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

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. «**Планирование процессов**»
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 механизмов планирования процессов, разработка алгоритма приоритетного планирования и внедрение разработанного алгоритма в учебную операционную систему Pintos.**2. Постановка задачи**

Разработать новый алгоритм планирования, использующий для оценки важности процесса в системе значение приоритета процесса. Новый алгоритм должен отвечать следующим требованиям:

a) Каждый процесс имеет свой приоритет, который задается при его создании в функции thread\_create(). В ОС Pintos реализованы 64 уровня приоритета, по умолчанию процесс имеет приоритет 31;

b) Планировщик выбирает на исполнение процесс с наивысшим приоритетом из очереди готовых к выполнению;

c) Переключение контекста не должно происходить, если в очереди готовых к выполнению процессов нет такого процесса, приоритет которого превосходит приоритет текущего процесса;

d) Если несколько процессов в очереди готовых к выполнению имеют одинаковый приоритет, они должны быть запланированы по алгоритму Round Robin;

e) Если несколько процессов ожидают входа в критическую секцию, войти в нее первым должен процесс с наивысшим приоритетом. Соответствующие изменения должны быть внесены в реализации примитивов синхронизации ОС Pintos (synch.c);

f) Алгоритм планирования должен быть реализован оптимальным способом и должен иметь возможность не производить значительный объем вычислений при переключении процессов.

**3. Теоретические исследования**

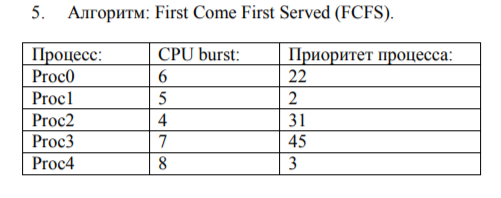


Рис. 1. «Таблица для варианта 5».

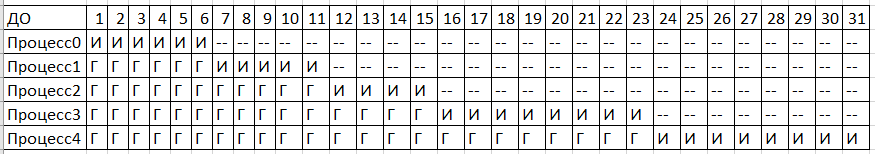


Таблица 1. «Диаграмма исполнения процессов в изначальном алгоритме»

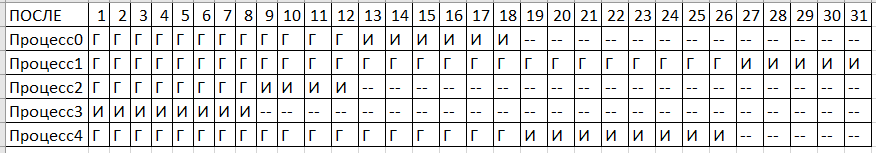


Таблица 2. «Диаграмма исполнения процессов в измененном алгоритме (с учетом приоритета и CPU burst»

Таблица 3. «Описание функций, участвующих в процессе планирования.»

|  |  |
| --- | --- |
| Название функции | Описание действий |
| sema\_down struct semaphore \*sema | Уменьшает значение SEMA, блокирует поток. |
| bool sema\_try\_down | Уменьшает значение SEMA (максимум до нуля). Возвращает значение истина, если декремент произошёл, иначе ложь. |
| void sema\_up | Увеличивает значение SEMA и будит поток, если он есть. |
| void lock\_init | Инициализация замка. |
| void lock\_acquirе | Занимает замок. Спит, пока её не вызовут. |
| bool lock\_try\_acquire | Попытка занять замок. Возвращает истину, если удачно, ложь, если нет. |
| void lock\_release | Освобождает замок. |
| bool lock\_held\_by\_current\_thread const | Возвращает значение истина, если текущий поток занимает замок. |
| void cond\_init | Инициализация монитора, для сигнализации одного участка кода другому. |
| void cond\_wait | Покидает замок и ожидает, сигнала монитора. После этого замок перезанимается перед возвратом. |
| void cond\_signal | Сигнализирует о том, что процесс должен проснуться. Замок должен быть занят перед вызовом этой функции. |
| void cond\_broadcast | Будит все потоки. |
| tid\_t thread\_create | Создаёт поток с учётом аргументов, переданных в функцию. |
| void thread\_block | Отправляет данный поток в сон. |
| void thread\_unblock | Будит поток. |
| void thread\_exit | Уничтожает поток. |
| void thread\_yield | Поток не спит и ждёт, когда его выберет планировщик. |
| void idle | Запускает пустой поток, когда нет других. |
| struct thread \*running\_thread | Возвращает запущенный поток. |
| static bool is\_thread | Проверка, что поток - валидный |
| static void init\_thread | Инициализация потока. |
| static struct thread \*next\_thread\_to\_run | Возвращает поток из очереди, который должен быть запланирован, пока очередь не опустеет. |
| void thread\_schedule\_tail | Переключает между потоками, уничтожает поток, который умер. |
| static void schedule | Планирование нового процесса. |

**4. Описание решения**

Для разработки алгоритма приоритетного планирования были выполнены 3 основные задачи:

1. Реализовать приоритетное планирование:

* Переключение только при наличии процесса с большим приоритетом
* Процессы с одинаковым приоритетом планируются по RR
* Учитывать приоритет при входе в критическую секцию

1. Внести изменения в работу примитивов синхронизации путем учета приоритетов процессов
2. Реализация механизма жертвования:

* Когда процесс пытается захватить ресурс
* Проанализировать ситуацию
* Пожертвовать приоритет всем нуждающимся процессам
* Когда процесс освобождает ресурс
* Вернуть исходный приоритет

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

Модификации thread.h:

Необходимо было добавить 3 поля в структуру потока: old\_priority – для возвращения потоку его исходного приоритета, list locks\_queue - очередь локов, которые держит данный поток, lock\* lock\_to\_wait - лок, который поток ожидает и хочет заполучить в дальнейшем (тоже нужно для пожертвований).

Для реализации задуманного алгоритма было необходимо получать доступ к списку ready\_list в нескольких файлах сразу. Для этого данный список был объявлен в thread.h, а в остальных файлах, где список нужен, происходит объявление через extern.

Модификации thread.c:

В некоторые места (init\_thread, thread\_create) были добавлены инициализации новых переменных из thread.h. Также при создании нового потока мы сравниваем его приоритет с приоритетом thread\_current() и если новый поток имеет больший приоритет, то мы сменяем процессы при помощи функции thread\_yield.

Глобально была изменена функция thread\_set\_priority (int new\_priority). В неё так же была добавлена инструкция смены потока при определенных выше условиях. Данная функция используется в случаях, когда потоку в какой-то момент времени переназначается приоритет. Важно назначить new\_priority именно в графу старого приоритета, для корректной работы программы.

Модификации synch.h и synch.c:

Для учета приоритета в программе было добавлено поле приоритета(потока) во все структуры (кроме cond). Это было сделано для некоторых сортировок уже имеющихся списков. В некоторых случаях данная графа нужна для обращения к приоритету потока через структуру элемента синхронизации (так как неясно, как получить приоритет потока иным способом).

Для получения доступа к критической секции через приоритет был отсортирован список cond->waiters, исходя из приоритета потоков, которые претендуют на семафор.

Затем были произведены модификации функций lock\_acquire и lock\_release.

В функции lock\_acquire происходит попытка завладеть определенным локом. Если им уже кто-то владеет, то в цикле производятся действия, которые в последствии освободят необходимый нам лок. После чего его владельцем назначается thread\_current(). Попутно мы жертвуем приоритеты, чтобы у потоков с маленьким приоритетом тоже был шанс выполниться.

Функция lock\_release() выполняет основную операцию по освобождению блокировки. Она устанавливает поле holder блокировки в NULL, увеличивает семафор блокировки с помощью sema\_up() и удаляет блокировку из списка блокировок текущего потока. Затем функция проверяет, если список блокировок текущего потока не пустой, то сортирует его по приоритету и сравнивает максимальный приоритет блокировки с приоритетом текущего потока. Если максимальный приоритет блокировки выше приоритета текущего потока, то выполняется переключение потоков с помощью функции thread\_yield(). Если список блокировок текущего потока пустой, то функция возвращает ему его старый приоритет и также проверяет, если есть более высокоприоритетный поток в готовом списке, то выполняется переключение потоков с помощью thread\_yield().

На этом модификация программы завершена.

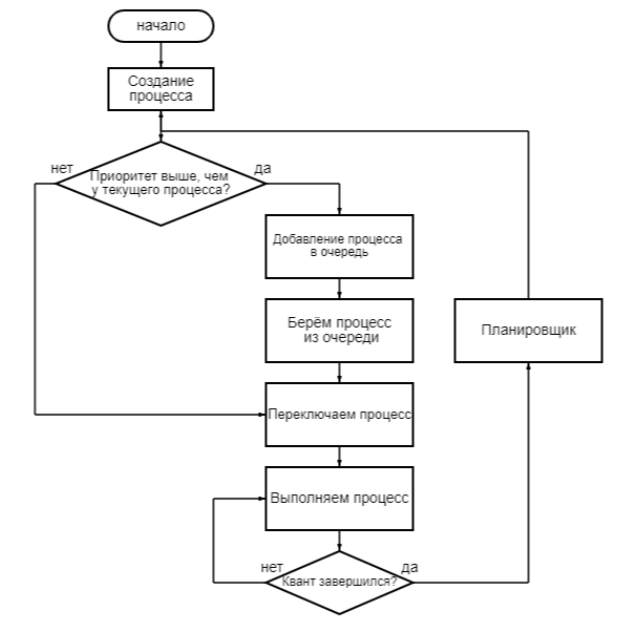


Рис. 2. «Блок-схема модифицированного алгоритма».

**5. Тестирование и результаты работы программы**

Все необходимые тесты были успешно пройдены:

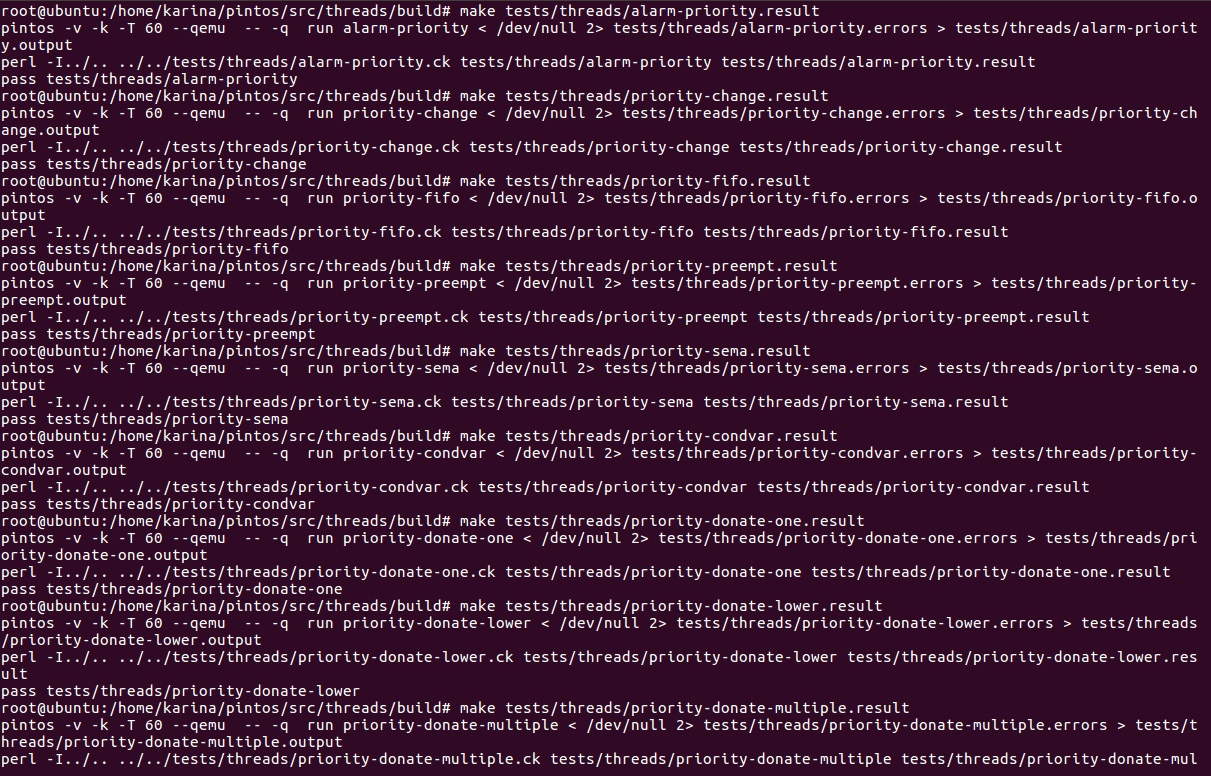
  


Рис. 3. «Результаты тестов».

**6. Выводы**

В результате выполнения данной лабораторной работы разобраны механизмы планирования процессов, модифицирован алгоритм приоритетного планирования в OS Pintos. Теперь процессы выполняются не по порядку, а по величине приоритетов с учётом квантования.

Основные проблемы, встретившиеся в ходе работы заключались в следующем: 1) При проходе по ready\_list была попытка использования функции list\_max, чтобы взять поток с наивысшим приоритетом на выполнение. Однако, это ломало правильный ход выполнения программы, поскольку максимальный приоритет может быть у 2х идущих подряд потоков. Но мы должны выполнять их в порядке поступления (если их приоритеты одинаковые), следовательно, такое обращение оказалось ошибочным. 2) Если какой-либо из тестов не проходил, то производился анализ того, что выводит наша программа по факту и того, что она должна выводить на самом деле. Исходя из этого были произведены изменения логики действий, чтобы достичь необходимого результата.

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

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

int old\_priority; //старый приоритет потока для донатов в семафоре

struct list locks\_queue; //очередь локов, которые держит данный поток

struct lock\* lock\_to\_wait; //лок, который поток ожидает и хочешь заполучить в дальнейшем

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\* 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. \*/

};

struct list ready\_list;

void

thread\_set\_priority (int new\_priority)

{

//thread\_current ()->priority = new\_priority;

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

if (current->priority == current->old\_priority) {

current->priority = new\_priority;

current->old\_priority = new\_priority;

}

else {

current->old\_priority = new\_priority;

}

//Сменяемость потоков из рейди листа

if (!list\_empty(&ready\_list)) {

struct list\_elem\* e = list\_begin(&ready\_list);

struct thread\* tmp = list\_entry(e, struct thread, elem);

if (tmp != NULL) {

if (tmp->priority > new\_priority) {

thread\_yield();

}

}

}

}

tid\_t

thread\_create (const char \*name, int priority,

thread\_func \*function, void \*aux)

{

struct thread \*t;

struct kernel\_thread\_frame \*kf;

struct switch\_entry\_frame \*ef;

struct switch\_threads\_frame \*sf;

tid\_t tid;

//enum intr\_level old\_level;

ASSERT (function != NULL);

/\* Allocate thread. \*/

t = palloc\_get\_page (PAL\_ZERO);

if (t == NULL)

return TID\_ERROR;

/\* Initialize thread. \*/

init\_thread (t, name, priority);

/\*list\_init(&t->locks\_queue);\*/

tid = t->tid = allocate\_tid ();

/\* Подготовить поток для первого выполнения, инициализируя его стек.

Сделать это атомарно, чтобы промежуточные значения члена 'stack' не могли быть наблюдаемыми. \*/

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

/\* Stack frame for kernel\_thread(). \*/

kf = alloc\_frame (t, sizeof \*kf);

kf->eip = NULL;

kf->function = function;

kf->aux = aux;

/\* Stack frame for switch\_entry(). \*/

ef = alloc\_frame (t, sizeof \*ef);

ef->eip = (void (\*) (void)) kernel\_thread;

/\* Stack frame for switch\_threads(). \*/

sf = alloc\_frame (t, sizeof \*sf);

sf->eip = switch\_entry;

sf->ebp = 0;

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

/\* Add to run queue. \*/

thread\_unblock (t);

struct thread\* current\_thread = thread\_current(); //Если приоритет нового потока больше, чем текущего, то сменяем потоки

if (priority >= current\_thread->priority) {

thread\_yield();

}

return tid;

}

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); //ПОМЕНЯТЬ ЭТУ ФУНКЦИЮ НА ТУ, КОТОРАЯ БУДЕТ СОРТИРОВАТЬ ИХ В РЕЙДИ ЛИСТ ПО ПРИОРИТЕТУ

//list\_insert\_ordered(&ready\_list, &t->elem, wakeup\_less , NULL);

list\_sort(&ready\_list, wakeup\_less, NULL);

t->status = THREAD\_READY;

/\* проверка на то, что приоритет разблокированного потока больше, чем у текущего. Если так -> сменяй потоки \*/

if (thread\_current() != idle\_thread && thread\_current()->priority < t->priority) {

thread\_yield();

}

intr\_set\_level (old\_level);

}

static void

init\_thread (struct thread \*t, const char \*name, int priority)

{

enum intr\_level old\_level;

ASSERT (t != NULL);

ASSERT (PRI\_MIN <= priority && priority <= PRI\_MAX);

ASSERT (name != NULL);

memset (t, 0, sizeof \*t);

t->status = THREAD\_BLOCKED;

strlcpy (t->name, name, sizeof t->name);

t->stack = (uint8\_t \*) t + PGSIZE;

list\_init(&t->locks\_queue);

t->priority = priority;

t->old\_priority = priority;

t->lock\_to\_wait = NULL;

t->magic = THREAD\_MAGIC;

old\_level = intr\_disable();

list\_push\_back (&all\_list, &t->allelem);

intr\_set\_level(old\_level);

}

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (!lock\_held\_by\_current\_thread (lock));

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

struct lock\* tmp\_lock = lock;

//struct thread\* cur\_thread = thread\_current();

struct thread\* lock\_first\_holder = lock->holder;

thread\_current()->lock\_to\_wait = lock;

// Проверка, что lock еще не захвачен ни одним потоком

if (lock\_first\_holder == NULL) {

// Установка приоритета lock равным приоритету текущего потока

tmp\_lock->lock\_priority = thread\_current()->priority;

}

if (lock\_first\_holder != NULL) {

// Цикл повторяется до тех пор, пока lock не будет освобожден и \

приоритет текущего потока будет выше приоритета текущего владельца лока1

while (lock\_first\_holder != NULL && thread\_current()->priority > lock\_first\_holder->priority) { //попробовать дописать првоерку что кур\_тред!=лок->локхолдер

lock\_first\_holder->priority = thread\_current()->priority;

// Проверка, является ли текущий поток владельцем lockа и при этом список готовых потоков не пуст \

подсасываем приоритет большего

if (lock\_first\_holder == thread\_current() && !list\_empty(&ready\_list)) {

struct list\_elem\* thread\_elem = list\_begin(&ready\_list);

struct thread\* max\_thread = list\_entry(thread\_elem, struct thread, elem);

if (max\_thread!=NULL && max\_thread->priority > thread\_current()->priority) {

thread\_yield();

}

}

//фактически lock->priority - это наибольший приоритет среди трдов, которые им владеют/владели. Так что если \

приоритет текущего потока больше, то донатим локу его приоритет

if (tmp\_lock->lock\_priority < thread\_current()->priority) {

tmp\_lock->lock\_priority = thread\_current()->priority;

}

//переменная tmp\_lock обновляется как блокировка, которую ждал предыдущий владелец блокировки, чтобы передать\

"владение".Если значение tmp\_lock не равно NULL, то владелец блокировки также обновляется как владелец tmp\_lock.\

Если же tmp\_lock равно NULL, то цикл прерывается, так как ожидаемая блокировка освободилась.

tmp\_lock = lock\_first\_holder->lock\_to\_wait; //обнавляем tmp\_lock как лок, который ждал владелец нашего изначального лока, чтобы передать звание "владельца"

if (tmp\_lock != NULL) {

lock\_first\_holder = tmp\_lock->holder;

}

else {

break; // Прерывание цикла, если tmp\_lock равен NULL, то есть ожидаемый лок освободился

}

}

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

sema\_down (&lock->semaphore);

lock->holder = thread\_current ();

lock->holder->lock\_to\_wait = NULL;

//закидываем текущий лок в список локов, которыми владеет текущий поток

list\_push\_back(&(lock->holder->locks\_queue), &(lock->lock\_list\_element));

list\_sort(&(lock->holder->locks\_queue), priority\_sort\_L, NULL);

}

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (lock\_held\_by\_current\_thread (lock));

struct thread\* curr = thread\_current();

lock->holder = NULL;

sema\_up (&lock->semaphore);

list\_remove(&lock->lock\_list\_element);

if (!list\_empty(&thread\_current()->locks\_queue)) {

list\_sort(&(thread\_current()->locks\_queue), priority\_sort\_L, NULL);

struct list\_elem\* elem\_front = list\_front(&(thread\_current()->locks\_queue));

struct lock\* first = list\_entry(elem\_front, struct lock, lock\_list\_element);

//подсасываем для нашего текущего thread\_current приоритет, которым обладает максимальный лок в его владениях \

если он меньше, чем какой-то в рейди лсте, то сменяем потоки

thread\_current()->priority = first->lock\_priority;

if (!list\_empty(&ready\_list)) {

struct list\_elem\* thread\_elem = list\_begin(&ready\_list);

struct thread\* max\_thread = list\_entry(thread\_elem, struct thread, elem);

if (max\_thread != NULL && max\_thread->priority > first->lock\_priority){

thread\_yield();

}

}

}

else {

//если список владений треда пуст, то мы просто возвращаем ему его старый приоритет. и так же сменяем поток, если в рейди листе есть круче

thread\_current()->priority = thread\_current()->old\_priority;

if (!list\_empty(&ready\_list)) {

struct list\_elem\* thread\_elem = list\_begin(&ready\_list);

struct thread\* max\_thread = list\_entry(thread\_elem, struct thread, elem);

if (max\_thread!=NULL && max\_thread->priority > thread\_current()->old\_priority) {

thread\_yield();

}

}

}

}

void

cond\_wait (struct condition \*cond, struct lock \*lock)

{

struct semaphore\_elem waiter;

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

sema\_init (&waiter.semaphore, 0);

waiter.semaphore.thread\_priority = thread\_current()->priority;

list\_push\_back (&cond->waiters, &waiter.elem);

/\* сортировка используется для правильного упорядочивания потоков, ожидающих условия,

по их приоритетам. Это позволяет выбрать поток с самым высоким приоритетом для выполнения, когда условие будет выполнено. \*/

list\_sort(&cond->waiters, priority\_sort\_S, NULL);

lock\_release (lock);

sema\_down (&waiter.semaphore);

lock\_acquire (lock);

}