1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Высшая школа кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «**Системный таймер**»
2. по дисциплине «Операционные системы»
3. Выполнил
4. студент гр. 5131001/20001 Маронова К.Д.

<*подпись*>

1. Преподаватель
2. Огнёв Р.А.

<*подпись*>

1. Санкт-Петербург
2. 2023

**1. Цель работы**

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

**2. Постановка задачи**

Разработать новый алгоритм работы системного таймера, который позволит избежать активного ожидания. Программа должна удовлетворять следующим требованиям:

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 рекомендуется создать очереди готовых к исполнению и заблокированных таймером процессов, а также использовать функции, которые переводят процессы из состояния в состояние.

c) Процессы, находящиеся в такой очереди должны храниться в отсортированном виде. Очередь должна быть реализована в виде отсортированного динамического списка или массива

**3. Теоретические исследования**

**Таблица 1. Функции работы с таймером в timer.c**

|  |  |
| --- | --- |
| **Название функции** | **Описание** |
| timer\_init | Устанавливает прерывание таймера каждые TIMER\_FREQ(константа = 100) раз в секунду и регистрирует данные прерывания |
| timer\_calibrate | Калибрует loops\_per\_tick, который используется для реализации кратких задержек |
| timer\_ticks | Возвращает кол-во тиков таймера, прошедших с момента запуска системы |
| timer\_elapsed | Возвращает кол-во тиаков таймера, истёкших с then (значение, возвращаемое timer\_ticks()) |
| timer\_sleep | Усыпляет поток на заданное число тактов таймера TICKS. Прерывания должны быть включены |
| timer\_msleep | Усыпляет поток на заданное число миллисекунд. Прерывания должны быть включены |
| timer\_usleep | Усыпляет поток на заданное число микросекунд. Прерывания должны быть включены |
| timer\_nsleep | Усыпляет поток на заданное число наносекунд. Прерывания должны быть включены |
| timer\_mdelay | Поток ожидает заданное кол-во миллисекунд. Прерывания не обязаны быть включены. Ожидание тратит впустую циклы процессора, и ожидание с прерываниями, выключенными на интервал между  тактами таймерами или дольше приведет к потере тактов. |
| timer\_udelay | Аналог функции выше, но в микросекундах |
| timer\_ndelay | Аналог функции выше, но в наносекундах |
| timer\_print\_stats | Выводит статистику работы таймера |

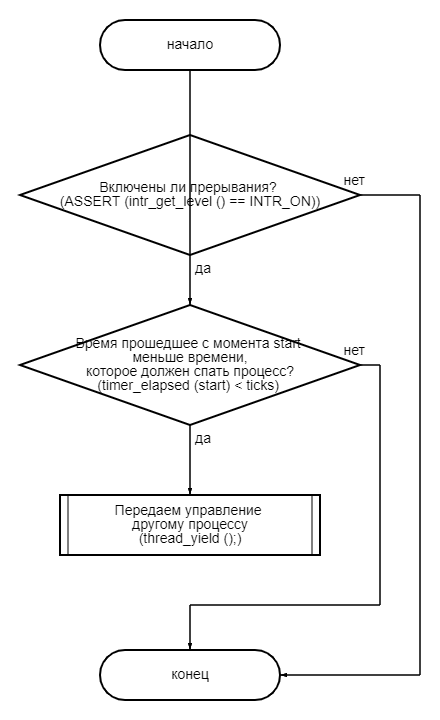
****

Рис. 1. «Блок-схема изначального алгоритма работы функции timer\_sleep() системного таймера в ОС Pintos».

**4. Описание решения**

Для модификации таймера было разработано следующее решение:

1. Необходимо добавлять потоки в спящую очередь, которая бы сортировалась по времени сна потоков.
2. Каждый поток будет разблокирован по истечении времени сна.
3. Разблокированные потоки будут добавляться в список готовых к исполнению процессов в отсортированном по приоритету порядке.

Для реализации данного решения были внесены изменения в файлы timer.c, thread.c и thread.h.

В структуру потока struct thread было добавлено 2 элемента: 1) struct list\_elem sleeping\_elem (для обращения к спящим потокам) 2) int64\_t up\_time (для фиксации времени, которое поток должен спать).

Также в thread.h было добавлено объявление двух функций: void wakeup\_sleep\_threads(int64\_t cur\_tick) и void go\_to\_bed(int64\_t wakewake). Они будут описаны далее.

В файл thread.c проводились основные модификации. Для начала необходимо было добавить список спящих потоков static struct list sleep\_queue. Для того, чтобы инициализировать этот список воспользуемся функцией thread\_init: list\_init(&sleep\_queue) (функция list\_init содержится в файле list.c и служит для создания двусвязного списка).

Все последующие модификации файла завязаны на функции thread\_tick. Сделаем так, чтобы она принимала на вход параметр (int64\_t ticks). Это необходимо, чтобы через функцию timer\_interrupt (struct intr\_frame \*args UNUSED) (в Pintos функция timer\_interrupt вызывается при каждом прерывании таймера. Прерывание таймера генерируется аппаратно через таймерный контроллер) передать значение int64\_t ticks, возвращаемое из функции timer\_ticks() (Возвращает количество тиков с момента запуска ОС). Затем в thread\_tick вызовем собственноручно написанную функцию wakeup\_sleep\_threads(ticks), которая в последующем будет пробуждать потоки, если количество тиков, на которые они должны быть заблокированы истекло.

Для пробуждения потоков используем исходную функцию thread\_unblock. Однако, для того, чтобы готовые к исполнению процессы пробуждались по приоритету добавим к функции list\_push\_back (&ready\_list, &t->elem) сортировку по приоритетам с помощью функции list\_sort из list.c. Далее потоку присваивается статус «THREAD\_READY».

В timer.c была изменена функция void timer\_sleep (int64\_t ticks). Так как для thread\_block необходимо, чтобы прерывания были отключены, то мы запоминаем прошлое состояние системы, где прерывания включены, затем отключаем их, а в конце функции возвращаем все назад, для стабильной работы ОС.

Для того, чтобы можно было отследить истекло ли время блокировки потока создадим переменную int64\_t wakewake = ticks + timer\_ticks(). Затем ее передадим в собственноручно написанную функцию void go\_to\_bed(int64\_t wakewake). Она фиксирует текущий поток с помощью thread\_current() и записывает в список спящих потоков через list\_push\_back(&sleep\_queue, &current\_thread->sleeping\_elem), а затем сортирует по времени сна (с помощью функции list\_sort из list.c).

В файл thread.c были добавлены две вспомогательные функции: sleep\_ticks\_less (сравнивает потоки по времени сна, для сортировки) и вышеупомянутая wakeup\_less (сравнивает приоритеты для сортировки).

На этом модификация таймера завершена.

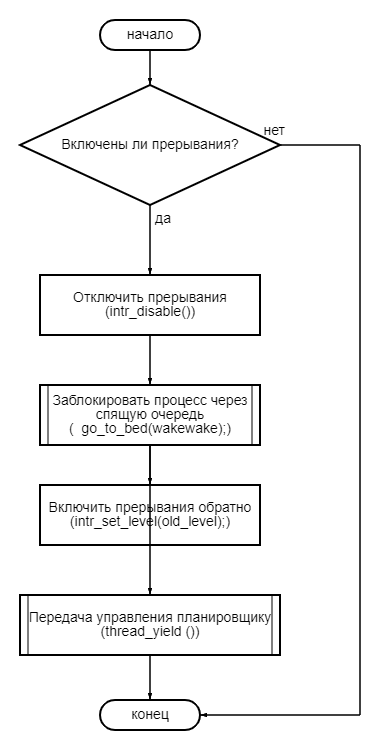


Рис. 2. «Блок-схема модифицированного алгоритма работы функции timer\_sleep() системного таймера в ОС Pintos».

**5. Тестирование и результаты работы программы**

Все необходимые тесты были успешно пройдены:

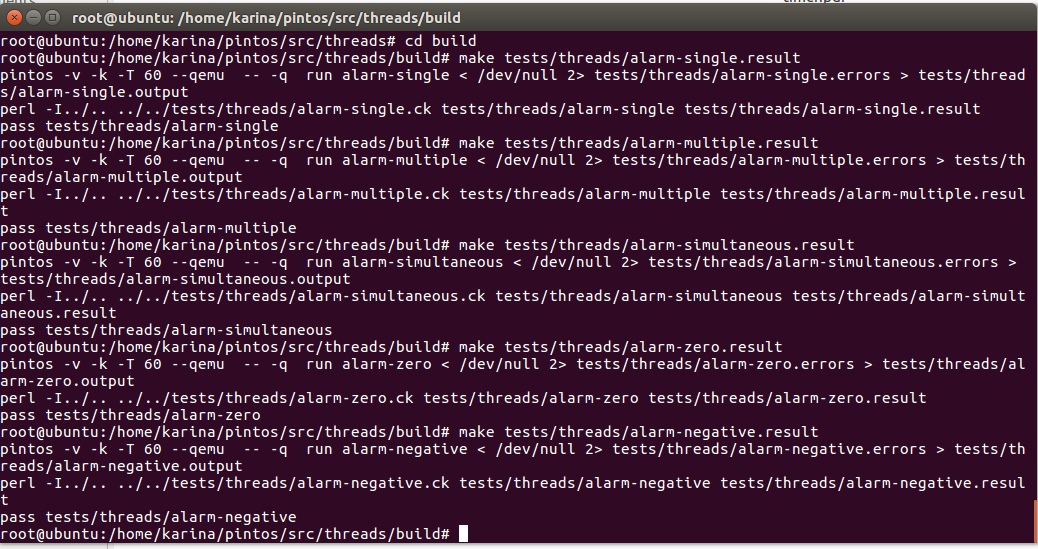


Рис. 3. «Результаты тестов».

**6. Выводы**

В процессе выполнения данной лабораторной работы были проведены эксперименты с системами контроля процессов и изучена функциональность системного таймера в операционной системе Pintos. После обнаружения недостатков системного таймера была произведена его модификация, что привело к улучшению производительности системы благодаря замене на более эффективный алгоритм «активного» ожидания.

Основные проблемы, встретившиеся в ходе работы заключались в следующем: возникали проблемы с обращениями к элементам списков через функции list\_begin, list\_next, list\_end (для решения проблемы необходимо было более детально изучить структуру функций) и добавлением элементов в список с помощью функций list\_push\_back и list\_push\_front (от которых в результате пришлось уйти, однако проблема, предположительно, решалась путем проверки наличия элементов в массиве априори).

Листинг программы:

struct thread

{

/\* 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. \*/

int64\_t up\_time; /\* время пробуждения \*/

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* Элемент списка для всех потоков. \*/

struct list\_elem sleeping\_elem; //для спящих процессов

/\* Shared between thread.c and synch.c. \*/

struct list\_elem elem; /\*Элемент списка. \*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Директория страниц. \*/

#endif

/\* Owned by thread.c. \*/

unsigned magic; /\*Проверка переполнения стека. \*/

};

static struct list sleep\_queue;

thread\_init (void)

{

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

lock\_init (&tid\_lock);

list\_init (&ready\_list);

list\_init (&all\_list);

list\_init(&sleep\_queue);

initial\_thread = running\_thread ();

init\_thread (initial\_thread, "main", PRI\_DEFAULT);

initial\_thread->status = THREAD\_RUNNING;

initial\_thread->tid = allocate\_tid ();

}

int wakeup\_less(const struct list\_elem\* a, const struct list\_elem\* b, void\* aux UNUSED) {

struct thread\* thread\_a = list\_entry(a, struct thread, elem);

struct thread\* thread\_b = list\_entry(b, struct thread, elem);

if (thread\_a->priority > thread\_b->priority) {

return 1;

}

else {

return 0;

}

}

int sleep\_ticks\_less(const struct list\_elem\* a, const struct list\_elem\* b, void\* aux UNUSED) {

struct thread\* ticks\_a = list\_entry(a, struct thread, elem);

struct thread\* ticks\_b = list\_entry(b, struct thread, elem);

if (ticks\_a->up\_time > ticks\_b->up\_time) {

return 1;

}

else {

return 0;

}

}

void go\_to\_bed(int64\_t wakewake) {

struct thread\* current\_thread = thread\_current();

current\_thread->up\_time = wakewake;

list\_push\_back(&sleep\_queue, &current\_thread->sleeping\_elem);

list\_sort(&sleep\_queue, sleep\_ticks\_less, NULL);

thread\_block();

}

void wakeup\_sleep\_threads(int64\_t cur\_tick) {

struct list\_elem\* tek\_elem = list\_begin(&sleep\_queue);;

while (tek\_elem != list\_end(&sleep\_queue)) {

struct thread\* cur\_thread = list\_entry(tek\_elem, struct thread, sleeping\_elem);

if (cur\_thread->up\_time <= cur\_tick) {

cur\_thread->up\_time = 0;

list\_remove(&cur\_thread->sleeping\_elem);

thread\_unblock(cur\_thread);

}

tek\_elem = list\_next(tek\_elem);

}

}

void

thread\_tick (int64\_t ticks) {

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++;

wakeup\_sleep\_threads(ticks);

/\* Enforce preemption. \*/

if (++thread\_ticks >= TIME\_SLICE)

intr\_yield\_on\_return ();

}

void

thread\_unblock (struct thread \*t)

{

enum intr\_level old\_level;

ASSERT (is\_thread (t));

old\_level = intr\_disable ();

ASSERT (t->status == THREAD\_BLOCKED);

list\_push\_back (&ready\_list, &t->elem);

list\_sort(&sleep\_queue, wakeup\_less, NULL);

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

void

timer\_sleep (int64\_t ticks) //кол-во тиков на которые должен быть преостановлен таймер

{

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

//фиксируем, что прерывания включены и отключаем их, чтобы потом вернуть

enum intr\_level old\_level;

old\_level = intr\_disable();

int64\_t wakewake = ticks + timer\_ticks();

go\_to\_bed(wakewake);

intr\_set\_level(old\_level);

thread\_yield (); //текущий поток передает управление другим потокам, позволяя им выполняться и вызываться по необходимости.

}