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.

<*подпись*>

Руководитель

программист Огнёв Р. А.

<*подпись*>

Санкт-Петербург

2024г.

**СОДЕРЖАНИЕ**

[ЦЕЛЬ РАБОТЫ 2](#_Toc1924022309)

[ХОД РАБОТЫ 2](#_Toc1495616488)

[1. Таблица функций системного таймера. 3](#_Toc1437398489)

[2. Блок-схемы алгоритма работы системного таймера до и после модификации. 4](#_Toc451617191)

[3. Описание внесенных модификаций в код ОС Pintos с подробными комментариями. 6](#_Toc1643549253)

[4. Выводы тестов, полученные при их запуске. 8](#_Toc441509122)

[ВЫВОД 9](#_Toc242310453)

[ПРИЛОЖЕНИЕ А 10](#_Toc578604424)

[ПРИЛОЖЕНИЕ Б 12](#_Toc385028170)

**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

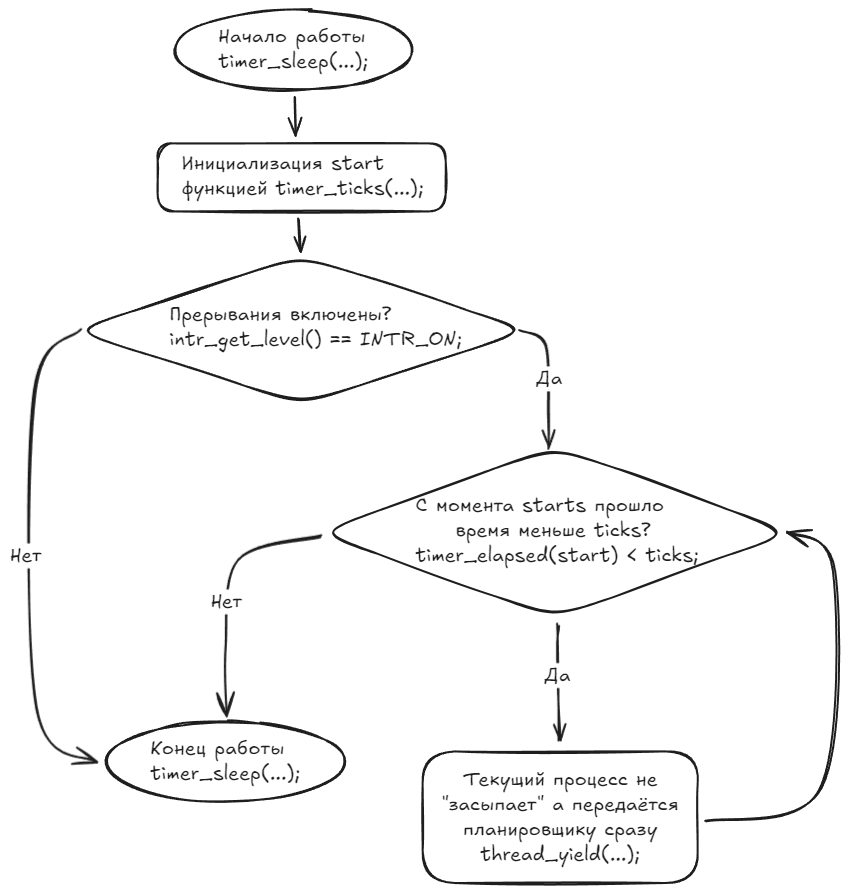
## **Таблица функций системного таймера.**

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:

|  |  |  |
| --- | --- | --- |
| void timer\_init (void); | Аргументов нет. | Подготавливает таймер для прерывания TIMER\_FREQ раз в секунду с регистрацией совершенных прерываний. |
| void timer\_calibrate (void); | Аргументов нет. | Калибрует значение loops\_per\_tick, используемое для создания кратковременных задержек. |
| int64\_t timer\_ticks (void); | Аргументов нет. | Возвращает количество тиков с момента запуска системы. |
| int64\_t timer\_elapsed (int64\_t then); | int64\_t then - отправная точка для отсчета тиков, значение timer\_ticks(). | Возвращает количество тиков, которое прошло с момента then. |
| void timer\_sleep (int64\_t ticks); | int64\_t ticks – количество тиков. | “Усыпляет” текущий процесс примерно на TICKS тиков. Прерывания должны быть включены. |
| void timer\_msleep (int64\_t milliseconds); | int64\_t milliseconds - количество миллисекунд. | “Усыпляет” текущий процесс примерно на milliseconds миллисекунд. Прерывания должны быть включены. |
| void timer\_usleep (int64\_t microseconds); | int64\_t microseconds - количество микросекунд. | “Усыпляет” текущий процесс примерно на microseconds микросекунд. Прерывания должны быть включены. |
| void timer\_nsleep (int64\_t nanoseconds); | int64\_t nanoseconds - количество наносекунд. | “Усыпляет” текущий процесс примерно на nanoseconds наносекунд. Прерывания должны быть включены. |
| void timer\_mdelay (int64\_t milliseconds); | int64\_t milliseconds - количество миллисекунд. | Использует “активное-ожидание” примерно milliseconds миллисекунд. Прерывания не должны быть включены. Расходует процессорные циклы. |
| void timer\_udelay (int64\_t microseconds); | int64\_t microseconds - количество микросекунд. | Использует “активное-ожидание” примерно microseconds микросекунд. Прерывания не должны быть включены. Расходует процессорные циклы. |
| void timer\_ndelay (int64\_t nanoseconds); | int64\_t nanoseconds - количество наносекунд. | Использует “активное-ожидание” примерно nanoseconds наносекунд. Прерывания не должны быть включены. Расходует процессорные циклы. |
| void timer\_print\_stats (void); | Аргументов нет. | Показывает статистику таймера. |

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

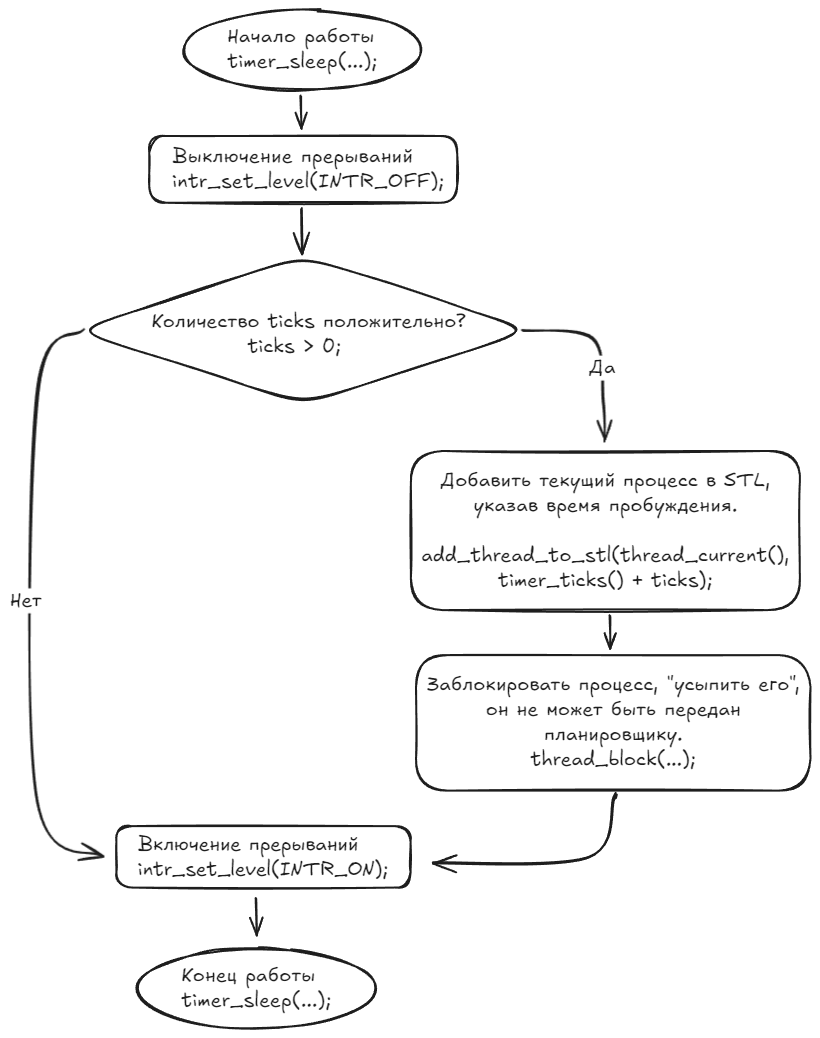
Блок схема работы алгоритма системного таймера до модификации выглядит так:



*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:

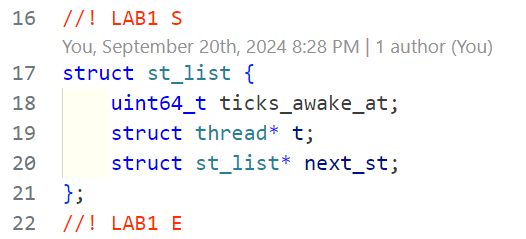


*Рис. 2. Блок-схема timer\_sleep() после модификации*

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

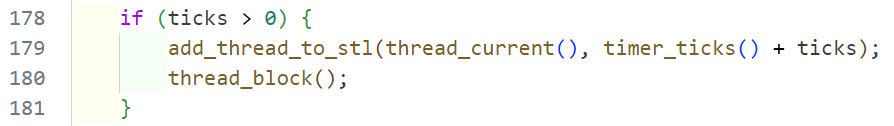
## **Описание внесенных модификаций в код ОС Pintos с подробными комментариями.**

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***):



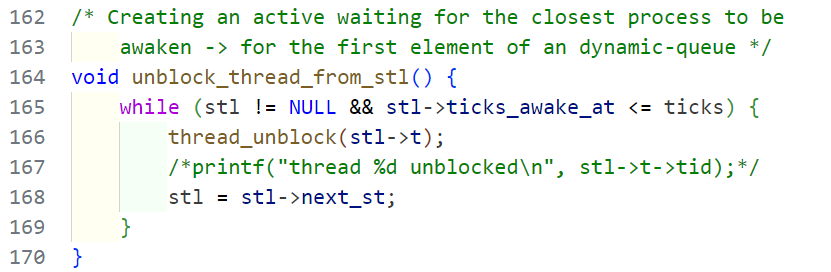
*Рис. 3. Структура элемента STL в timer.h*

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



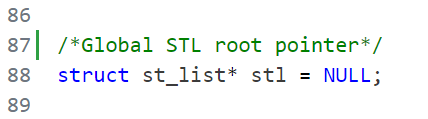
*Рис. 4. Добавление процесса в STL*

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:



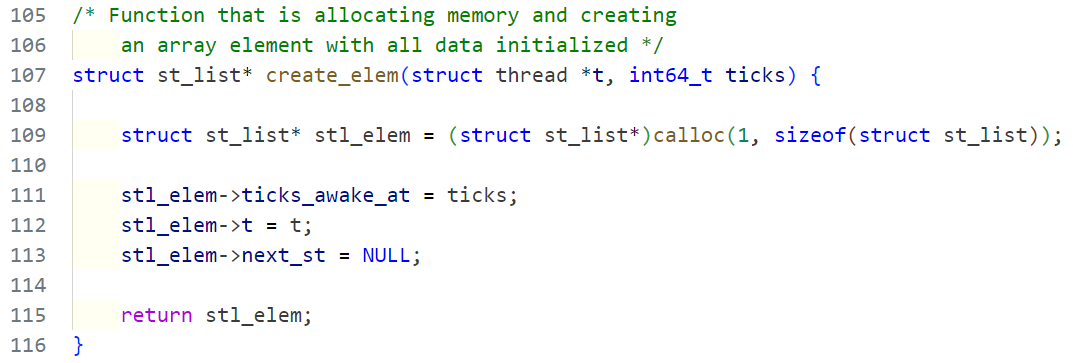
*Рис. 5. Функция unblock\_thread\_from\_stl()*

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.



*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.



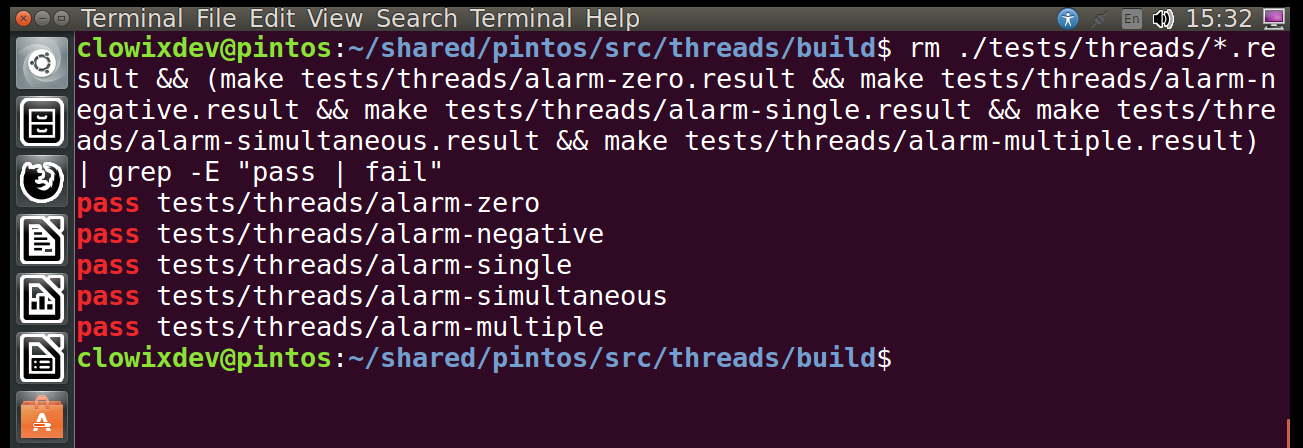
*Рис. 7. Функция создания элемента STL*

## **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:



*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 использования активного ожидания и была произведена модификация алгоритма таймера, для устранения недостатка.

**ПРИЛОЖЕНИЕ А**

Листинг 1 - исходный код реализованных элементов на языке С (***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 С (***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 awaken \*/

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

awaken -> for the first element of an dynamic-queue \*/

void unblock\_thread\_from\_stl() {

while (stl != NULL && stl->ticks\_awake\_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()***

