

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Санкт-Петербургский политехнический университет Петра Великого»

—
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА №1

ВВЕДЕНИЕ. СИСТЕМНЫЙ ТАЙМЕР

по дисциплине «Операционные системы»

Выполнил
студенты гр. 5131001/30002

Мишенев Н. С.

Руководитель
программист

<подпись>

Огнёв Р. А.

<подпись>

Санкт-Петербург
2024г.

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ	3
ХОД РАБОТЫ	4
1. Таблица функций системного таймера.....	4
2. Блок-схемы алгоритма работы системного таймера до и после модификации.	6
3. Описание внесенных модификаций в код ОС Pintos с подробными комментариями.....	8
4. Выводы тестов, полученные при их запуске.	10
ВЫВОД.....	11
ПРИЛОЖЕНИЕ А.....	12
ПРИЛОЖЕНИЕ Б.....	13

ЦЕЛЬ РАБОТЫ

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

ХОД РАБОТЫ

1. Таблица функций системного таймера.

В ходе изучения файла *timer.h* была составлена таблица определенных в нем функций, в которой описано действие самой функции и описание принимаемых на вход аргументов:

<code>void timer_init (void);</code>	Аргументов нет.	Подготавливает таймер для прерывания <code>TIMER_FREQ</code> раз в секунду с регистрацией совершенных прерываний.
<code>void timer_calibrate (void);</code>	Аргументов нет.	Калибрует значение <code>loops_per_tick</code> , используемое для создания кратковременных задержек.
<code>int64_t timer_ticks (void);</code>	Аргументов нет.	Возвращает количество тиков с момента запуска системы.
<code>int64_t timer_elapsed (int64_t then);</code>	<code>int64_t then</code> - отправная точка для отсчета тиков, значение <code>timer_ticks()</code> .	Возвращает количество тиков, которое прошло с момента <code>then</code> .
<code>void timer_sleep (int64_t ticks);</code>	<code>int64_t ticks</code> - количество тиков.	"Усыпляет" текущий процесс примерно на <code>TICKS</code> тиков. Прерывания должны быть включены.
<code>void timer_msleep (int64_t milliseconds);</code>	<code>int64_t milliseconds</code> - количество миллисекунд.	"Усыпляет" текущий процесс примерно на <code>milliseconds</code> миллисекунд. Прерывания должны быть включены.
<code>void timer_usleep (int64_t microseconds);</code>	<code>int64_t microseconds</code> - количество микросекунд.	"Усыпляет" текущий процесс примерно на <code>microseconds</code> микросекунд. Прерывания должны быть включены.
<code>void timer_nsleep (int64_t nanoseconds);</code>	<code>int64_t nanoseconds</code> - количество наносекунд.	"Усыпляет" текущий процесс примерно на <code>nanoseconds</code> наносекунд. Прерывания должны быть включены.
<code>void timer_mdelay (int64_t milliseconds);</code>	<code>int64_t milliseconds</code> - количество миллисекунд.	Использует "активное-ожидание" примерно <code>milliseconds</code> миллисекунд. Прерывания не должны быть включены. Расходует процессорные циклы.
<code>void timer_udelay (int64_t microseconds);</code>	<code>int64_t microseconds</code> - количество микросекунд.	Использует "активное-ожидание" примерно <code>microseconds</code> микросекунд. Прерывания не должны быть включены.

		Расходует процессорные циклы.
<code>void timer_ndelay (int64_t nanoseconds);</code>	<code>int64_t nanoseconds</code> - количество наносекунд.	Использует "активное-ожидание" примерно <code>nanoseconds</code> наносекунд. Прерывания не должны быть включены. Расходует процессорные циклы.
<code>void timer_print_stats (void);</code>	Аргументов нет.	Показывает статистику таймера.

2. Блок-схемы алгоритма работы системного таймера до и после модификации.

Блок схема работы алгоритма системного таймера до модификации выглядит так:

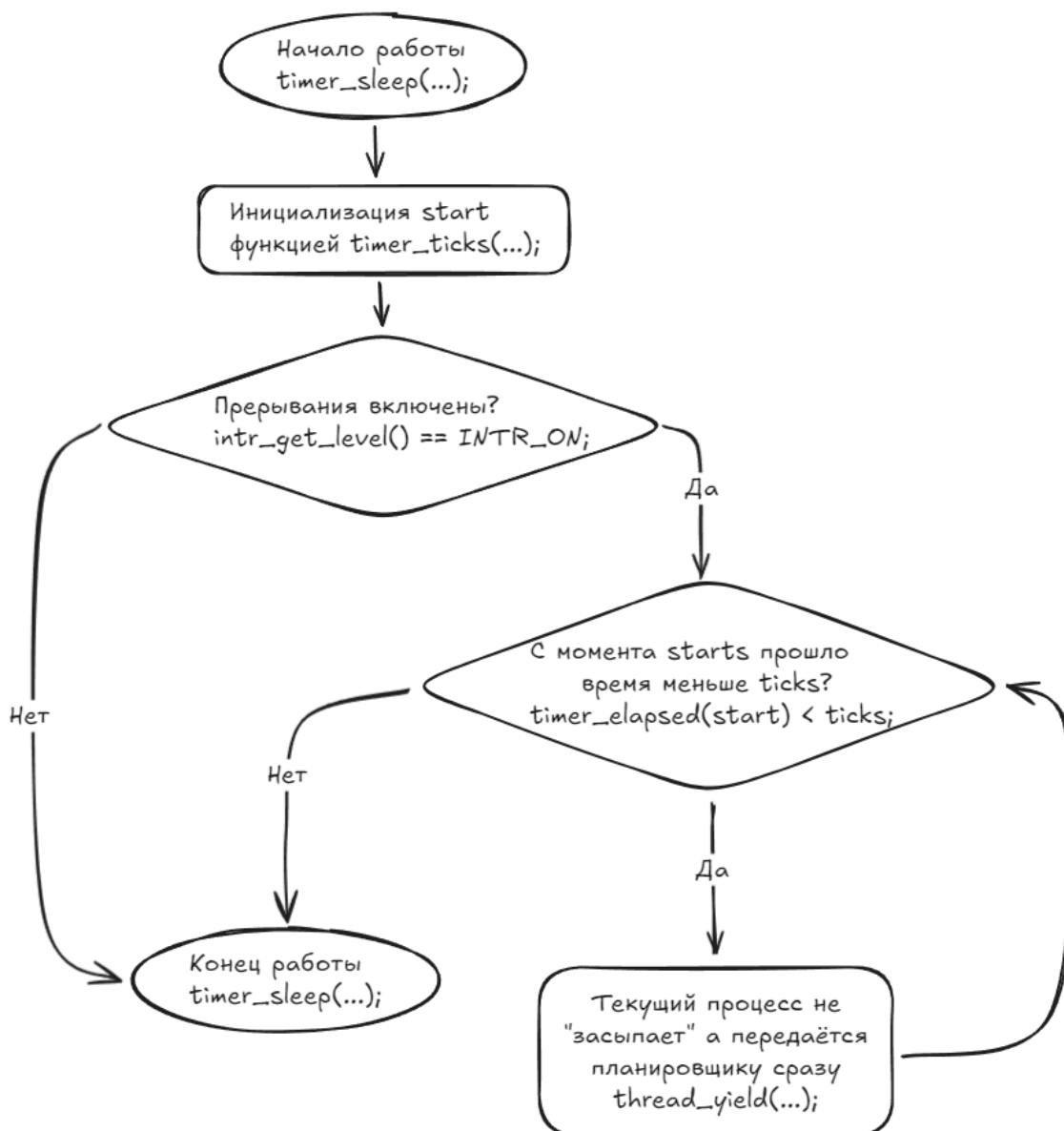


Рис. 1. Блок схема `timer_sleep()` до модификации

Исходя из этой блок-схемы, можно сказать, что алгоритм, используемый в функции `timer_sleep()`, не оптимизирован и использует активное ожидание, которое поглощает системные ресурсы, многократно проверяя одно и то же условие.

После модификации алгоритма, путём введения очереди ожидания процессов, блок-схема выглядит так:

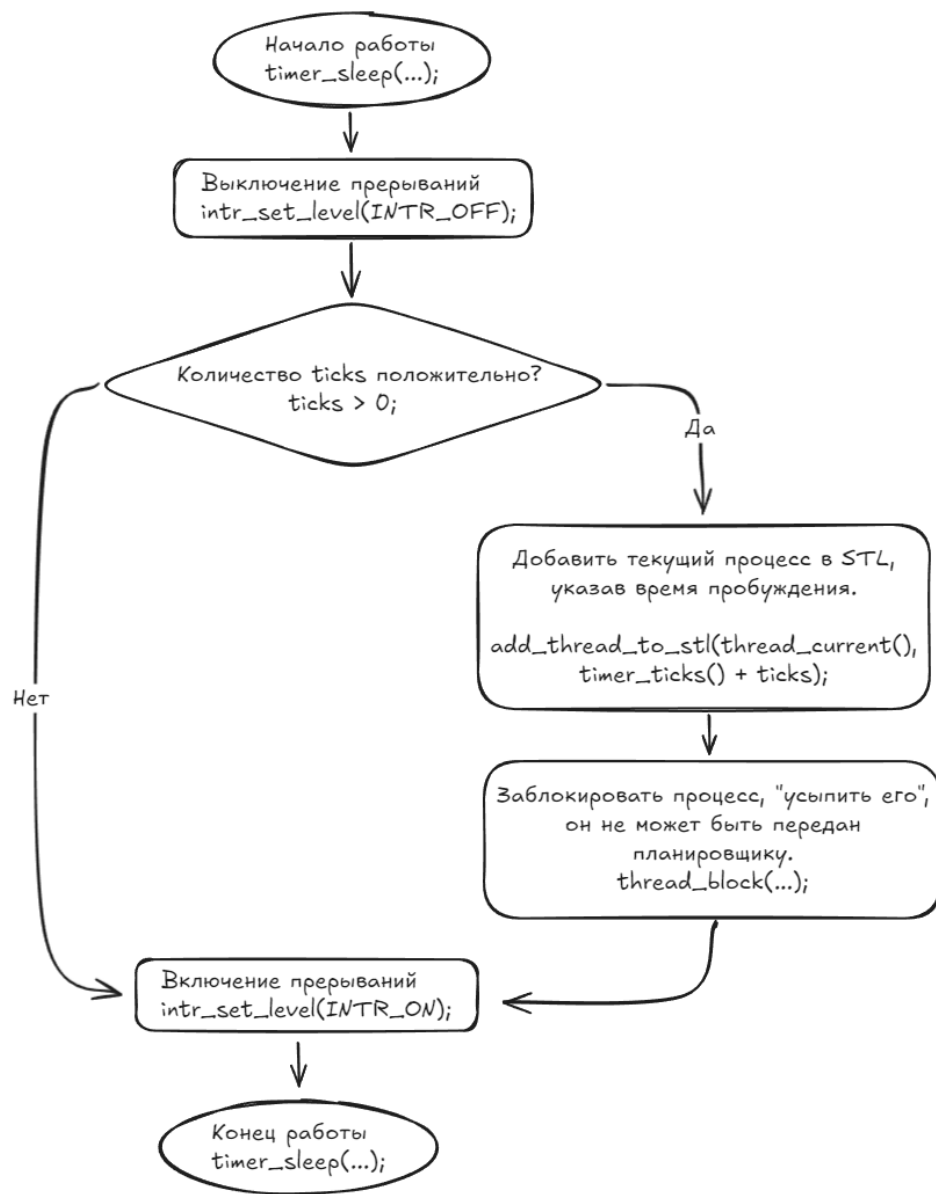


Рис. 2. Блок-схема `timer_sleep()` после модификации

Теперь, таймер не выполняет никаких многократных проверок, за счет чего, активное ожидание исключено из алгоритма его работы.

3. Описание внесенных модификаций в код ОС Pintos с подробными комментариями.

В процессе модификации исходного кода таймера, были затронуты файлы *timer.c* и *timer.h* и использована следующая идея для модификации алгоритма: при вызове таймера, для того чтобы усыпить процесс на некоторое количество тиков, процесс добавляется в *Sleeping Thread List* (далее *STL*):

```
16  //! LAB1 S  
    You, September 20th, 2024 8:28 PM | 1 author (You)  
17  struct st_list {  
18      uint64_t ticks_awake_at;  
19      struct thread* t;  
20      struct st_list* next_st;  
21  };  
22  //! LAB1 E
```

Рис. 3. Структура элемента STL в timer.h

Там, при добавлении, процесс сортируется по времени пробуждения и затем блокируется функцией *thread_block()*.

```
178      if (ticks > 0) {  
179          add_thread_to_stl(thread_current(), timer_ticks() + ticks);  
180          thread_block();  
181      }
```

Рис. 4. Добавление процесса в STL

После этого, при следующем вызове функции *timer_interrupt(...)*, отвечающей за обработку прерываний, вызовется функция *unblock_thread_from_stl()*, которая запустит процесс активного ожидания для процесса, время пробуждения которого наиболее близко к текущему, а после пробуждения, сдвинет начало списка на следующий элемент:


```

162  /* Creating an active waiting for the closest process to be
163     awoken -> for the first element of an dynamic-queue */
164  void unblock_thread_from_stl() {
165     while (stl != NULL && stl->ticks_awake_at <= ticks) {
166         thread_unblock(stl->t);
167         /*printf("thread %d unblocked\n", stl->t->tid);*/
168         stl = stl->next_st;
169     }
170 }

```

Рис. 5. Функция *unblock_thread_from_stl()*

При добавлении элемента в **STL**, используется функция *add_thread_to_stl()*, которая ищет подходящее место в списке для текущего процесса и добавляет его туда (см. Приложение Б). Она построена таким образом, что, глобальный указатель на начало списка — это процесс, который нужно будет разбудить раньше всех.

```

86
87 | /*Global STL root pointer*/
88  struct st_list* stl = NULL;
89

```

Рис. 6. Глобальный указатель на начало STL

Каждый раз при создании элемента **STL** используется функция, которая инициализирует все значения нового элемента и аллоцирует необходимое количество памяти для этого элемента.

```

105  /* Function that is allocating memory and creating
106     an array element with all data initialized */
107  struct st_list* create_elem(struct thread *t, int64_t ticks) {
108
109     struct st_list* stl_elem = (struct st_list*)calloc(1, sizeof(struct st_list));
110
111     stl_elem->ticks_awake_at = ticks;
112     stl_elem->t = t;
113     stl_elem->next_st = NULL;
114
115     return stl_elem;
116 }

```

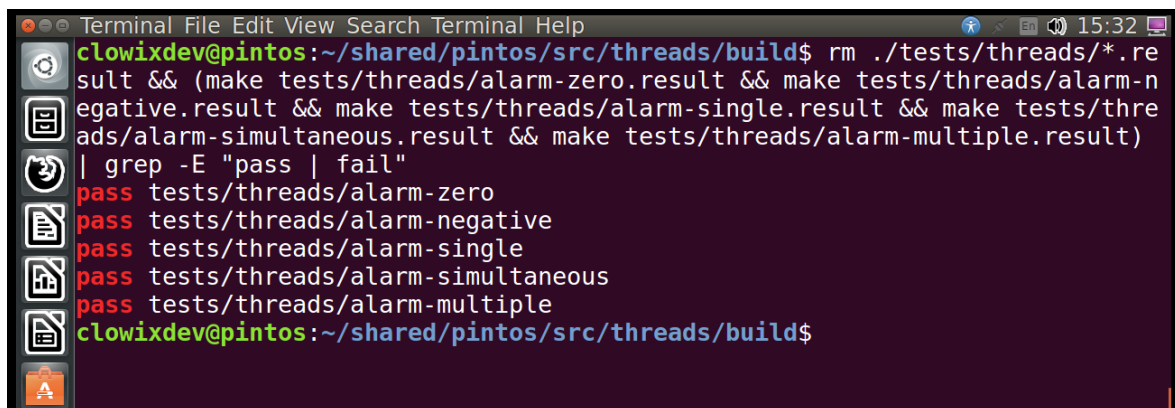
Рис. 7. Функция создания элемента STL

4. Выводы тестов, полученные при их запуске.

После внесения вышеупомянутых изменений, на Ubuntu 16.04 были запущены все необходимые тесты с помощью:

```
rm tests/threads/*.result && (make tests/threads/alarm-zero.result &&
make tests/threads/alarm-negative.result && make tests/threads/alarm-
single.result && make tests/threads/alarm-simultaneous.result && make
tests/threads/alarm-multiple.result) | grep -E "pass | fail"
```

И был получен соответствующий вывод:

A screenshot of a terminal window with a dark background. The window title is "Terminal File Edit View Search Terminal Help". The prompt is "clowixdev@pintos:~/shared/pintos/src/threads/build\$". The command entered is "rm ./tests/threads/*.result && (make tests/threads/alarm-zero.result && make tests/threads/alarm-negative.result && make tests/threads/alarm-single.result && make tests/threads/alarm-simultaneous.result && make tests/threads/alarm-multiple.result) | grep -E 'pass | fail'". The output shows five lines, each starting with "pass" in red text, followed by the test name: "tests/threads/alarm-zero", "tests/threads/alarm-negative", "tests/threads/alarm-single", "tests/threads/alarm-simultaneous", and "tests/threads/alarm-multiple". The prompt then changes to "clowixdev@pintos:~/shared/pintos/src/threads/build\$".

```
clowixdev@pintos:~/shared/pintos/src/threads/build$ rm ./tests/threads/*.result && (make tests/threads/alarm-zero.result &&
make tests/threads/alarm-negative.result && make tests/threads/alarm-single.result && make tests/threads/alarm-simultaneous.result && make tests/threads/alarm-multiple.result) | grep -E "pass | fail"
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/alarm-single
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-multiple
clowixdev@pintos:~/shared/pintos/src/threads/build$
```

Рис. 8. Результаты тестов

ВЫВОД

В ходе работы была изучена система управления процессами, а также механизм работы системного таймера в ОС Pintos, был проведен анализ его недостатков, в процессе которого выявилось использование активного ожидания и была произведена модификация алгоритма таймера, для устранения недостатка.

ПРИЛОЖЕНИЕ А

Листинг 1 - исходный код реализованных элементов на языке C (*timer.h*)

```
//! LAB1 S
struct st_list {
    uint64_t ticks_awake_at;
    struct thread* t;
    struct st_list* next_st;
};
//! LAB1 E
```

Листинг 2 - исходный код реализованных элементов на языке C (*timer.c*)

```
//! LAB 1 S

struct st_list* stl = NULL;

/* Function that shows current state of an dynamic-queue */
void show_queue(void) {
    struct st_list *stl_elem = stl;
    printf("queue: ");
    while (stl_elem != NULL) {
        printf("[%d %d %d] ",
            stl_elem->ticks_awake_at,
            stl_elem->t->tid,
            stl_elem->next_st);
        stl_elem = stl_elem->next_st;
    }
    printf("\n");
}

/* Function that is allocating memory and creating
   an array element with all data initialized */
struct st_list* create_elem(struct thread *t, int64_t ticks) {

    struct st_list* stl_elem = (struct st_list*)calloc(1, sizeof(struct st_list));

    stl_elem->ticks_awake_at = ticks;
    stl_elem->t = t;
    stl_elem->next_st = NULL;

    return stl_elem;
}

/* Function that inserts sleeping thread into a sleeping_thread_list(stl)
   taking into account at what ticks this thread should be awoken */
void add_thread_to_stl(struct thread *t, int64_t ticks) {
    struct st_list* stl_elem = create_elem(t, ticks);
    if (stl == NULL) {
        stl = stl_elem;
        /*printf("thread %d added as root\n", t->tid);
        show_queue();*/
        return;
    } else {
        struct st_list *current = stl;
        struct st_list *before = NULL;

        while (ticks >= current->ticks_awake_at) {
            before = current;
            current = current->next_st;
        }

        if (current == NULL) {
            before->next_st = stl_elem;
        }
    }
}
```

```

        /*printf("thread %d added after %d\n", t->tid, before->t->tid);
        show_queue();*/
        return;
    }
}

if (current == stl) {
    stl = stl_elem;
    stl->next_st = current;

    /*printf("thread %d added and changed root\n", t->tid);*/
} else {
    before->next_st = stl_elem;
    stl_elem->next_st = current;

    /*printf("thread %d added after %d and before %d\n",
        t->tid,
        before->t->tid,
        current->t->tid
    );*/
}

// show_queue();
}
}

/* Creating an active waiting for the closest process to be
   awoken -> for the first element of an dynamic-queue */
void unblock_thread_from_stl() {
    while (stl != NULL && stl->ticks_aware_at <= ticks) {
        thread_unblock(stl->t);
        /*printf("thread %d unblocked\n", stl->t->tid);*/
        stl = stl->next_st;
    }
}

/* Sleeps for approximately TICKS timer ticks. Interrupts must
   be turned on. */
void timer_sleep(int64_t ticks) {
    ASSERT(intr_get_level() == INTR_ON);
    intr_set_level(INTR_OFF);

    if (ticks > 0) {
        add_thread_to_stl(thread_current(), timer_ticks() + ticks);
        thread_block();
    }

    ASSERT(intr_get_level() == INTR_OFF);
    intr_set_level(INTR_ON);
}

//! LAB1 E

/* Timer interrupt handler. */
static void
timer_interrupt(struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick();
    //! LAB1 S
    unblock_thread_from_stl();
    //! LAB2 E
}

```

ПРИЛОЖЕНИЕ Б

Рис. 8. функция *add_thread_to_stl()*

```
117  /* Function that inserts sleeping thread into a sleeping_thread_list(stl)
118     taking into account at what ticks this thread should be awoken */
119  void add_thread_to_stl(struct thread *t, int64_t ticks) {
120      struct st_list* stl_elem = create_elem(t, ticks);
121      if (stl == NULL) {
122          stl = stl_elem;
123          /*printf("thread %d added as root\n", t->tid);
124             show_queue();*/
125          return;
126      } else {
127          struct st_list *current = stl;
128          struct st_list *before = NULL;
129
130          while (ticks >= current->ticks_awake_at) {
131              before = current;
132              current = current->next_st;
133
134              if (current == NULL) {
135                  before->next_st = stl_elem;
136                  /*printf("thread %d added after %d\n", t->tid, before->t->tid);
137                     show_queue();*/
138                  return;
139              }
140          }
141
142          if (current == stl) {
143              stl = stl_elem;
144              stl->next_st = current;
145
146              /*printf("thread %d added and changed root\n", t->tid);*/
147          } else {
148              before->next_st = stl_elem;
149              stl_elem->next_st = current;
150
151              /*printf("thread %d added after %d and before %d\n",
152                 t->tid,
153                 before->t->tid,
154                 current->t->tid
155                 );*/
156          }
157
158          // show_queue();
159      }
160  }
```