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

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «Системный таймер»
2. по дисциплине «Операционные сисетмы»
3. Выполнил
4. студент гр. 5151004/20001 Панов Д.Е.

<*подпись*>

1. Преподаватель
2. Югай П.Э.

<*подпись*>

1. Санкт-Петербург
2. 2023
3. **Цель работы**

Познакомиться с операционной системой pintos и с эмулятором qemu или bochs, а также исправить один из модулей данной операционной системы.

1. **Постановка задачи**

Разработать новый алгоритм работы системного таймера операционной системы pintos, который позволит избежать активного ожидания.

1. **Теоретические исследования**

Таблица — Функции системного таймера

|  |  |  |
| --- | --- | --- |
| **Название функции** | **Описание принимаемых аргументов** | **Описание функции** |
| void timer\_init(void) | Не принимает аргументы | Настраивает таймер на прерывание TIMER\_FREQ раз в секунду и регистрирует соответствующее прерывание. |
| void timer\_calibrate(void) | Не принимает аргументы | Калибрует loops\_per\_tick, используемый для реализации кратковременных задержек. |
| int64\_t timer\_ticks(void) | Не принимает аргументы | Возвращает количество тиков таймера с момента загрузки операционной системы. |
| int64\_t timer\_elapsed(int64\_t) | Аргумент – время, полученное при помощи timer\_ticks в любой момент работы OC | Возвращает значение времени, которое прошло с момента получения аргумента до настоящего момента |
| void timer\_sleep(int64\_t ticks) | Аргумент – время, в тиках, на которое необходимо приостановить работу таймера | При вызове начинает отсчёт времени и совершает итерации в цикле while, пока отсчитываемое время не станет меньше аргумента функции(ticks). Прерывания должны быть включены |
| void timer\_msleep (int64\_t milliseconds) | Аргумент – время, в миллисекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000). Прерывания должны быть включены |
| void timer\_usleep (int64\_t microseconds) | Аргумент – время, в микросекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000 \* 1000). Прерывания должны быть включены |
| void timer\_nsleep (int64\_t nanoseconds) | Аргумент – время, в наносекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000 \* 1000 \* 1000). Прерывания должны быть включены |
| static void real\_time\_sleep(int64\_r num, int32\_t denom) | Аргументы – время, в миллисекундах/микросекундах/наносекундах, на которое необходимо приостановить работу таймера, а также множитель | При вызове переводит миллисекунды/микросекунды/наносекунды в тики. Если тики > 0, то вызывается timer\_sleep(ticks), иначе вызывается real\_time\_delay(num, denom) |
| static void real\_time\_delay(int64\_t num, int32\_t denom) | Аргументы – время, в миллисекундах/микросекундах/наносекундах, на которое необходимо приостановить работу таймера, а также множитель | Уменьшает числитель и знаменатель на 1000 чтобы избежать переполнения, также вызывает busy\_wait(), которая повторяет простой цикл несколько раз, для реализации кратковременных задержек, чтобы результат был предсказуемым |
| void timer\_mdelay (int64\_t milliseconds) | Аргумент – время, в миллисекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000). Прерывания должны быть отключены |
| void timer\_udelay (int64\_t microseconds) | Аргумент – время, в микросекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000 \* 1000). Прерывания должны быть отключены |
| void timer\_ndelay (int64\_t nanoseconds) | Аргумент – время, в наносекундах, на которое необходимо приостановить работу таймера | При вызове вызывает функцию real\_time\_sleep() со значениями (Аргумент, 1000 \* 1000 \* 1000). Прерывания должны быть отключены |
| void timer\_print\_stats (void) | Не принимает аргументы | Выводит название таймера и время в тиках, сколько он работает |

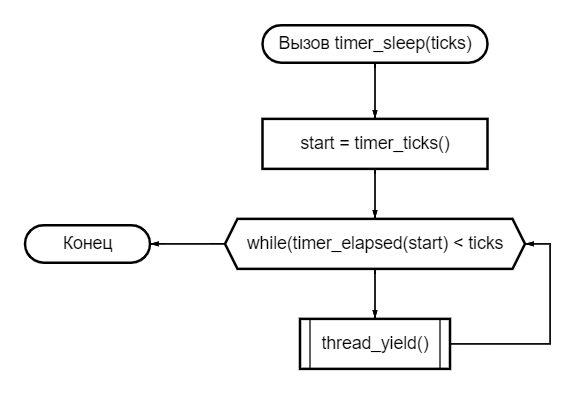


Рисунок — Блок схема текущего алгоритма работы функции timer\_sleep()

1. **Описание решения**

При разработке нового алгоритма работы системного таймера было необходимо избавиться от активного ожидания, так как оно затрачивало много ресурсов ЦП. Все изменения вносились только в файл timer.c. Первым делом я решил реализовать очередь, в которой будут храниться все спящие процессы:

struct array\_sleep\_process{

int64\_t time; // В какой момент времени надо пробудить

struct thread\* now; // Переменная для хранения данных о потоке

struct array\_sleep\_process\* next; // Указатель на след. процесс

struct array\_sleep\_process\* prev; // Указатель на пред. процесс

};

А также инициализировал корень данного списка:

struct array\_sleep\_process\* root = NULL;

Далее необходимо объявить и реализовать функцию добавления процессов в очередь, также необходимо, чтобы при добавлении в очередь они сразу сортировались по возрастанию времени, которое им осталось до пробуждения:

void insert (struct array\_sleep\_process\*\* root, struct thread\* current\_thread, int ticks);

void

insert(struct array\_sleep\_process\*\* root, struct thread\* current\_thread, int ticks)

{

int timer = ticks + timer\_ticks(); // Время в которое необходимо пробудить

struct array\_sleep\_process\* cur\_list = (struct array\_sleep\_process\*)malloc(sizeof(struct array\_sleep\_process)); // Выделение памяти для нового процесса

cur\_list->time = timer;

cur\_list->now = current\_thread;

if (\*root == NULL) { // Если корень ещё не задан

cur\_list->next = NULL;

\*root = cur\_list; // Добавляем новый процесс в корень

//printf("(debug) thread inserted in the root %lld\n", cur\_list->time);

}

else {

struct array\_sleep\_process\* tmp\_list = \*root;

while (timer > tmp\_list->time) { // Пока не найдёт процесс у которого время сна больше, чем у добавленного процесса

if (tmp\_list->next == NULL) { // Если дошёл до конца списка, то добавляет процесс в конец списка

cur\_list->next = NULL;

cur\_list->prev = tmp\_list;

tmp\_list->next = cur\_list;

//printf("(debug) thread inserted in the end %lld\n", cur\_list->time);

return;

}

tmp\_list = tmp\_list->next;

}

if (tmp\_list == (\*root)) { // Если у корня время пробуждения больше чем у добавленного процесса

(\*root) = cur\_list; // Новый процесс становится корнем

cur\_list = tmp\_list;

cur\_list->prev = (\*root);

(\*root)->next = cur\_list; // Процесс из (в прошлом) корня ставится перед добавленным процессом

//printf("(debug) thread inserted before root %lld and now he's root\n", (\*root)->time);

}

else { // Иначе добавленный процесс ставится между корнем и концом по критерию сортировки

cur\_list->prev = tmp\_list->prev;

tmp\_list->prev->next = cur\_list;

cur\_list->next = tmp\_list;

tmp\_list->prev = cur\_list;

//printf("(debug) thread inserted after root and before end %lld\n", cur\_list->time);

}

}

}

После этого необходимо изменить функцию timer\_sleep():

void

timer\_sleep (int64\_t ticks)

{

if (ticks<=0) return;

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

enum intr\_level old\_level = intr\_disable(); // Отключение прерываний

insert(&root, thread\_current(), ticks); // Вставка нового спящего процесса в список

thread\_block(); // Блокировка этого процесса

intr\_set\_level(old\_level); // Включение прерываний

}

Также необходимо объявить и реализовать функцию, которая будет убирать процесс, если его время сна закончилось:

void unblock\_sleep\_thread(int64\_t ticks);

void

unblock\_sleep\_thread(int64\_t ticks)

{

while (root != NULL && root->time <= ticks){ // Надо ли будить первый процесс из списка

thread\_unblock(root->now); // Пробуждение процесса

root = root->next; // Убираем первый процесс из списка

}

}

И добавить вызов unblock\_sleep\_thread(int64\_t ticks) в timer\_interrupt():

static void

timer\_interrupt (struct intr\_frame \*args UNUSED)

{

ticks++;

thread\_tick ();

unblock\_sleep\_thread(ticks);

}

Таким образом алгоритм работы системного таймера получился следующим: если поток необходимо погрузить в сон, то он добавляется в список спящих процессов и блокируется (процессы в списке отсортированы по возрастанию времени, в которое их необходимо разбудить), а также при каждом прерывании времени проверяется, надо ли будить первый процесс из списка.

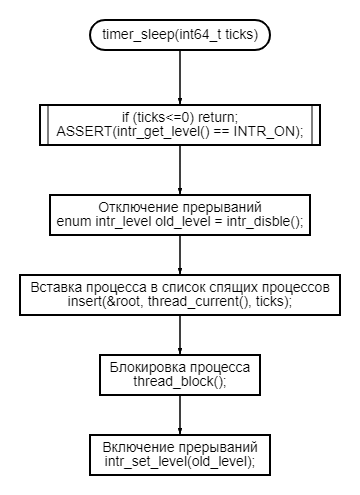


Рисунок — Блок-схема алгоритма работы системного таймера после модификации

1. **Тестирование и результаты работы программы**

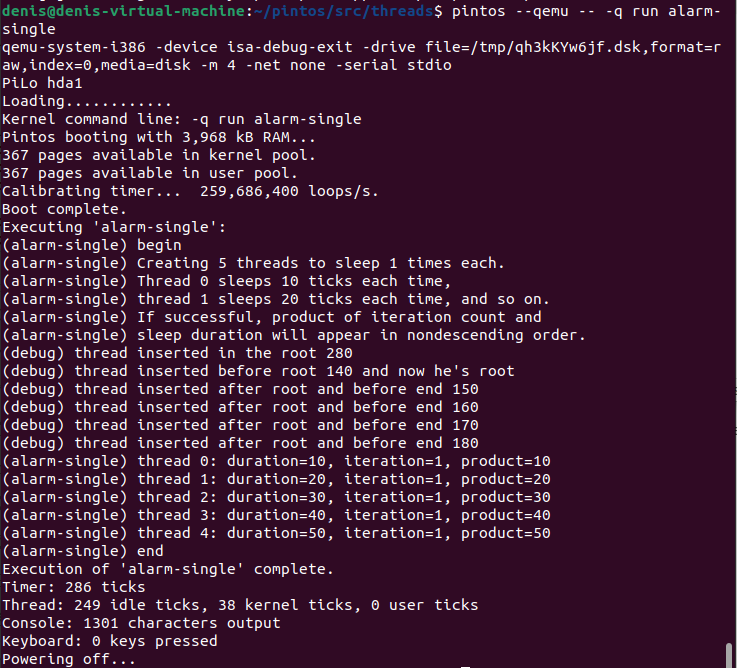


Рисунок — Запуск alarm-single

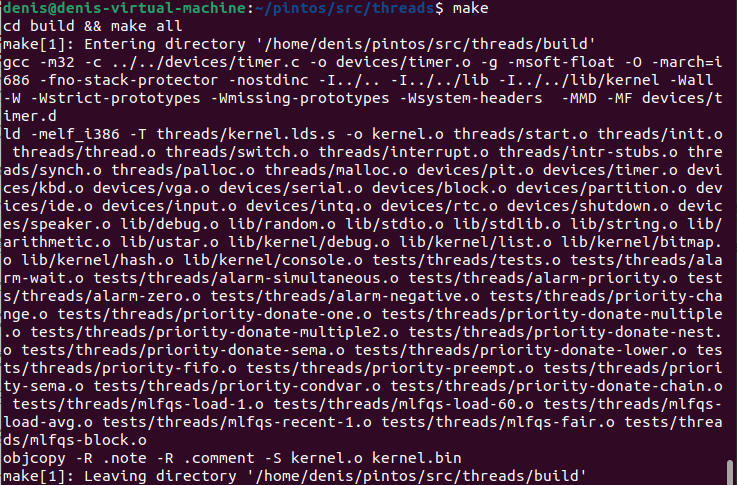


Рисунок 4 — Сборка /pintos/src/threads

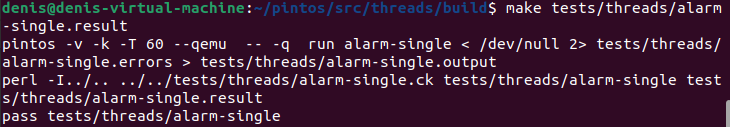


Рисунок 5 — Тест alarm-single

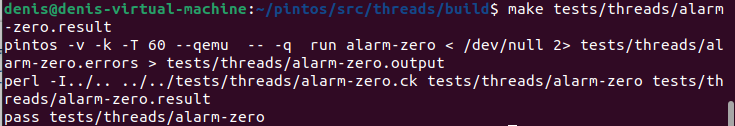


Рисунок — Тест alarm-zero

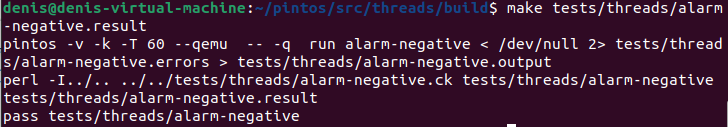


Рисунок — Тест alarm-negative

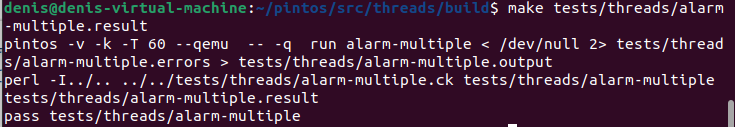


Рисунок — Тест alarm-multiple

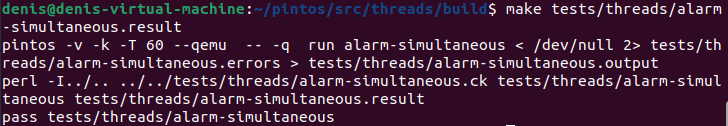


Рисунок — Тест alarm-simultaneous

1. **Выводы**

В результате данной работы была изучена реализация системного таймера в операционной системой Pintos. При анализе данного алгоритма было изучены понятия «активное ожидание», «потоки», «прерывания». Благодаря этому была разработана модификация, которая убирает активное ожидание и создает двунаправленный линейный список, в который помещаются активные потоки. Это сделано для того, чтобы была снижена нагрузка на процессор.

Список используемых источников

1. <http://ibks.spbstu.ru:3508/files/11/download>
2. <http://ibks.spbstu.ru:3508/files/5/download>
3. <http://ibks.spbstu.ru:3508/files/7/download>
4. <http://ibks.spbstu.ru:3508/files/6/download>

ПРИЛОЖЕНИЕ А

Листинг программы «timer.c»

struct array\_sleep\_process{

int64\_t time; // В какой момент времени надо пробудить

struct thread\* now; // Переменная для хранения данных о потоке

struct array\_sleep\_process\* next;

struct array\_sleep\_process\* prev;

};

struct array\_sleep\_process\* root = NULL;

void insert (struct array\_sleep\_process\*\* root, struct thread\* current\_thread, int ticks);

void unblock\_sleep\_thread(int64\_t ticks);

void

insert(struct array\_sleep\_process\*\* root, struct thread\* current\_thread, int ticks)

{

int timer = ticks + timer\_ticks();

struct array\_sleep\_process\* cur\_list = (struct array\_sleep\_process\*)malloc(sizeof(struct array\_sleep\_process));

cur\_list->time = timer;

cur\_list->now = current\_thread;

if (\*root == NULL) {

cur\_list->next = NULL;

\*root = cur\_list;

//printf("(debug) thread inserted in the root %lld\n", cur\_list->time);

}

else {

struct array\_sleep\_process\* tmp\_list = \*root;

while (timer > tmp\_list->time) {

if (tmp\_list->next == NULL) {

cur\_list->next = NULL;

cur\_list->prev = tmp\_list;

tmp\_list->next = cur\_list;

//printf("(debug) thread inserted in the end %lld\n", cur\_list->time);

return;

}

tmp\_list = tmp\_list->next;

}

if (tmp\_list == (\*root)) {

(\*root) = cur\_list;

cur\_list = tmp\_list;

cur\_list->prev = (\*root);

(\*root)->next = cur\_list;

//printf("(debug) thread inserted before root %lld and now he's root\n", (\*root)->time);

}

else {

cur\_list->prev = tmp\_list->prev;

tmp\_list->prev->next = cur\_list;

cur\_list->next = tmp\_list;

tmp\_list->prev = cur\_list;

//printf("(debug) thread inserted after root and before end %lld\n", cur\_list->time);

}

}

}

void

timer\_sleep (int64\_t ticks)

{

if (ticks<=0) return;

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

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

insert(&root, thread\_current(), ticks);

thread\_block();

intr\_set\_level(old\_level);

}

void

unblock\_sleep\_thread(int64\_t ticks)

{

while (root != NULL && root->time <= ticks){

thread\_unblock(root->now);

root = root->next;

}

}

static void

timer\_interrupt (struct intr\_frame \*args UNUSED)

{

ticks++;

thread\_tick ();

unblock\_sleep\_thread(ticks);

}