1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и кибербезопасности
5. **Кафедра «Высшая школа кибербезопасности»**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «**Введение. Системный таймер**»
2. по дисциплине «Операционные системы»
3. Выполнил
4. студент гр. 5131001/30003 Шевчук Н.Е.

<*подпись*>

Преподаватель Огнёв Р.А.

<*подпись*>

1. Санкт-Петербург
2. 2024

**Оглавление**

[1 Цель работы 3](#_Toc177494013)

[2 Ход работы 3](#_Toc177494014)

[2.1 Изучение работы системного таймера 3](#_Toc177494015)

[*2.2* Модификация функции *timer\_sleep()* 5](#_Toc177494016)

[2.3 Тестирование реализованного решения 8](#_Toc177494017)

[3 Вывод 8](#_Toc177494018)

# Цель работы

Изучить системы управления процессами, а также механизма работы системного таймера в ОС Pintos, проанализировать его недостатков и модифицировать его алгоритм.

# Ход работы

## Изучение работы системного таймера

С помощью текстового редактора nano просматриваем файлы timer.c и timer.h каталога devices.

|  |  |  |
| --- | --- | --- |
| Функция | Входные аргументы | Назначение |
| void timer\_init | - | Инициализирует системный таймер. Устанавливает таймер на прерывание определенное количество раз в секунду с помощью макроса TIMER\_FREQ, и регистрирует соответствующее прерывание. |
| void timer\_calibrate | - | Калибрует *loops\_per\_tick* как наибольшую степень двойки все еще меньшую одного тика таймера, используемые для реализации коротких задержек. |
| int64\_t timer\_ticks | - | Возвращает текущее количество тиков с момента запуска ОС. |
| int64\_t timer\_elapsed | int64\_t then — количество уже осуществленных тиков таймера, вычисленных с помощью timer\_ticks(), начиная с которых необходимо найти количество прошедших тиков. | Возвращает количество тиков, прошедших с момента then. |
| void timer\_sleep | int64\_t ticks — количество тиков таймера, на которое текущий поток должен быть приостановлен. | Останавливает выполнение текущего потока на указанное количество тиков таймера. Во время работы этой функциипрерывания включены, что позволяет системе выполнять другие задачи или обработку прерываний в течение периода ожидания. |
| void timer\_msleep  void timer\_usleep  void timer\_nsleep | int64\_t ms  int64\_t us  int64\_t ns - количество милли-/микро-/нано- секунд, на которое текущий поток должен быть приостановлен. | Останавливает выполнение текущего потока на указанное количество милли-/микро-/нано- секунд. Во время работы этой функциипрерывания включены, что позволяет системе выполнять другие задачи или обработку прерываний в течение периода ожидания. |
| void timer\_mdelay  void timer\_udelay  void timer\_ndelay | int64\_t ms  int64\_t us  int64\_t ns - количество милли-/микро-/нано- секунд, на которое текущий поток должен быть приостановлен. | Останавливает (откладывает) выполнение текущего потока на указанное количество милли-/микро-/нано- секунд. Прерывания могут быть отключены, что может привести к тому, что процессорбудет простаивать**,** занимаясь только ожиданием времени. |
| void timer\_print\_stats | - | Выводит на экран состояние таймера (количество тиков) |
| static void timer\_interrupt | struct intr\_frame \*args UNUSED — структура, реализующая стек прерываний | Обрабатывает (хранит) прерывания таймера. |
| static bool too\_many\_loops | unsigned loops | Проверяет, ожидают ли loops итераций менее (TRUE) или более (FALSE) тика таймера. |
| static void NO\_INLINE busy\_wait | int64\_t loops | Реализация коротких задержек с помощью повторения цикла loops раз. |
| static void real\_time\_sleep | int64\_t num  int32\_t denom — приблизительное количество секунд, на которое текущий поток должен быть приостановлен. | Останавливает выполнение текущего потока на указанное количество секунд. |
| static void real\_time\_delay | int64\_t num, int32\_t denom | Останавливает (задерживает) выполнение текущего потока на указанное количество секунд. Прерывания могут быть отключены. |

*Таблица 1. Функции timer.c*

Рассмотрим подробнее функцию timer\_sleep():

void timer\_sleep (int64\_t ticks)

{

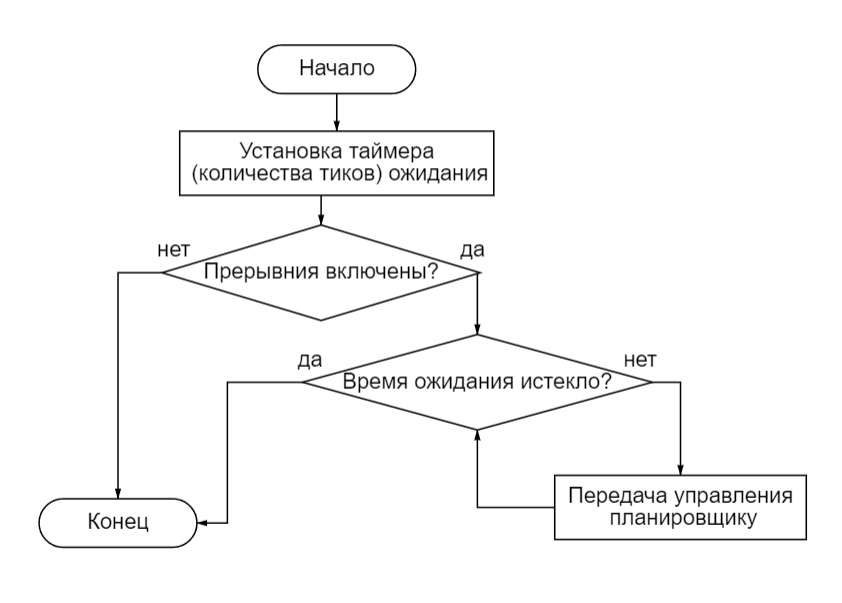
int64\_t start = timer\_ticks ();

ASSERT (intr\_get\_level () == INTR\_ON);

while (timer\_elapsed (start) < ticks)

thread\_yield ();

}



*Рисунок 1. Блок-схема функции timer\_sleep()*

Видно, что базовая (т.е. предоставленная разработчиками системы) реализация системного таймера содержит в себе цикл активного ожидания. Активное ожидание – это состояние процесса, при котором он многократно проверяет истинность некоторого условия, в данном случае, превышает ли время текущее время с момента старта установленное время ожидания. Очевидно, что такая реализация не только не выполняет никакой полезной работы, но и расходует ресурсы системы: использование активного ожидания приводит к бесполезному расходованию процессорного времени и, как следствие, снижению общей производительности системы.

## Модификация функции *timer\_sleep()*

Разработаем новый алгоритм работы системного таймера, который позволит избежать активного ожидания. Для этого внесём следующие изменения в код devices/timer.c/:

1. Создадим очередь “спящих”, т.е. готовых к исполнению и заблокированных таймером, процессов. Для этого сначала создадим структуру *struct threads\_sleep* с двумя полями: *struct thread\* thread* – указатель на поток; *int64\_t sleep\_ticks* – заданное время (в тиках) блокировки. Объявление структуры прописываем в diveces/timer.h/:

struct threads\_sleep {

struct thread\* thread;

int64\_t sleep\_ticks;

};

1. Пропишем процедуру *q\_init()*, инициализирующую массив типа *struct thread\_sleep* спящих процессов. Эта процедура будет вызываться внутри *timer\_init().*

void q\_init(void) {

q = (struct threads\_sleep\*)malloc(sizeof(struct threads\_sleep) \* 10);

ASSERT(q != NULL);

q\_capacity = 10;

q\_size = 0;

}

Также создадим глобальные переменные:

struct threads\_sleep\* q = NULL; // массив спящих процессов

size\_t q\_size = 0; // размер массива

size\_t q\_capacity = 0; // вместимость массива

1. Пропишем процедуру *q\_realloc()*, изменяющую размер динамического массива при необходимости при добавлении в очередь новых процессов.

void q\_realloc(void) {

if (q\_size >= q\_capacity) {

q\_capacity \*= 2;

q = (struct threads\_sleep\*)realloc(q, sizeof(struct threads\_sleep) \* q\_capacity);

ASSERT(q != NULL);

}

1. Пропишем функцию *push(struct threads\_sleep new\_q)*, добавляющую новый процесс в очередь так, чтоб массив остался отсортированным (сравнивая по времени блокировки).

void push(struct threads\_sleep new\_q) {

q\_realloc();

size\_t i = q\_size;

while (i > 0 && q[i - 1].sleep\_ticks > new\_q.sleep\_ticks) {

q[i] = q[i - 1];

i--;

}

q[i] = new\_q;

q\_size++;

}

1. Модифицируем функцию timer\_sleep (int64\_t ticks). Проверяем положительность времени блокировки. Затем отключаем прерывания на время выполнения функции. Вычисляем значение итогового времени блокировки (включая текущие с момента запуска ОС). Добавляем текущий поток в массив спящих процессов и блокируем его.

timer\_sleep(int64\_t ticks)

{

if (ticks <= 0) return;

enum intr\_level old\_level = intr\_disable();

int64\_t start = timer\_ticks();

int64\_t end = start + ticks;

struct threads\_sleep cur;

cur.thread = thread\_current();

cur.sleep\_ticks = end;

push(cur);

thread\_block();

intr\_set\_level(old\_level);

}

1. Пропишем процедуру *timer\_awake()*, пробуждающая процессы в порядке очереди по заданному времени блокировки. Тоже отключаем прерывания на время выполнения процедуры. Эта процедура вызывается внутри *timer\_interrupt()*.

void timer\_awake(void) {

enum intr\_level old\_level = intr\_disable();

int64\_t cur\_time = timer\_ticks();

size\_t i = 0;

while (i < q\_size && q[i].sleep\_ticks <= cur\_time) {

thread\_unblock(q[i].thread);

i++;

}

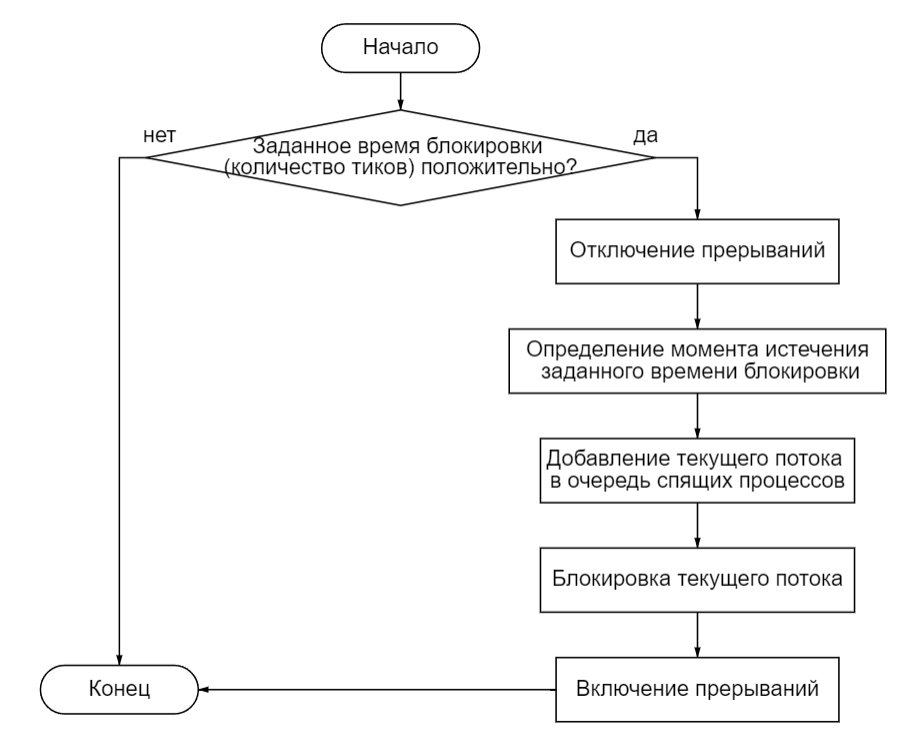
1. Добавим новые функции в код devices/timer.h/:

void q\_init(void);

void q\_realloc(void);

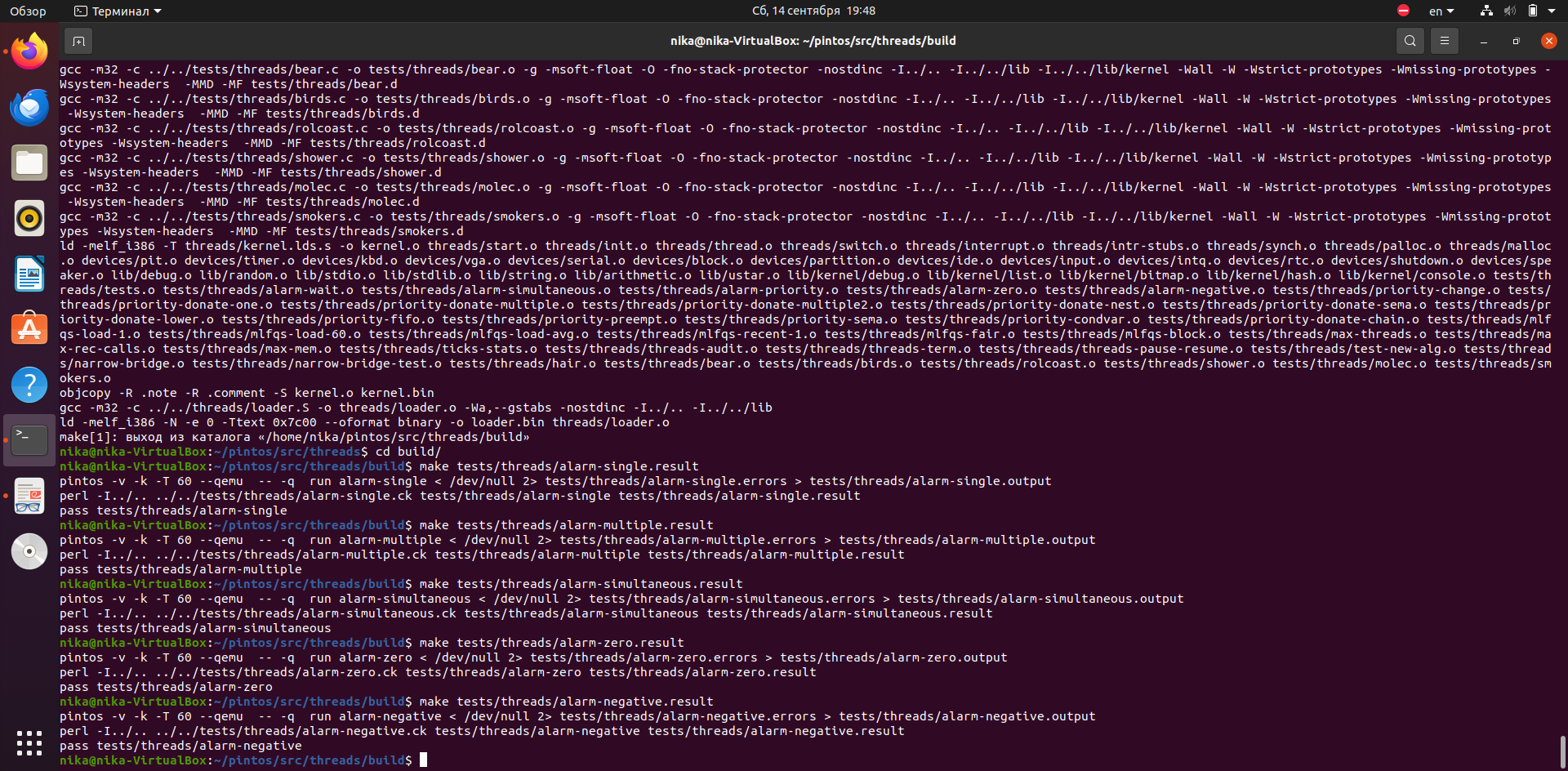
void push(struct threads\_sleep new\_q);

void timer\_awake(void);



*Рисунок 2. Блок-схема модифицированной функции timer\_sleep()*

## Тестирование реализованного решения

Для оценки модифицированного алгоритма запустим тесты alarm-single, alarm-multiple, alarm-simultaneous, alarm-zero, alarm-negative. Как видно с скриншота экрана ниже, все тесты были успешно пройдены.

*Рисунок 3. Снимок экрана ВМ с результатами тестов*

# Вывод

В ходе лабораторной работы были изучены системы управления процессами и механизм работы системного таймера в ОС Pintos. Механизм системного таймера был неэффективен, поэтому также сделана его модификация.