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 № 2**

1. «ПЛАНИРОВАНИЕ ПРОЦЕССОВ»
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](#_Toc183701099)

[2 Ход работы 3](#_Toc183701100)

[2.1 Текущий алгоритм планирования 3](#_Toc183701101)

[2.2 Алгоритм приоритетного планирования 4](#_Toc183701102)

[2.3 Алгоритм разделения приоритета 9](#_Toc183701103)

[2.4 Результаты тестов 12](#_Toc183701104)

[2.5 Тест test-new-alg 20](#_Toc183701105)

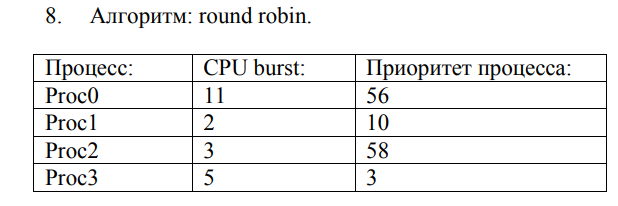
[3 Вывод 23](#_Toc183701106)

# Цель работы

Изучить механизмы планирования процессов, разработать алгоритм приоритетного планирования и внедрить разработанный алгоритм в учебную операционную систему Pintos.

# Ход работы

## Текущий алгоритм планирования



*Рисунок 1. Исходные данные по процессам*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Такты | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| Proc0 | и | и | и | и | г | г | г | г | г | г | г | г | г | и | и | и | и | г | и | и | и |
| Proc1 | г | г | г | г | и | и | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Proc2 | г | г | г | г | г | г | и | и | и | - | - | - | - | - | - | - | - | - | - | - | - |
| Proc3 | г | г | г | г | г | г | г | г | г | и | и | и | и | г | г | г | г | и | - | - | - |

*Таблица 1. Диаграмма исполнения процессов при изначальном алгоритме планировщика*

Оценку эффективности алгоритма планирования проводят по двум показателям – среднее время ожидания и среднее время исполнения.

Среднее время ожидания:

Среднее время исполнения:

Из диаграммы видно, что при текущем алгоритме не учитывается приоритет процессов. При модификации алгоритма планирования (добавления учета приоритета) изменится порядок исполнения процессов:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Такты | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| Proc0 | г | г | г | и | и | и | и | г | г | г | г | г | г | и | и | и | и | г | и | и | и |
| Proc1 | г | г | г | г | г | г | г | и | и | - | - | - | - | - | - | - | - | - | - | - | - |
| Proc2 | и | и | и | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Proc3 | г | г | г | г | г | г | г | г | г | и | и | и | и | г | г | г | г | и | - | - | - |

*Таблица 2. Диаграмма исполнения процессов при модифицированном алгоритме планировщика*

Среднее время ожидания:

Среднее время исполнения:

Благодаря приоритетному планированию уменьшилось среднее время ожидания. И, как, следствие, уменьшилось среднее время исполнения.

Ключевые функции, отвечающие за планирование:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Функция | Входные параметры | Выходные параметры | Назначение | Связь функций друг с другом | |
| thread\_block() | - | - | Переводит поток в состояние «заблокирован» и вызывает планировщик | Вызывает функцию schedule() |
| thread\_unblock() | struct thread\* t | - | Переводит заблокированный поток в состояние «готов к исполнению» | - | |
| thread\_exit() | - | - | Переводит поток в состояние «завершается» и вызывает планировщик | Вызывает функцию schedule() | |
| thread\_yield() | - | - | Переводит поток в состояние «готов к исполнению» и вызывает планировщик | Вызывает функцию schedule() | |
| schedule() | - | - | Осуществляет выбор нового потока и переключение на него | Вызывает функцию switch\_threads(), вызывает thread\_schedule\_tail() | |
| next\_thread\_to\_run () | - | Struct thread\* | Определяет поток, на который произойдёт переключение | Используется в функции schedule() | |
| switch\_threads() | struct thread\* cur, struct thread\* next | struct thread\* | Осуществляет непосредственное переключение контекстов | Используется в schedule() для переключения контекста между процессами | |
| thread\_schedule\_tail() | struct thread\* prev | - | Переводит поток в состояние «исполняется», а также освобождает память прошлого потока, если он в состоянии «завершается» | - | |

*Таблица 3. Функции thread.c, отвечающие за планирование*

## Алгоритм приоритетного планирования

Чтобы при планировании процессов учитывался их приоритет в исходные коды ОС Pintos были внесены следующие изменения:

1. В *thread.c*:
   1. Добавлена функция thread\_priority\_comparator() для сравнения приоритетов процессов:

bool

thread\_priority\_comparator (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

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

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

return thread\_a->priority > thread\_b->priority;

}

* 1. Добавлена функция list\_reorder(), сортирующая список по убыванию приоритета:

void list\_reorder(struct list \*list\_)

{

if (list\_empty(list\_))

return;

struct list temp\_list;

list\_init(&temp\_list);

while (!list\_empty(list\_))

{

struct list\_elem\* highest\_priority\_elem = list\_min(list\_, thread\_priority\_comparator, NULL);

list\_remove(highest\_priority\_elem);

list\_push\_back(&temp\_list, highest\_priority\_elem);

}

while (!list\_empty(&temp\_list)) {

struct list\_elem \*elem = list\_pop\_front(&temp\_list);

list\_push\_back(list\_, elem);

}

}

* 1. В функцию thread\_create() добавлено присваивание приоритета потоку по умолчанию:

t->priority=priority; //my

* 1. В конце функции thread\_create() добавлен вызов функции thread\_yield(), чтобы обеспечить вытесняющий механизм планирования – если приоритет текущего процесса прекращает быть наивысшим, процессор освобождается от его выполнения.

tid\_t

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

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

…

thread\_unblock (t);

thread\_yield(); //my

return tid;

}

* 1. В функции thread\_unblock() после вставки процесса в список готовых к исполнению, список сортируется.

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\_reorder(&ready\_list); //my

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

* 1. В функции thread\_yield() после вставки процесса в список готовых к исполнению, список сортируется.

void

thread\_yield (void)

{

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

enum intr\_level old\_level;

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

if (cur != idle\_thread) {

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

list\_reorder(&ready\_list); //my

}

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

* 1. В функции thread\_set\_priority() добавлены отключения прерываний на время изменения приоритета потока. Если приоритет потока понижается, то вызывается функция thread\_yield(), и поток освобождает процессор. Таким образом также реализуется вытесняющее планирование.

void

thread\_set\_priority (int new\_priority)

{

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

thread\_current ()->priority = new\_priority; //no my

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

struct thread \*first = list\_entry(list\_front(&ready\_list), struct thread, elem);

if (first != NULL && first->priority > new\_priority) {

thread\_yield();

}

}

intr\_set\_level(old\_level);

}

1. В synch.c:
   1. В функции sema\_down() после добавления потока в список waiters, этот список сортируется по убыванию приоритета.

void

sema\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

while (sema->value == 0)

{

list\_push\_back (&sema->waiters, &thread\_current ()->elem);

list\_reorder(&sema->waiters); //my

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

* 1. В функции sema\_up() из отсортированного списка waiters для разблокировки выбирается самый высокоприоритетный поток. В случае, если его приоритет выше приоритета текущего потока, происходит вытеснение.

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

struct thread\* highest\_priority\_elem = NULL; //my

old\_level = intr\_disable ();

if (!list\_empty (&sema->waiters)) {

list\_reorder(&sema->waiters); //my

highest\_priority\_elem = list\_entry (list\_pop\_front (&sema->waiters),struct thread, elem); //my

thread\_unblock(highest\_priority\_elem);

}

sema->value++;

intr\_set\_level (old\_level);

if (highest\_priority\_elem!=NULL && highest\_priority\_elem->priority > thread\_current()->priority) //my

thread\_yield(); //my

}

* 1. В файле synch.h в структуру semaphore\_elem было добавлено поле priority, для работы с мониторами. Также созданы новые функции cond\_priority\_comparator() и cond\_list\_reorder(), отвечающие за сортировку списка waiters мониторов.

struct semaphore\_elem

{

struct list\_elem elem;

struct semaphore semaphore;

int priority; //my

}

bool

cond\_priority\_comparator (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

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

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

return thread\_a->priority > thread\_b->priority;

}

void cond\_list\_reorder(struct list \*list\_)

{

if (list\_empty(list\_))

return;

struct list temp\_list;

list\_init(&temp\_list);

while (!list\_empty(list\_))

{

struct list\_elem\* highest\_priority\_elem = list\_min(list\_, cond\_priority\_comparator, NULL);

list\_remove(highest\_priority\_elem);

list\_push\_back(&temp\_list, highest\_priority\_elem);

}

while (!list\_empty(&temp\_list)) {

struct list\_elem \*elem = list\_pop\_front(&temp\_list);

list\_push\_back(list\_, elem);

}

}

* 1. В функции cond\_wait() список waiters сортируется после вставки в него элемента

void

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

{

struct semaphore\_elem waiter;

waiter.priority=thread\_current()->priority; //my

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

lock->from\_cond = true;

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

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

cond\_list\_reorder(&cond->waiters); //my

lock\_release (lock);

sema\_down (&waiter.semaphore);

lock\_acquire (lock);

}

* 1. В функции cond\_signal() список waiters сортируется перед увеличение семафора

void

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

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

if (!list\_empty (&cond->waiters)) {

cond\_list\_reorder(&cond->waiters); //my

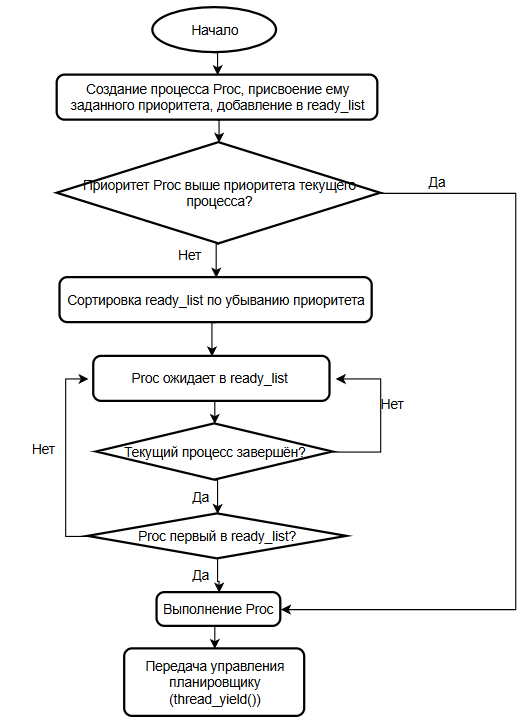
sema\_up (&list\_entry (list\_pop\_front (&cond->waiters),

struct semaphore\_elem, elem)->semaphore);

lock->from\_cond = false;

}

}



*Рисунок 2. Блок-схема реализованного алгоритма*

## Алгоритм разделения приоритета

Для реализации механизма разделения приоритетов при работе с критической секцией были внесены следующие изменения:

1. В thread.h в структуру thread добавлены новые поля:

struct thread

{

. . .

int priority; /\* Priority. \*/

int base\_priority; //для сохранения изначального приоритета

struct list held\_locks; //лист замков, захваченных потoком

struct lock\* waiting\_lock; //замок, который ожидает поток

struct thread \*donor; //поток, приоритет которого был заимствован

struct list held\_locks\_start; //лист замков, захваченных главным потоком

. . .

};

1. В synch.h в структуру lock добавлены новые поля:

struct lock

{

struct thread \*holder; /\* Thread holding lock (for debugging). \*/

struct semaphore semaphore; /\* Binary semaphore controlling access. \*/

struct thread \*donor\_thread; //поток, удерживающий замок

bool is\_priority\_donated;//флаг - было ли донорство

struct list\_elem elem;

struct list\_elem elem1;

bool from\_cond;

};

1. В synch.c:
   1. Добавлена новая рекурсивная функция donate\_priority(), в которой осуществляется непосредственно передача приоритета; и также отвечающая за случай, когда несколько потоков ожидают входа в критическую секцию. В этом случае приоритет передаётся по цепочке.

void donate\_priority(struct thread \*donor, struct thread \*donee) {

if (donor->priority > donee->priority) {

donee->priority = donor->priority;

donee->donor = donor;

if (donee->waiting\_lock != NULL) {

donate\_priority(donor, donee->waiting\_lock->holder);

}

}

}

* 1. Добавлена новая функция restore\_priority(), восстанавливающая исходный приоритет потока при освобождении критической секции.

void restore\_priority(struct thread \*t) {

t->priority = t->base\_priority;

if (!list\_empty(&t->held\_locks)) {

struct list\_elem \*e;

for (e = list\_begin(&t->held\_locks); e != list\_end(&t->held\_locks); e = list\_next(e)) {

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

if (!list\_empty(&lock->semaphore.waiters)) {

struct thread \*max\_waiter = list\_entry(list\_front(&lock->semaphore.waiters), struct thread, elem);

if (max\_waiter->priority > t->priority) {

t->priority = max\_waiter->priority;

}

}

}

}

if (!list\_empty(&t->held\_locks\_start)) {

struct list\_elem \*e = list\_begin(&t->held\_locks\_start);

struct lock \*lock = list\_entry(e, struct lock, elem1);

if (!list\_empty(&lock->semaphore.waiters)) {

struct thread \*max\_waiter = list\_entry(list\_front(&lock->semaphore.waiters), struct thread, elem);

if (max\_waiter->priority > t->priority) {

t->priority = max\_waiter->priority;

}

}

}

}

* 1. В функцию lock\_init() добавлено присваивание начальных значений новых полей.

void

lock\_init (struct lock \*lock)

{

ASSERT (lock != NULL);

lock->holder = NULL;

sema\_init (&lock->semaphore, 1);

lock->is\_priority\_donated = false;

lock->from\_cond = false;

}

3.4. Изменена функция lock\_acquire(). При захвате замка, поток добавляется в его список held\_locks и вызывается функция donate\_priority().

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

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

if (lock->from\_cond == false) {

if (lock->holder != NULL && lock->holder->priority < current -> priority) {

current->waiting\_lock=lock;

donate\_priority(current, lock->holder);

lock->is\_priority\_donated = true;

lock->donor\_thread = current;

lock->holder->donor=current;

list\_push\_back(&current->held\_locks, &lock -> elem);

}

else

list\_push\_back(&current->held\_locks\_start, &lock -> elem1);

}

sema\_down (&lock->semaphore);

current->waiting\_lock = NULL;

lock->holder = current;

}

3.5. Изменена функция lock\_release(). Теперь при освобождении замка он удаляется из списка held\_locks, после чего происходит восстановление приоритета в функции restore-priority.

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

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

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

if (lock->from\_cond == false) {

list\_remove(&lock->elem1);

if (lock -> is\_priority\_donated == true) {

if (!(list\_empty(&current->held\_locks)))

list\_remove(&lock->elem);

restore\_priority(current);

if (list\_empty(&current->held\_locks\_start))

lock->is\_priority\_donated = false;

lock->donor\_thread = NULL;

lock->holder->donor = NULL;

}

}

lock->holder = NULL;

sema\_up (&lock->semaphore);

}

1. В thread.c. были внесены изменения в функцию thread\_set\_priority() так, чтобы пока поток заимствует приоритет, он не мог его понизить (изменяется только base\_priority).

void

thread\_set\_priority (int new\_priority)

{

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

thread\_current ()->base\_priority = new\_priority; //no my

if (thread\_current()->donor == NULL || new\_priority > thread\_current()->priority)

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

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

struct thread \*first = list\_entry(list\_front(&ready\_list), struct thread, elem);

if (first != NULL && first->priority > new\_priority) {

thread\_yield();

}

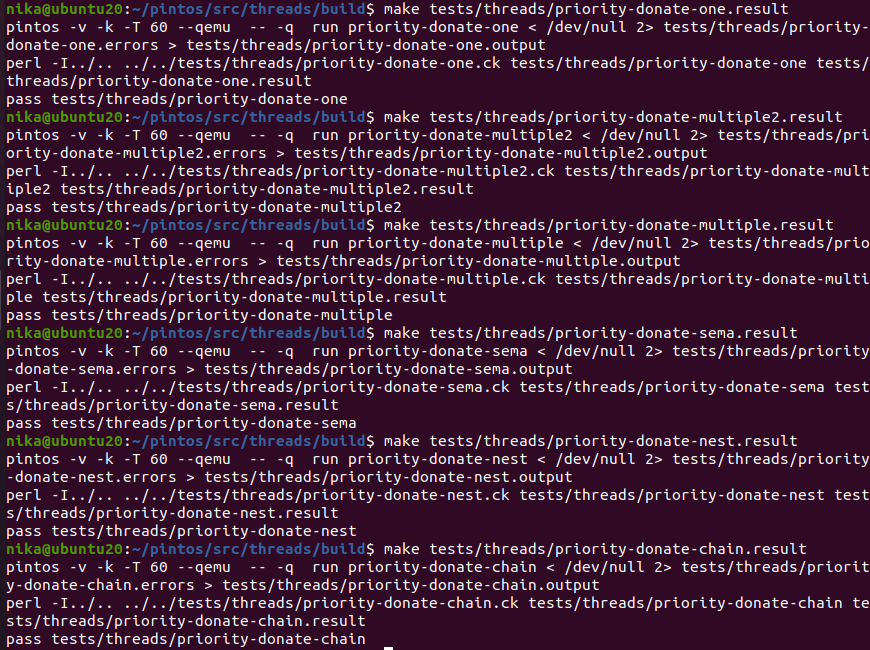
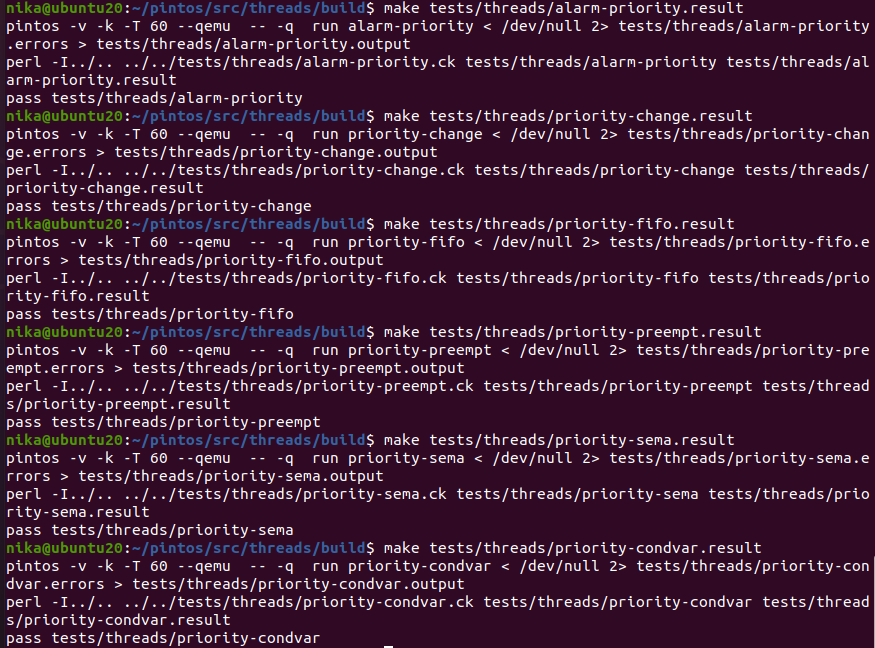
}

intr\_set\_level(old\_level);

}

## Результаты тестов

После внесённых изменений все представленные в задании тесты проходят корректно:

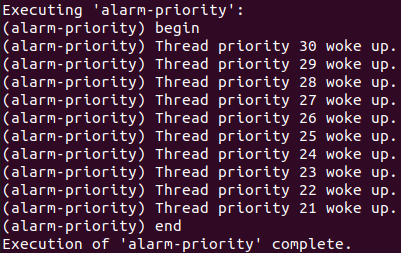


*Рисунок 3. Результаты тестов*

Рассмотрим каждый тест более подробно:

1. аlarm-priority

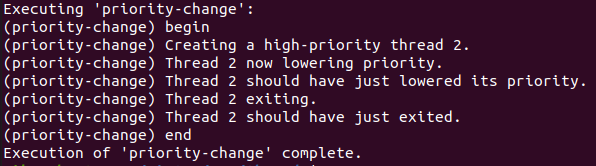
Проверяет, что, когда таймер пробуждает потоки, потоки с более высоким приоритетом запускаются первыми.



*Рисунок 4. Вывод теста alarm-priority*

1. priority-change

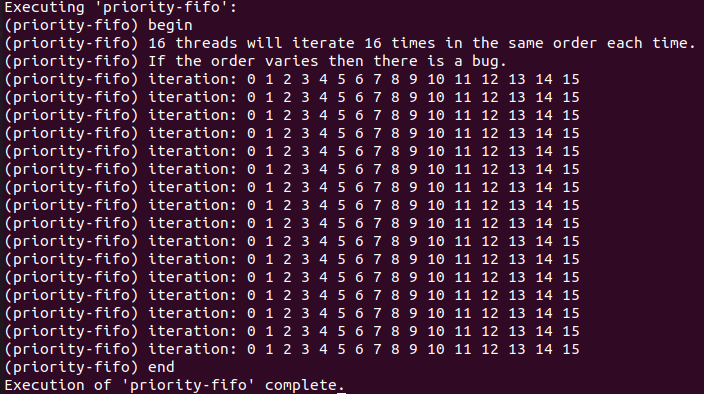
Проверяет, что снижение приоритета потока таким образом, что он больше не является потоком с наивысшим приоритетом в системе, приводит к его немедленному завершению.



*Рисунок 5. Вывод теста priority-change*

1. priority-fifo

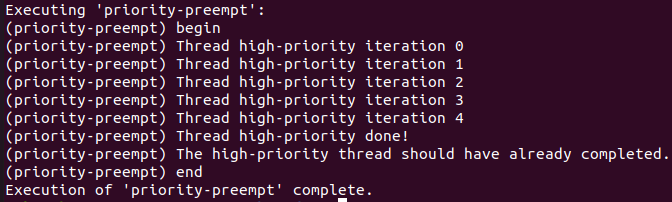
Создает несколько потоков с одинаковым приоритетом и гарантирует, что они последовательно выполняются в одном и том же циклическом порядке (по алгоритму Round Robin).



*Рисунок 6. Вывод теста priority-fifo*

1. priority-preempt

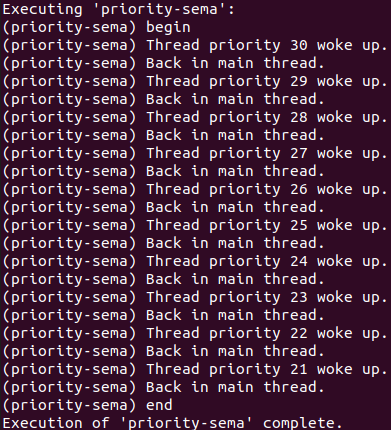
Гарантирует, что высокоприоритетный поток действительно будет запущен с вытеснением.



*Рисунок 7. Вывод теста priority-preempt*

1. priority-sema

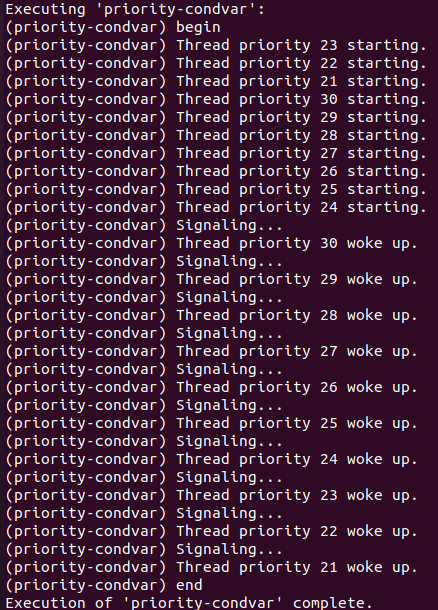
Проверяет, что поток с наивысшим приоритетом, ожидающий семафора, просыпается первым.



*Рисунок 8. Вывод теста priority-sema*

1. priority-condvar

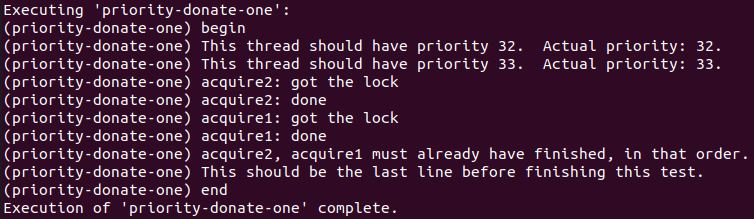
Проверяет, что функция cond\_signal() запускает поток с наивысшим приоритетом, ожидающий в cond\_wait().



*Рисунок 9. Вывод теста priority-condvar*

1. priority-donate-one

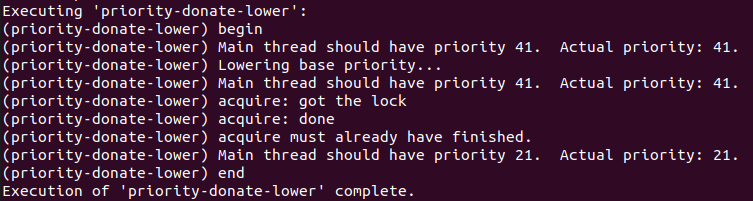
Основной поток захватывает замок. Затем он создает два потока с более высоким приоритетом, которые блокируются при захвате замка, в результате чего они передают свои приоритеты основному потоку. Когда основной поток освобождает замок, другие потоки должны захватывать его в порядке приоритета.



*Рисунок 10. Вывод теста priority-donate-one*

1. priority-donate-lower

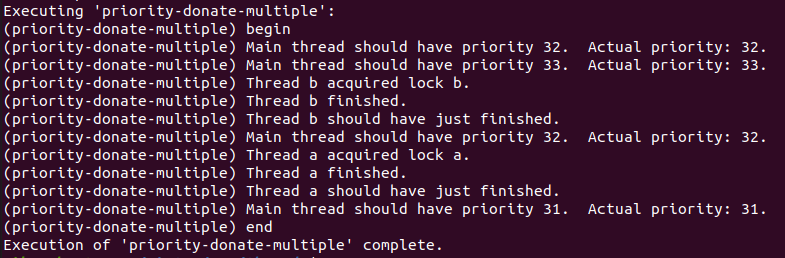
Основной поток захватывает замок. Затем он создает поток с более высоким приоритетом, который блокируется при захвате замка, в результате чего он передает свой приоритет основному потоку. Основной поток пытается понизить свой приоритет, что не должно происходить до тех пор, пока происходит пожертвование.



*Рисунок 11. Вывод теста priority-donate-lower*

1. priority-donate-multiple

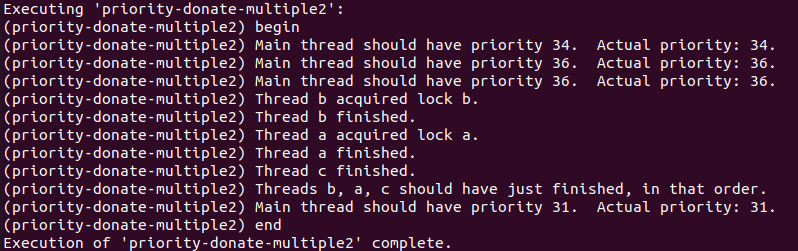
Главный поток захватывает замки A и B, затем он создает два потока с более высоким приоритетом. Каждый из этих потоков блокируется при захвате одного из замков и, таким образом, передает свой приоритет главному потоку. Главный поток, в свою очередь освобождает замки, и отказывается от своих переданных приоритетов.



*Рисунок 12. Вывод теста priority-donate-multiple*

1. priority-donate-multiple2

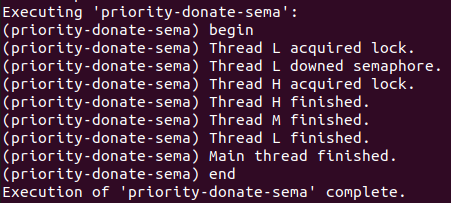
Основной поток захватывает замки A и B, затем он создает три потока с более высоким приоритетом. Первые два из этих потоков блокируются при захвате одного из замков и, таким образом, передают свой приоритет основному потоку. Основной поток, в свою очередь, освобождает замки и отказывается от своих переданных приоритетов, позволяя третьему потоку работать.



*Рисунок 13. Вывод теста priority-donate-multiple2*

1. priority-donate-sema

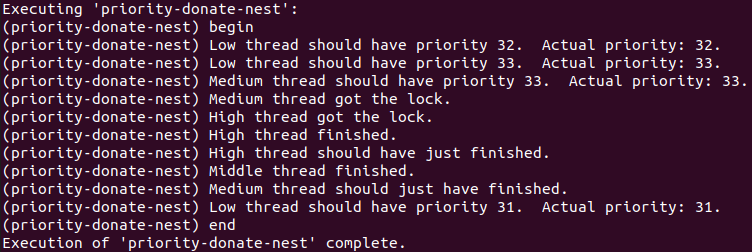
Поток L с низким приоритетом захватывает замок, затем блокирует уменьшение семафора. Поток M со средним приоритетом блокирует ожидание на том же семафоре. Затем поток H с высоким приоритетом пытается захватить замок, передавая свой приоритет L. Затем основной поток увеличивает семафор, пробуждая L. L освобождает замок, пробуждая H. H увеличивает семафор, пробуждая M. H завершает работу, затем M, затем L и, наконец, основной поток.



*Рисунок 14. Вывод теста priority-donate-sema*

1. priority-donate-nest

Основной поток L с низким приоритетом захватывает замок A. Поток M со средним приоритетом захватывает замок B и блокируется при захвате замка A. Поток H с высоким приоритетом затем блокируется при захвате замка B. Таким образом, поток H передает свой приоритет потоку M, который, в свою очередь, передает его потоку L.



*Рисунок 15. Вывод теста priority-donate-nest*

1. priority-donate-chain

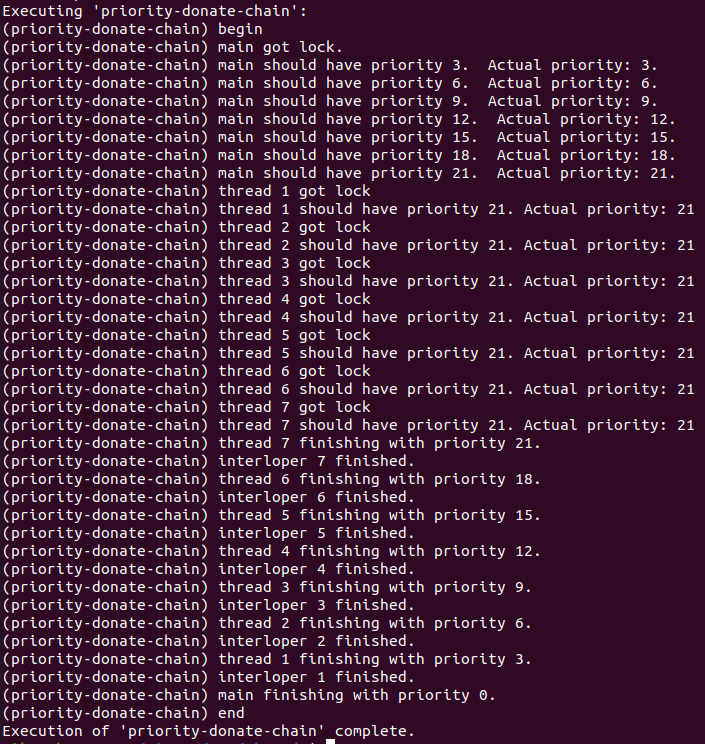
Основной поток установил свой приоритет в PRI\_MIN и создает 7 потоков (поток 1..7) с приоритетами PRI\_MIN + 3, 6, 9, 12, ...

Основной поток инициализирует 8 замков: замки 0..7 и захватывает замок 0.

Когда поток [i] запускается, он сначала захватывает замок[i]. Впоследствии поток[i] пытается захватить замок[i-1], который уже захвачен потоком[i-1], за исключением замка [0], который удерживается основным потоком. Поскольку замок удерживается, поток [i] передает свой приоритет потоку [i-1], который передает приоритет потоку [i-2], и так далее, пока основной поток не получит пожертвование.

После того, как потоки [1..7] были созданы и заблокированы на захвате замков[0..7], основной поток освобождает замок[0], разблокируя поток[1] и вытесняя его. Поток[1] затем завершает захват замка[0], затем освобождает замок[0], затем освобождает замок[1], разблокирует поток[2] и т.д… Поток [7], наконец, захватывает и освобождает замок[7] и завершает работу, позволяя потоку [6], затем потоку [5] и т.д. запускаться и завершаться, пока, наконец, основной поток не завершит работу.

Кроме того, создаются потоки-нарушители с приоритетами p = PRI\_MIN + 2, 5, 8, 11, ... которые не должны запускаться до тех пор, пока не завершится соответствующий поток с приоритетом p + 1.



*Рисунок 16. Вывод теста priority-donate-chain*

## Тест test-new-alg

Алгоритм Round Robin уже реализован в ОС Pintos в thread.c в функции thread\_tick() c TIME\_SLICE = 4. Однако, необходимо внедрить корректное завершение потока по истечении его cpu\_burst. Для этого в структуру thread были добавлены следующие новые поля:

struct thread

{

...

uint64\_t cpu\_burst; // непосредственно cpu\_burst

uint64\_t cpu\_burst\_max; // копия cpu\_burst для корректного вывода

bool cpu\_print; // нужно ли печатать сообщение о количестве тиков

bool needs\_exit; // для корректного завершения потока

...

};

Была создана функция thread\_create\_cpu() полностью аналогичная функции thread\_create(), но принимающая на вход также cpu\_burst потока, а также присваивающая needs\_exit значение false.

tid\_t

thread\_create\_cpu (const char \*name, int priority, uint64\_t cpu\_burst,

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

{

...

t->cpu\_burst = cpu\_burst;

t->cpu\_burst\_max = cpu\_burst;

t->cpu\_print = false;

t->needs\_exit = false;

...

}

В функцию thread\_tick() было добавлено условие для завершения потока по окончании его cpu\_burst. Поток удаляется из общего списка потоков, изменяет поле needs\_exit и вызывает thread\_yield() через intr\_yield\_on\_return(). Также в этой функции осуществляется вывод сообщения о количестве тиков, отработанных потоком.

void

thread\_tick (void)

{

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++;

//my beg

if (t!=idle\_thread && t->cpu\_print) {

t->cpu\_burst--;

msg("%s has worked %d ticks", t->name, t->cpu\_burst\_max - t->cpu\_burst);

if (t->cpu\_burst == 0) {

list\_remove (&t->allelem);

t->needs\_exit = true;

intr\_yield\_on\_return();

msg("%s has finished", t->name);

}

}

//my end

/\* Enforce preemption. \*/

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

intr\_yield\_on\_return ();

}

В функцию thread\_yield() была добавлена проверка поля needs\_exit. Если флаг needs\_exit поднят (true), то поток не добавляется в ready\_list, а наоборот изменяет свой статус на “завершается”.

void

thread\_yield (void)

{

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

enum intr\_level old\_level;

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

if (cur != idle\_thread && cur->needs\_exit==false) {

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

list\_reorder(&ready\_list); //my

}

//my beg

if (cur -> needs\_exit) {

cur->status = THREAD\_DYING;

schedule ();

}

//my end

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

В самом же тесте в функции test\_cpu, исполняемой каждым из созданных потоков, только поднимается флаг cpu\_print, чтобы выводилось сообщение о тиках, отработанных потоком. Код теста test-new-alg:

#include <stdio.h>

#include "tests/threads/tests.h"

#include "threads/malloc.h"

#include "threads/thread.h"

#include "threads/synch.h"

#include "devices/timer.h"

void test\_cpu (void \* arg UNUSED);

void test\_new\_alg(void)

{

thread\_set\_priority (PRI\_MAX);

msg ("Main thread should have the highest priority %d", thread\_get\_priority ());

int i;

int priority[] = {56, 10, 58, 3};

uint64\_t cpu\_burst[] = {11, 2,3,5};

char name[16];

for (i=0; i<4; i++) {

snprintf (name, sizeof name, "Proc%d", i);

thread\_create\_cpu (name, priority[i], cpu\_burst[i], test\_cpu, NULL);

msg ("Proc%d was created with priority %d, cpu %d", i, priority[i], cpu\_burst[i]);

}

thread\_set\_priority (PRI\_MIN);

msg ("Main thread should have the lowest priority %d", thread\_get\_priority ());

msg ("Main thread should have been finished");

}

void test\_cpu(void \*arg UNUSED) {

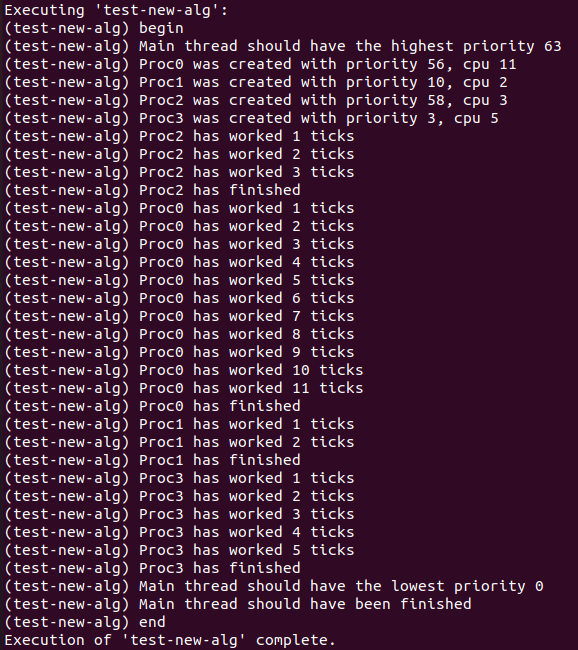
struct thread \*cur = thread\_current();

cur->cpu\_print = true;

while (true) {

}

}



*Рисунок 17. Вывод теста test-new-alg*

# Вывод

В ходе лабораторной работы были изучены алгоритмы планирования, их реализация в ОС Pintos с помощью каталогов thread.c и synch.c. В существующую систему планирования успешно внедрен учет приоритета потоков. Также внедрен механизм разделения приоритета, который решает проблему инверсии приоритета при доступе к критической секции. На примере нового созданного теста была рассмотрена работа алгоритма Round Robin с заданным набором потоков.