc

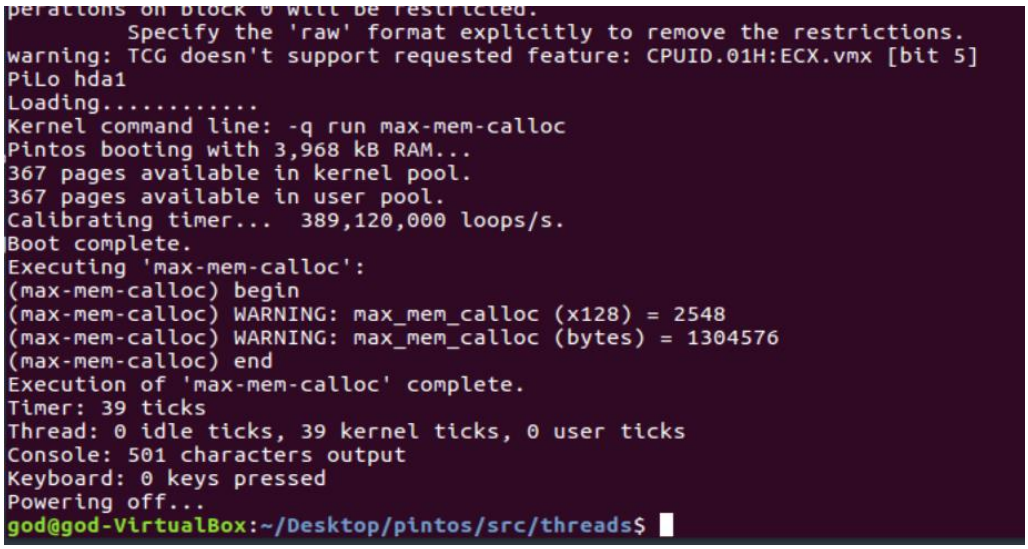
```c
void test_max_mem_calloc(void)
{
  void *vrm = calloc(128, sizeof(int));
  int max_mem_calloc = 0;
  while (vrm != NULL){
    vrm = calloc(128, sizeof(int));
    max_mem_calloc++;
  }
  msg("WARNING: max_mem_calloc (x128) = %d", max_mem_calloc);
  msg("WARNING: max_mem_calloc (bytes) = %d", max_mem_calloc * 128);

}
```

3) Реализация теста max-mem-palloc схожа по реализации, но стоит оговорить, что используется функция выделения ни конкретных объемов памяти, а сразу страницами, именно страницами, а не разрозненными ячейками памяти.

```c
void test_max_mem_palloc(void)
{
  void *vrm = palloc_get_page(PAL_USER);
  int max_mem_palloc = 0;
  while (vrm != NULL){
    vrm = palloc_get_page(PAL_USER);
    max_mem_palloc++;
  }
  msg("WARNING: max_mem_palloc (pages) = %d", max_mem_palloc);
}
```

Окончательные результаты выполнения тестов представлены ниже:

```
Automatically detecting the format is dangerous for raw images, w
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
PiLo hda1
Loading.............
Kernel command line: -q run max-mem-malloc
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  389,120,000 loops/s.
Boot complete.
Executing 'max-mem-malloc':
(max-mem-malloc) begin
(max-mem-malloc) WARNING: max_mem_malloc (x256) = 5460
(max-mem-malloc) WARNING: max_mem_malloc (bytes) = 1397760
(max-mem-malloc) end
Execution of 'max-mem-malloc' complete.
Timer: 40 ticks
Thread: 0 idle ticks, 40 kernel ticks, 0 user ticks
Console: 501 characters output
Keyboard: 0 keys pressed
Powering off...
god@god-VirtualBox:~/Desktop/pintos/src/threads$
```



Выводы по результатам теста:

Количество памяти, выделяемая malloc и calloc примерно равна, и практически полностью равна общему объему памяти в кучи ядра, palloc, выделяющий память страницами, выделил всю доступную память, а именно 367 страниц из 367.

Задание 4 (Реализуйте в src/threads.c подсчет количества тиков процессора, потраченного каждым потоком, т.е. сколько тиков каждый поток был в состоянии RUNNING. Для этого можно ввести дополнительное поле в struct thread)

Для реализации теста понадобилось внести изменения в ряд функции:

Добавления дополнительного поля в структуру thread (которая хранит значение running процесса:

```
struct thread_status
  {
    /* Owned by thread.c. */
    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. */
    int wake_up;
    int time_running;
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
```

Также был введен подсчет этого времени в функции thread_tick:

```
thread_tick (void)
{
  struct thread *t = thread_current ();


  if (t != idle_thread && t->status == THREAD_RUNNING){
    t->time_running++;
  }
  /* 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 ();
}
```

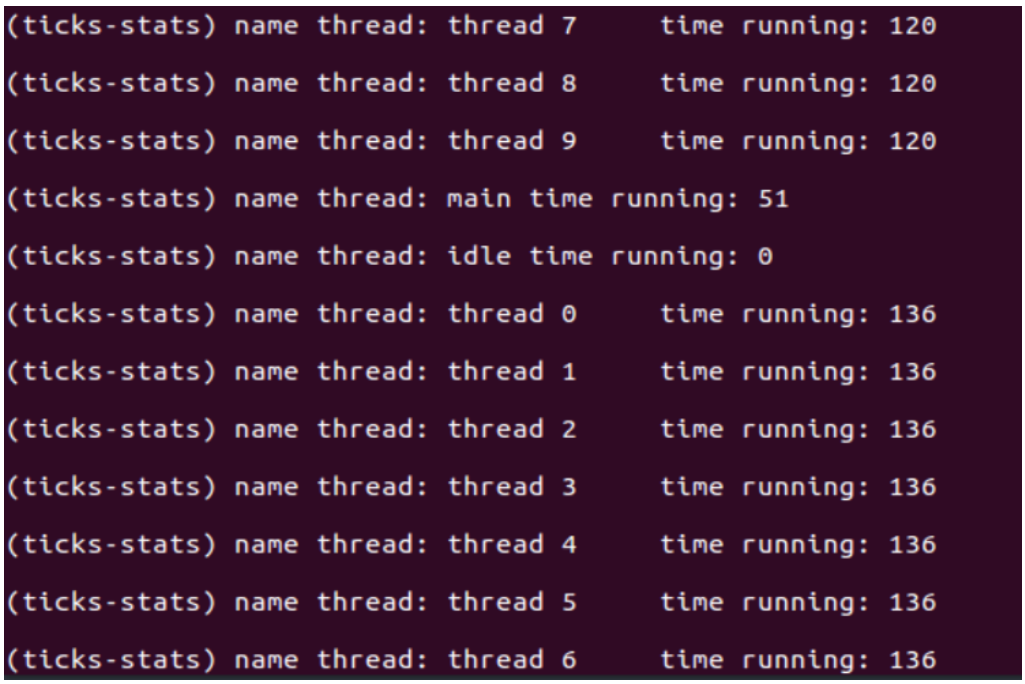
Для использования вывода результата была добавлена процедура в файл thread.c и объявлена в thread.h, связано это с тем, что необходимо обратиться к списку всех процессов all_list, который не виден в файле ticks_stats, поэтому в файле теста мы лишь обращаемся к ней:

```c
void
put_threads_from_all_list(void)
{
  struct list_elem *e;

  for (e = list_begin(&all_list); e != list_end (&all_list);
       e = list_next (e))
    {
      struct thread *f = list_entry (e, struct thread, allelem);
      msg("name thread: %s\ttime running: %d\n", f->name, f->time_running);
    }

}
```

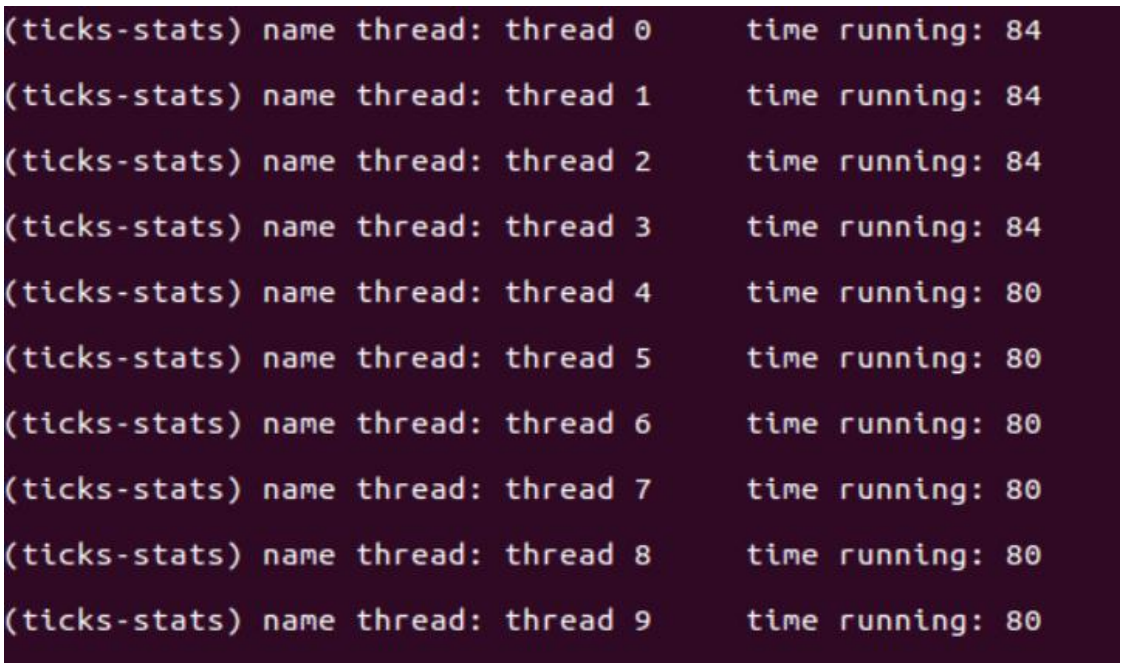
Генерация потоков реализована через цикл for, полностью аналогично с исходного тест-файла alarm- simultaneous:

```c
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/malloc.h"
#include "threads/thread.h"
#include "devices/timer.h"


void infinity_cycle(void){
  while(1 > 0){
  }
}

void test_ticks_stats(void)
{


  // thread_create("new_thread_1", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_2", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_3", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_4", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_5", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_6", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_7", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_8", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_9", PRI_DEFAULT, infinity_cycle, NULL);
  // thread_create("new_thread_10", PRI_DEFAULT, infinity_cycle, NULL);

  for (int i = 0; i < 10; i++){
    char name[16];
    snprintf (name, sizeof name, "thread %d", i);
    thread_create (name, PRI_DEFAULT, infinity_cycle, NULL);

  }

  for (int i = 0; i < 10; i++){
    timer_sleep(100);
    put_threads_from_all_list();
  }
```

Окончательные результаты выполнения теста представлены ниже:



После запуска 10 итераций мы видим плавное увеличение значений с каждым запуском 10 процессов, однако внутри итерации имеются отличия по количеству тиков, ровно на 4:



Обуславливается это стандартной системе в pintos – round robin, который изначально выделяет всем одинаковый приоритет и выделяет по 4 тика процессора на поток, именно поэтому отличия ровно в 4 тика, так как первые процессы, иногда, успевали исполниться еще раз, до наступления timer_sleep().