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

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. «Планирование процессов»
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**

Изучение механизмов планирования процессов, разработка алгоритма приоритетного планирования и внедрение разработанного алгоритма в учебную операционную систему Pintos.

1. **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 возможноть не производить значительный объем вычислений при переключении процессов.

1. **Теоретические исследования**

Алгоритм планирования в соответствии с выданным вариантом – Round Robin. Очередность процессов, находящихся в состоянии «готовность» организована циклическим образом. Каждый процесс находится возле процессора фиксированный квант времени, на время которого он получает процессор в свое распоряжение (обычно 10 - 100 миллисекунд). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться. Очередь организована методом FIFO(First In First Out). При больших величинах кванта времени алгоритм вырождается в FCFS.

Таблица — Процессы на исполнение

|  |  |  |
| --- | --- | --- |
| Процесс: | CPU burst: | Приоритет процесса: |
| Proc0 | 8 | 3 |
| Proc1 | 4 | 28 |
| Proc2 | 7 | 62 |
| Proc3 | 3 | 31 |

Таблица — Диаграмма исполнения процессов для исходного алгоритма планирования Pintos

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 0 | и | и | и | и | и | и | и | и | в | в | в | в | в | в | в | в | в | в | в | в | в | в |
| 1 | г | г | г | г | г | г | г | г | и | и | и | и | в | в | в | в | в | в | в | в | в | в |
| 2 | г | г | г | г | г | г | г | г | г | г | г | г | и | и | и | и | и | и | и | в | в | в |
| 3 | г | г | г | г | г | г | г | г | г | г | г | г | г | г | г | г | г | г | г | и | и | и |

Среднее время ожидания:

Среднее время исполнения:

Таблица — Диаграмма исполнения процессов для модифицированного алгоритма

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 0 | г | г | г | г | г | г | г | г | г | г | г | г | г | г | и | и | и | и | и | и | и | и |
| 1 | г | г | г | г | г | г | г | г | г | г | и | и | и | и | в | в | в | в | в | в | в | в |
| 2 | и | и | и | и | и | и | и | в | в | в | в | в | в | в | в | в | в | в | в | в | в | в |
| 3 | г | г | г | г | г | г | г | и | и | и | в | в | в | в | в | в | в | в | в | в | в | в |

Среднее время ожидания:

Среднее время исполнения:

Вывод: при модификации алгоритма планирования уменьшается среднее время ожидания, среднее время исполнения не меняется.

Таблица — Основные функции, выполняющие планирование

|  |  |
| --- | --- |
| Функция | Описание функции |
| static void  schedule (void) | В этой функции выполняется планирование процессов. Для начала вызываются две вспомогательные функции, которые определяют текущий процесс и процесс, которые будет выполняться следующим. Далее вызывает switch\_threads() для переключения потоков. Остальная часть планировщика реализована в функции thread\_schedule\_tail(), которая помечает новый процесс как выполняющийся. |
| static void  idle (void \*idle\_started\_ UNUSED) | Если в данный момент системе нет готовых к выполнению процессов, то планировщик выбирает специальный "пустой" процесс. |
| struct thread \*switch\_threads (struct thread \*cur, struct thread \*next); | Помещает состояние регистров и указателя вершины стека в соответствующее поле структуры thread текущего процесса и восстанавливает эти данные у следующего процесса. Используется в функции schedule() |
| void  thread\_block (void) | Переводит поток в состояние «Заблокирован» и вызывает планировщик |
| void  thread\_exit (void) | Завершает процесс и вызывает планировщик |
| void  thread\_yield (void) | Переводит поток в состояние «Готов к исполнению» и вызывает планировщик |
| void  thread\_schedule\_tail (struct thread \*prev) | Помечает поток в состояние «Исполняется». Используется в функции schedule() |
| struct thread \*  running\_thread (void) | Возвращает запущенный процесс. Используется в функции schedule() |
| static struct thread \*  next\_thread\_to\_run (void) | Возвращает следующий запланированный поток. Возвращает поток из очереди выполнения, если только очередь выполнения не пуста. Используется в функции schedule() |

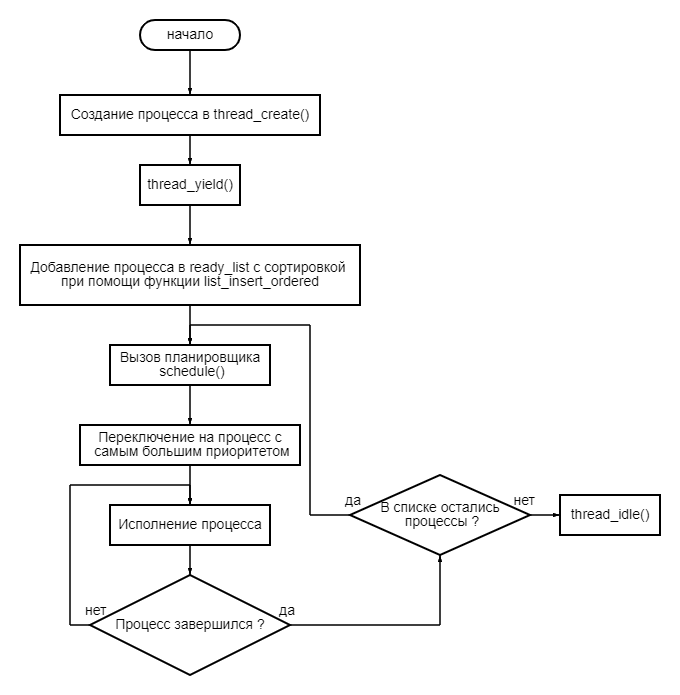


Рисунок — Блок схема разработанного алгоритма

1. **Описание решения**

При создании планирования процессов было необходимо использовать список, чтобы сортировать в нём процессы по убыванию приоритетов, так как так проще всего определять процесс, который необходимо запускать следующим.

Для этого был изучен файл list.c, в котором реализован двунаправленный список, одну из функций этого файла можно использовать для вставки с сортировкой по какому-либо признаку, это

list\_insert\_ordered (struct list \*list, struct list\_elem \*elem, list\_less\_func \*less, void \*aux)

Данная функция ищет место, куда вставить значение, после чего вызывает list\_insert(e, elem). Также данной функции необходим критерий, по которому будет осуществляться вставка в список, в моём случае, мне необходимо знать, больше ли у следующего элемента приоритет , чем у текущего, для этого была написана булиевая функция thread\_cmp\_priority()

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

return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority;}

Для следующего теста – priority\_change, необходимо сделать так, чтобы при изменении приоритета или создании процесса, выполняющийся поток вставал в списке в соответствии со своим новым приоритетом, а также, возможно, прекращал работу. Для этого я вызываю thread\_yield() при изменении приоритета, а также при создании нового потока в случае, если его приоритет выше выполняющегося, после этого вызовется и thread\_current(), соответственно этот поток встанет перед потоками с меньшим приоритетом.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Для теста priority-sema, необходимо сортировать потоки по убыванию приоритета для того, чтобы при их разблокировке сначала шли самые приоритетные.

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

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

list\_sort(&sema->waiters, &thread\_compare\_priority, NULL);

thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),

struct thread, elem));

sema->value++;

thread\_yield();

intr\_set\_level (old\_level);

}

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\_insert\_ordered(&sema->waiters, &thread\_current ()->elem, thread\_compare\_priority, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

При тесте priority-condvar мы уже не можем отсортировать потоки стоящие на семафоре, так как у них не предусмотрен параметр приоритета, поэтому нам необходимо его добавить в структуру semaphore\_elem.

struct semaphore\_elem

{

struct list\_elem elem; /\* List element. \*/

struct semaphore semaphore; /\* This semaphore. \*/

int priority;

};

После этого необходимо написать новое условие сортировки, которое будет работать со структурами элементов семафора.

bool sema\_cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

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

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

return (first->semaphore.priority > second->semaphore.priority);

}

И наконец после всех этих операций мы можем добавить сортировку потоков, стоящих на семафоре в функции cond\_wait:

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

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

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

list\_sort(&cond->waiters, &sema\_cmp\_priority, NULL);

lock\_release (lock);

Далее необходимо реализовать пожертвование приоритета потоков.

Если поток блокирует другие потоки, приоритет которых выше, чем у него самого и при этом он занимает ресурсы процессора, то система простаивает, ведь она захочет поставит поток с наивысшим приоритетом на выполнение, в то время как он заблокирован другим потоком, из-за чего операционная система не сможет продолжать работу.

Необходимо сделать так, чтобы заблокированный поток с наивысшим приоритетом делился своим приоритетом с держателем замка для того, чтобы второй закончил работу и освободил поток с большим приоритетом.

Для этого необходимо отслеживать ситуацию, при которой держатель замка занимает ресурсы, после чего, среди всех потоков заблокированных обладателем замка необходимо осуществить поиск потока с наибольшим приоритетом, после этого реализовать обновление приоритета и возврат приоритета потоку, который его жертвовал.

Для этого необходимо переписать функции lock\_acquire и lock\_release, которые отвечают за наложение и снятие замка, а также добавить в структуру потока значение приоритета до пожертвования, список потоков – обладателей замков и заблокированных потоков.

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

if (!sema\_try\_down(&lock->semaphore)){

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

temp\_thread->blocked\_by = lock;

while (temp\_thread->blocked\_by != NULL){

if (temp\_thread->priority\_before\_donate > temp\_thread->blocked\_by->holder->priority\_before\_donate){//приоритет текущего больше приоритета процесса обладателя замка

temp\_thread->blocked\_by->holder->priority\_before\_donate = temp\_thread->priority\_before\_donate;//приоритет обладателя замка будет равным приоритету текущего процесса

temp\_thread = temp\_thread->blocked\_by->holder;//рассматриваем обладателя замка

}

else

break;

}

sema\_down(&lock->semaphore);

}

enum intr\_level last = intr\_disable();//отключаем прерывания, запоминает предыдущее состояние прерываний

lock->holder = thread\_current();//текущий поток - держатель замка

list\_push\_back(&thread\_current()->blocked, &lock->locked);

thread\_current()->blocked\_by = NULL;

intr\_set\_level(last);

//sema\_down (&lock->semaphore);

//lock->holder = thread\_current ();

}

В функции lock\_acquire() происходит поиск заблокированного потока, после чего, если его приоритет больше приоритета обладателя замка, то этот приоритет присваивается обладателю замка.

В функции lock\_release(), если поток блокирует другие потоки, то осуществляется проход по всему списку блокируемых потоков, для каждого из которых происходит проход по списку потоков, ожидающих на семафоре.

Во время прохода осуществляется поиск максимального приоритета среди всех этих потоков. Затем этот приоритет присваивается текущему потоку и замок освобождается.

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

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

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

list\_remove(&lock->locked);

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

if (!list\_empty(&cur\_thread->blocked)){

struct thread \*wait\_th;

struct list\_elem \*blocked, \*ready;

struct lock \*lock\_curr;

int max = cur\_thread->priority;

for (blocked = list\_begin(&cur\_thread->blocked); blocked != list\_end(&cur\_thread->blocked); blocked = list\_next(blocked)){

lock\_curr = list\_entry(blocked, struct lock, locked);

for (ready = list\_begin(&lock\_curr->semaphore.waiters); ready != list\_end(&lock\_curr->semaphore.waiters); ready = list\_next(ready)){

wait\_th = list\_entry(ready, struct thread, elem);

if (wait\_th->priority\_before\_donate > max)

max = wait\_th->priority\_before\_donate;

}

}

cur\_thread->priority\_before\_donate = max;

}

else

cur\_thread->priority\_before\_donate = cur\_thread->priority;

lock->holder = NULL;

intr\_set\_level(last);

sema\_up(&lock->semaphore);

}

Также из-за добавления новых значений в структуру потока необходимо дополнить функцию инициализации потока.

t->priority\_before\_donate = priority;

t->blocked\_by = NULL;

list\_init(&t->blocked);

И исправить функцию переопределения приоритета потока.

void

thread\_set\_priority (int new\_priority)

{

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

if (list\_empty(&thread\_current()->blocked)){

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

thread\_yield();

}

if (new\_priority > thread\_current()->priority\_before\_donate){

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

return;

}

А также исправить функцию thread\_get\_priority() и добавить сортировку в функцию переключения потоков.

int

thread\_get\_priority (void)

{

return thread\_current ()->priority\_before\_donate;

}

1. **Тестирование и результаты работы программы**

Тест alarm-priority проверяет, что при пробуждении процессов, пробуждается процесс с наивысшим приоритетом.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-change проверяет, что если текущий процесс получает приоритет, который меньше приоритета процессов, готовых к использованию, то текущий процесс блокируется и начинает исполняться процесс с более высоким процессом.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-preempt проверяет, что не происходит переключение на другой процесс, если нет процесса с приоритетом выше текущего.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-fifo создает процессы с одинаковым приоритетом и смотрит, что данные процессы выполняются по очереди.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-sema проверяет, что первым просыпается процесс, который имеет самый высокий приоритет из ждущих семафора.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-condvar проверяет, что cond\_signal() пробуждает самый высокоприоритетный процесс, ждущий в cond\_wait().

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-one, поток main получает замок. Затем он создает два потока, имеющие больший приоритет. Эти процессы пытаются получить замок, но так он занят потоком main, другие процессы должны передать ему свой приоритет. Когда основной поток освобождает блокировку, другие процессы обязаны получить ее в порядке приоритета.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-lower, процесс main присваивает замок и создает процесс приоритетом выше, чем имеет сам. Новый процесс, пытаясь получить замок, передает свой приоритет main. После этого main понижает свой приоритет, что не должно отразиться на переданном приоритете, т.е. приоритет main должен возвращаться к исходному только после возврата пожертвования

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-multiple, main получает замки A и B, а затем создает два потока с более высокими приоритетами. Каждый из этих потоков пытается присвоить себе один из этих замков, для чего передает свой приоритет. Основной поток, получая приоритеты этих двух потоков, освобождает блокировки и отказывается от своих приоритетов.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-multiple2, главный поток получает замки A и B, потом создает 3 процесса с более высоким приоритетом. Два из потоков пытаются получить один из замков и передают свой приоритет основному процессу. Основной поток поочередно освобождает блокировки и возвращает исходные приоритеты, т.е. дает третьему потоку возможность работать.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-sema, процесс L с небольшим приоритетом получает замок, затем блокируется при уменьшении значения семафора. Процесс M с приоритетом выше, чем у L, блокируется на этом же семафоре. Процесс H с более высоким приоритетом, пытается присвоить замок и передает приоритет L. Далее основной поток увеличивает значения семафора, пробуждая L. L освобождает замок, что должно разбудить H. H снова увеличивает значение семафора, и просыпается M. В итоге процессы завершаются в порядке: H, M, L, основной поток.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-nest, основной поток L с низким приоритетом получает замок A. Поток M со средним приоритетом - замок B, затем пытается присвоит замок А и передает приоритет потоку L. Поток H c наибольшим приоритетом пытается получить замок B. В результате поток H передает свой приоритет потоку M, который жертвует этот приоритет потоку L.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Тест priority-donate-chain, основной поток с минимальным приоритетом создает 7 потоков, приоритет каждого из которых больше предыдущих. Затем создаются 8 замков. Первый присваивается основному потоку, а далее замки присваиваются в соответствии с порядком каждому потоку. Каждый из этих потоков пытается получить замок предыдущего, что приводит к цепочке. В итоге из-за цепочки блокировок происходит цепочка деления приоритета, то есть основной поток получает приоритет высшего потока и освобождает замок, передавая приоритет, завершает работу уже с исходным приоритетом и так до конца цепочки.

Изображение выглядит как текст, снимок экрана, Шрифт, информация

Автоматически созданное описание

1. **Выводы**

В результате этой лабораторной работы были изучены исходные коды ОС pintos, отвечающие за синхронизацию и планирование, а также сами алгоритмы планирования. Был получен алгоритм планирования и деления приоритетов. Полученный алгоритм более приемлем, потому что учитывает приоритет потоков.

Список используемых источников

1. <https://www.cs.usfca.edu/~benson/cs326/pintos/doc/pintos.pdf>
2. <https://web.stanford.edu/class/cs140/projects/pintos/pintos_6.html#SEC113>
3. <https://uchicago-cs.github.io/mpcs52030/switch.html>
4. <https://jhu-cs318.github.io/pintos-doxygen/html/timer_8c.html#ad9c100ae268edb874adef6889e7b5f25>

ПРИЛОЖЕНИЕ А

Листинг программы «thread.c»

#include "threads/thread.h"

#include <debug.h>

#include <stddef.h>

#include <random.h>

#include <stdio.h>

#include <string.h>

#include "threads/flags.h"

#include "threads/interrupt.h"

#include "threads/intr-stubs.h"

#include "threads/palloc.h"

#include "threads/switch.h"

#include "threads/synch.h"

#include "threads/vaddr.h"

#ifdef USERPROG

#include "userprog/process.h"

#endif

/\* Random value for struct thread's `magic' member.

Used to detect stack overflow. See the big comment at the top

of thread.h for details. \*/

#define THREAD\_MAGIC 0xcd6abf4b

/\* List of processes in THREAD\_READY state, that is, processes

that are ready to run but not actually running. \*/

static struct list ready\_list;

/\* List of all processes. Processes are added to this list

when they are first scheduled and removed when they exit. \*/

static struct list all\_list;

/\* Idle thread. \*/

static struct thread \*idle\_thread;

/\* Initial thread, the thread running init.c:main(). \*/

static struct thread \*initial\_thread;

/\* Lock used by allocate\_tid(). \*/

static struct lock tid\_lock;

/\* Donor thread(giving priority to other threads)\*/

//static struct thread \*donor;

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

struct kernel\_thread\_frame

{

void \*eip; /\* Return address. \*/

thread\_func \*function; /\* Function to call. \*/

void \*aux; /\* Auxiliary data for function. \*/

};

/\* Statistics. \*/

static long long idle\_ticks; /\* # of timer ticks spent idle. \*/

static long long kernel\_ticks; /\* # of timer ticks in kernel threads. \*/

static long long user\_ticks; /\* # of timer ticks in user programs. \*/

/\* Scheduling. \*/

#define TIME\_SLICE 4 /\* # of timer ticks to give each thread. \*/

static unsigned thread\_ticks; /\* # of timer ticks since last yield. \*/

/\* If false (default), use round-robin scheduler.

If true, use multi-level feedback queue scheduler.

Controlled by kernel command-line option "-o mlfqs". \*/

bool thread\_mlfqs;

static void kernel\_thread (thread\_func \*, void \*aux);

static void idle (void \*aux UNUSED);

static struct thread \*running\_thread (void);

static struct thread \*next\_thread\_to\_run (void);

static void init\_thread (struct thread \*, const char \*name, int priority);

static bool is\_thread (struct thread \*) UNUSED;

static void \*alloc\_frame (struct thread \*, size\_t size);

static void schedule (void);

void thread\_schedule\_tail (struct thread \*prev);

static tid\_t allocate\_tid (void);

bool thread\_cmp\_priority(const struct list\_elem\*, const struct list\_elem\*, void\*);

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

return list\_entry(a, struct thread, elem)->priority\_before\_donate > list\_entry(b, struct thread, elem)->priority\_before\_donate;

}

/\* Initializes the threading system by transforming the code

that's currently running into a thread. This can't work in

general and it is possible in this case only because loader.S

was careful to put the bottom of the stack at a page boundary.

Also initializes the run queue and the tid lock.

After calling this function, be sure to initialize the page

allocator before trying to create any threads with

thread\_create().

It is not safe to call thread\_current() until this function

finishes. \*/

void

thread\_init (void)

{

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

lock\_init (&tid\_lock);

list\_init (&ready\_list);

list\_init (&all\_list);

/\* Set up a thread structure for the running thread. \*/

initial\_thread = running\_thread ();

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

initial\_thread->status = THREAD\_RUNNING;

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

}

/\* Starts preemptive thread scheduling by enabling interrupts.

Also creates the idle thread. \*/

void

thread\_start (void)

{

/\* Create the idle thread. \*/

struct semaphore idle\_started;

sema\_init (&idle\_started, 0);

thread\_create ("idle", PRI\_MIN, idle, &idle\_started);

/\* Start preemptive thread scheduling. \*/

intr\_enable ();

/\* Wait for the idle thread to initialize idle\_thread. \*/

sema\_down (&idle\_started);

}

/\* Called by the timer interrupt handler at each timer tick.

Thus, this function runs in an external interrupt context. \*/

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++;

/\* Enforce preemption. \*/

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

intr\_yield\_on\_return ();

}

/\* Prints thread statistics. \*/

void

thread\_print\_stats (void)

{

printf ("Thread: %lld idle ticks, %lld kernel ticks, %lld user ticks\n",

idle\_ticks, kernel\_ticks, user\_ticks);

}

/\* Creates a new kernel thread named NAME with the given initial

PRIORITY, which executes FUNCTION passing AUX as the argument,

and adds it to the ready queue. Returns the thread identifier

for the new thread, or TID\_ERROR if creation fails.

If thread\_start() has been called, then the new thread may be

scheduled before thread\_create() returns. It could even exit

before thread\_create() returns. Contrariwise, the original

thread may run for any amount of time before the new thread is

scheduled. Use a semaphore or some other form of

synchronization if you need to ensure ordering.

The code provided sets the new thread's `priority' member to

PRIORITY, but no actual priority scheduling is implemented.

Priority scheduling is the goal of Problem 1-3. \*/

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;

ASSERT (function != NULL);

/\* Allocate thread. \*/

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

if (t == NULL)

return TID\_ERROR;

/\* Initialize thread. \*/

init\_thread (t, name, priority);

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

/\* 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;

/\* Add to run queue. \*/

thread\_unblock (t);

if (thread\_current()->priority < priority){

thread\_yield();

}

return tid;

}

/\* Puts the current thread to sleep. It will not be scheduled

again until awoken by thread\_unblock().

This function must be called with interrupts turned off. It

is usually a better idea to use one of the synchronization

primitives in synch.h. \*/

void

thread\_block (void)

{

ASSERT (!intr\_context ());

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

thread\_current ()->status = THREAD\_BLOCKED;

schedule ();

}

/\* Transitions a blocked thread T to the ready-to-run state.

This is an error if T is not blocked. (Use thread\_yield() to

make the running thread ready.)

This function does not preempt the running thread. This can

be important: if the caller had disabled interrupts itself,

it may expect that it can atomically unblock a thread and

update other data. \*/

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\_insert\_ordered(&ready\_list, &t->elem, (list\_less\_func \*) &thread\_cmp\_priority, NULL);

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

/\* Returns the name of the running thread. \*/

const char \*

thread\_name (void)

{

return thread\_current ()->name;

}

/\* Returns the running thread.

This is running\_thread() plus a couple of sanity checks.

See the big comment at the top of thread.h for details. \*/

struct thread \*

thread\_current (void)

{

struct thread \*t = running\_thread ();

/\* Make sure T is really a thread.

If either of these assertions fire, then your thread may

have overflowed its stack. Each thread has less than 4 kB

of stack, so a few big automatic arrays or moderate

recursion can cause stack overflow. \*/

ASSERT (is\_thread (t));

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

return t;

}

/\* Returns the running thread's tid. \*/

tid\_t

thread\_tid (void)

{

return thread\_current ()->tid;

}

/\* Deschedules the current thread and destroys it. Never

returns to the caller. \*/

void

thread\_exit (void)

{

ASSERT (!intr\_context ());

#ifdef USERPROG

process\_exit ();

#endif

/\* Remove thread from all threads list, set our status to dying,

and schedule another process. That process will destroy us

when it calls thread\_schedule\_tail(). \*/

intr\_disable ();

list\_remove (&thread\_current()->allelem);

thread\_current ()->status = THREAD\_DYING;

schedule ();

NOT\_REACHED ();

}

/\* Yields the CPU. The current thread is not put to sleep and

may be scheduled again immediately at the scheduler's whim. \*/

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\_insert\_ordered(&ready\_list, &cur->elem, &thread\_cmp\_priority, NULL);

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

/\* Invoke function 'func' on all threads, passing along 'aux'.

This function must be called with interrupts off. \*/

void

thread\_foreach (thread\_action\_func \*func, void \*aux)

{

struct list\_elem \*e;

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

for (e = list\_begin (&all\_list); e != list\_end (&all\_list);

e = list\_next (e))

{

struct thread \*t = list\_entry (e, struct thread, allelem);

func (t, aux);

}

}

/\* Sets the current thread's priority to NEW\_PRIORITY. \*/

void

thread\_set\_priority (int new\_priority)

{

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

if (list\_empty(&thread\_current()->blocked)){

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

thread\_yield();

}

if (new\_priority > thread\_current()->priority\_before\_donate){

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

return;

}

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

//thread\_yield();

}

/\* Returns the current thread's priority. \*/

int

thread\_get\_priority (void)

{

return thread\_current ()->priority\_before\_donate;

}

/\* Sets the current thread's nice value to NICE. \*/

void

thread\_set\_nice (int nice UNUSED)

{

/\* Not yet implemented. \*/

}

/\* Returns the current thread's nice value. \*/

int

thread\_get\_nice (void)

{

/\* Not yet implemented. \*/

return 0;

}

/\* Returns 100 times the system load average. \*/

int

thread\_get\_load\_avg (void)

{

/\* Not yet implemented. \*/

return 0;

}

/\* Returns 100 times the current thread's recent\_cpu value. \*/

int

thread\_get\_recent\_cpu (void)

{

/\* Not yet implemented. \*/

return 0;

}

/\* Idle thread. Executes when no other thread is ready to run.

The idle thread is initially put on the ready list by

thread\_start(). It will be scheduled once initially, at which

point it initializes idle\_thread, "up"s the semaphore passed

to it to enable thread\_start() to continue, and immediately

blocks. After that, the idle thread never appears in the

ready list. It is returned by next\_thread\_to\_run() as a

special case when the ready list is empty. \*/

static void

idle (void \*idle\_started\_ UNUSED)

{

struct semaphore \*idle\_started = idle\_started\_;

idle\_thread = thread\_current ();

sema\_up (idle\_started);

for (;;)

{

/\* Let someone else run. \*/

intr\_disable ();

thread\_block ();

/\* Re-enable interrupts and wait for the next one.

The `sti' instruction disables interrupts until the

completion of the next instruction, so these two

instructions are executed atomically. This atomicity is

important; otherwise, an interrupt could be handled

between re-enabling interrupts and waiting for the next

one to occur, wasting as much as one clock tick worth of

time.

See [IA32-v2a] "HLT", [IA32-v2b] "STI", and [IA32-v3a]

7.11.1 "HLT Instruction". \*/

asm volatile ("sti; hlt" : : : "memory");

}

}

/\* Function used as the basis for a kernel thread. \*/

static void

kernel\_thread (thread\_func \*function, void \*aux)

{

ASSERT (function != NULL);

intr\_enable (); /\* The scheduler runs with interrupts off. \*/

function (aux); /\* Execute the thread function. \*/

thread\_exit (); /\* If function() returns, kill the thread. \*/

}

/\* Returns the running thread. \*/

struct thread \*

running\_thread (void)

{

uint32\_t \*esp;

/\* Copy the CPU's stack pointer into `esp', and then round that

down to the start of a page. Because `struct thread' is

always at the beginning of a page and the stack pointer is

somewhere in the middle, this locates the curent thread. \*/

asm ("mov %%esp, %0" : "=g" (esp));

return pg\_round\_down (esp);

}

/\* Returns true if T appears to point to a valid thread. \*/

static bool

is\_thread (struct thread \*t)

{

return t != NULL && t->magic == THREAD\_MAGIC;

}

/\* Does basic initialization of T as a blocked thread named

NAME. \*/

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;

t->priority = priority;

t->magic = THREAD\_MAGIC;

t->priority\_before\_donate = priority;

t->blocked\_by = NULL;

list\_init(&t->blocked);

old\_level = intr\_disable ();

list\_insert\_ordered(&all\_list, &t->allelem, (list\_less\_func \*) &thread\_cmp\_priority, NULL);

intr\_set\_level (old\_level);

}

/\* Allocates a SIZE-byte frame at the top of thread T's stack and

returns a pointer to the frame's base. \*/

static void \*

alloc\_frame (struct thread \*t, size\_t size)

{

/\* Stack data is always allocated in word-size units. \*/

ASSERT (is\_thread (t));

ASSERT (size % sizeof (uint32\_t) == 0);

t->stack -= size;

return t->stack;

}

/\* Chooses and returns the next thread to be scheduled. Should

return a thread from the run queue, unless the run queue is

empty. (If the running thread can continue running, then it

will be in the run queue.) If the run queue is empty, return

idle\_thread. \*/

static struct thread \*

next\_thread\_to\_run (void)

{

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

return idle\_thread;

else

return list\_entry (list\_pop\_front (&ready\_list), struct thread, elem);

}

/\* Completes a thread switch by activating the new thread's page

tables, and, if the previous thread is dying, destroying it.

At this function's invocation, we just switched from thread

PREV, the new thread is already running, and interrupts are

still disabled. This function is normally invoked by

thread\_schedule() as its final action before returning, but

the first time a thread is scheduled it is called by

switch\_entry() (see switch.S).

It's not safe to call printf() until the thread switch is

complete. In practice that means that printf()s should be

added at the end of the function.

After this function and its caller returns, the thread switch

is complete. \*/

void

thread\_schedule\_tail (struct thread \*prev)

{

struct thread \*cur = running\_thread ();

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

/\* Mark us as running. \*/

cur->status = THREAD\_RUNNING;

/\* Start new time slice. \*/

thread\_ticks = 0;

#ifdef USERPROG

/\* Activate the new address space. \*/

process\_activate ();

#endif

/\* If the thread we switched from is dying, destroy its struct

thread. This must happen late so that thread\_exit() doesn't

pull out the rug under itself. (We don't free

initial\_thread because its memory was not obtained via

palloc().) \*/

if (prev != NULL && prev->status == THREAD\_DYING && prev != initial\_thread)

{

ASSERT (prev != cur);

palloc\_free\_page (prev);

}

}

/\* Schedules a new process. At entry, interrupts must be off and

the running process's state must have been changed from

running to some other state. This function finds another

thread to run and switches to it.

It's not safe to call printf() until thread\_schedule\_tail()

has completed. \*/

static void

schedule (void)

{

list\_sort(&ready\_list, &thread\_cmp\_priority, NULL);

struct thread \*cur = running\_thread ();

struct thread \*next = next\_thread\_to\_run ();

struct thread \*prev = NULL;

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

ASSERT (cur->status != THREAD\_RUNNING);

ASSERT (is\_thread (next));

if (cur != next)

prev = switch\_threads (cur, next);

thread\_schedule\_tail (prev);

}

/\* Returns a tid to use for a new thread. \*/

static tid\_t

allocate\_tid (void)

{

static tid\_t next\_tid = 1;

tid\_t tid;

lock\_acquire (&tid\_lock);

tid = next\_tid++;

lock\_release (&tid\_lock);

return tid;

}

/\* Offset of `stack' member within `struct thread'.

Used by switch.S, which can't figure it out on its own. \*/

uint32\_t thread\_stack\_ofs = offsetof (struct thread, stack);

ПРИЛОЖЕНИЕ Б

Листинг программы «thread.h»

#ifndef THREADS\_THREAD\_H

#define THREADS\_THREAD\_H

#include <debug.h>

#include <list.h>

#include <stdint.h>

/\* States in a thread's life cycle. \*/

enum thread\_status

{

THREAD\_RUNNING, /\* Running thread. \*/

THREAD\_READY, /\* Not running but ready to run. \*/

THREAD\_BLOCKED, /\* Waiting for an event to trigger. \*/

THREAD\_DYING /\* About to be destroyed. \*/

};

/\* Thread identifier type.

You can redefine this to whatever type you like. \*/

typedef int tid\_t;

#define TID\_ERROR ((tid\_t) -1) /\* Error value for tid\_t. \*/

/\* Thread priorities. \*/

#define PRI\_MIN 0 /\* Lowest priority. \*/

#define PRI\_DEFAULT 31 /\* Default priority. \*/

#define PRI\_MAX 63 /\* Highest priority. \*/

/\* A kernel thread or user process.

Each thread structure is stored in its own 4 kB page. The

thread structure itself sits at the very bottom of the page

(at offset 0). The rest of the page is reserved for the

thread's kernel stack, which grows downward from the top of

the page (at offset 4 kB). Here's an illustration:

4 kB +---------------------------------+

| kernel stack |

| | |

| | |

| V |

| grows downward |

| |

| |

| |

| |

| |

| |

| |

| |

+---------------------------------+

| magic |

| : |

| : |

| name |

| status |

0 kB +---------------------------------+

The upshot of this is twofold:

1. First, `struct thread' must not be allowed to grow too

big. If it does, then there will not be enough room for

the kernel stack. Our base `struct thread' is only a

few bytes in size. It probably should stay well under 1

kB.

2. Second, kernel stacks must not be allowed to grow too

large. If a stack overflows, it will corrupt the thread

state. Thus, kernel functions should not allocate large

structures or arrays as non-static local variables. Use

dynamic allocation with malloc() or palloc\_get\_page()

instead.

The first symptom of either of these problems will probably be

an assertion failure in thread\_current(), which checks that

the `magic' member of the running thread's `struct thread' is

set to THREAD\_MAGIC. Stack overflow will normally change this

value, triggering the assertion. \*/

/\* The `elem' member has a dual purpose. It can be an element in

the run queue (thread.c), or it can be an element in a

semaphore wait list (synch.c). It can be used these two ways

only because they are mutually exclusive: only a thread in the

ready state is on the run queue, whereas only a thread in the

blocked state is on a semaphore wait list. \*/

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. \*/

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

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

struct list\_elem elem; /\* List element. \*/

int priority\_before\_donate;

struct list blocked;

struct lock\* blocked\_by;

#ifdef USERPROG

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

uint32\_t \*pagedir; /\* Page directory. \*/

#endif

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

unsigned magic; /\* Detects stack overflow. \*/

};

/\* If false (default), use round-robin scheduler.

If true, use multi-level feedback queue scheduler.

Controlled by kernel command-line option "-o mlfqs". \*/

extern bool thread\_mlfqs;

void thread\_init (void);

void thread\_start (void);

void thread\_tick (void);

void thread\_print\_stats (void);

typedef void thread\_func (void \*aux);

tid\_t thread\_create (const char \*name, int priority, thread\_func \*, void \*);

void thread\_block (void);

void thread\_unblock (struct thread \*);

struct thread \*thread\_current (void);

tid\_t thread\_tid (void);

const char \*thread\_name (void);

void thread\_exit (void) NO\_RETURN;

void thread\_yield (void);

/\* Performs some operation on thread t, given auxiliary data AUX. \*/

typedef void thread\_action\_func (struct thread \*t, void \*aux);

void thread\_foreach (thread\_action\_func \*, void \*);

int thread\_get\_priority (void);

void thread\_set\_priority (int);

int thread\_get\_nice (void);

void thread\_set\_nice (int);

int thread\_get\_recent\_cpu (void);

int thread\_get\_load\_avg (void);

#endif /\* threads/thread.h \*/

ПРИЛОЖЕНИЕ В

Листинг программы «synch.c»

#include "threads/synch.h"

#include <stdio.h>

#include <string.h>

#include "threads/interrupt.h"

#include "threads/thread.h"

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

return list\_entry(a, struct thread, elem)->priority\_before\_donate > list\_entry(b, struct thread, elem)->priority\_before\_donate;

}

/\* Initializes semaphore SEMA to VALUE. A semaphore is a

nonnegative integer along with two atomic operators for

manipulating it:

- down or "P": wait for the value to become positive, then

decrement it.

- up or "V": increment the value (and wake up one waiting

thread, if any). \*/

void

sema\_init (struct semaphore \*sema, unsigned value)

{

ASSERT (sema != NULL);

sema->value = value;

list\_init (&sema->waiters);

}

/\* Down or "P" operation on a semaphore. Waits for SEMA's value

to become positive and then atomically decrements it.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but if it sleeps then the next scheduled

thread will probably turn interrupts back on. \*/

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\_insert\_ordered(&sema->waiters, &thread\_current ()->elem, thread\_compare\_priority, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

/\* Down or "P" operation on a semaphore, but only if the

semaphore is not already 0. Returns true if the semaphore is

decremented, false otherwise.

This function may be called from an interrupt handler. \*/

bool

sema\_try\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

bool success;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (sema->value > 0)

{

sema->value--;

success = true;

}

else

success = false;

intr\_set\_level (old\_level);

return success;

}

/\* Up or "V" operation on a semaphore. Increments SEMA's value

and wakes up one thread of those waiting for SEMA, if any.

This function may be called from an interrupt handler. \*/

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

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

list\_sort(&sema->waiters, &thread\_compare\_priority, NULL);

thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),

struct thread, elem));

}

sema->value++;

thread\_yield();

intr\_set\_level (old\_level);

}

static void sema\_test\_helper (void \*sema\_);

/\* Self-test for semaphores that makes control "ping-pong"

between a pair of threads. Insert calls to printf() to see

what's going on. \*/

void

sema\_self\_test (void)

{

struct semaphore sema[2];

int i;

printf ("Testing semaphores...");

sema\_init (&sema[0], 0);

sema\_init (&sema[1], 0);

thread\_create ("sema-test", PRI\_DEFAULT, sema\_test\_helper, &sema);

for (i = 0; i < 10; i++)

{

sema\_up (&sema[0]);

sema\_down (&sema[1]);

}

printf ("done.\n");

}

/\* Thread function used by sema\_self\_test(). \*/

static void

sema\_test\_helper (void \*sema\_)

{

struct semaphore \*sema = sema\_;

int i;

for (i = 0; i < 10; i++)

{

sema\_down (&sema[0]);

sema\_up (&sema[1]);

}

}

/\* Initializes LOCK. A lock can be held by at most a single

thread at any given time. Our locks are not "recursive", that

is, it is an error for the thread currently holding a lock to

try to acquire that lock.

A lock is a specialization of a semaphore with an initial

value of 1. The difference between a lock and such a

semaphore is twofold. First, a semaphore can have a value

greater than 1, but a lock can only be owned by a single

thread at a time. Second, a semaphore does not have an owner,

meaning that one thread can "down" the semaphore and then

another one "up" it, but with a lock the same thread must both

acquire and release it. When these restrictions prove

onerous, it's a good sign that a semaphore should be used,

instead of a lock. \*/

void

lock\_init (struct lock \*lock)

{

ASSERT (lock != NULL);

lock->holder = NULL;

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

}

/\* Acquires LOCK, sleeping until it becomes available if

necessary. The lock must not already be held by the current

thread.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but interrupts will be turned back on if

we need to sleep. \*/

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

if (!sema\_try\_down(&lock->semaphore)){

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

temp\_thread->blocked\_by = lock;

while (temp\_thread->blocked\_by != NULL){

if (temp\_thread->priority\_before\_donate > temp\_thread->blocked\_by->holder->priority\_before\_donate){//приоритет текущего больше чем приоритет обладателя

temp\_thread->blocked\_by->holder->priority\_before\_donate = temp\_thread->priority\_before\_donate;//приоритет обладателя замка будет равен приоритету текущего процесса

temp\_thread = temp\_thread->blocked\_by->holder;

}

else

break;

}

sema\_down(&lock->semaphore);

}

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

lock->holder = thread\_current();//текущий поток - держатель замка

list\_push\_back(&thread\_current()->blocked, &lock->locked);

thread\_current()->blocked\_by = NULL;

intr\_set\_level(last);

//sema\_down (&lock->semaphore);

//lock->holder = thread\_current ();

}

/\* Tries to acquires LOCK and returns true if successful or false

on failure. The lock must not already be held by the current

thread.

This function will not sleep, so it may be called within an

interrupt handler. \*/

bool

lock\_try\_acquire (struct lock \*lock)

{

bool success;

ASSERT (lock != NULL);

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

success = sema\_try\_down (&lock->semaphore);

if (success)

lock->holder = thread\_current ();

return success;

}

/\* Releases LOCK, which must be owned by the current thread.

An interrupt handler cannot acquire a lock, so it does not

make sense to try to release a lock within an interrupt

handler. \*/

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

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

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

list\_remove(&lock->locked);

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

if (!list\_empty(&cur\_thread->blocked)){

struct thread \*wait\_th;

struct list\_elem \*blocked, \*ready;

struct lock \*lock\_curr;

int max = cur\_thread->priority;

for (blocked = list\_begin(&cur\_thread->blocked); blocked != list\_end(&cur\_thread->blocked); blocked = list\_next(blocked)){

lock\_curr = list\_entry(blocked, struct lock, locked);

for (ready = list\_begin(&lock\_curr->semaphore.waiters); ready != list\_end(&lock\_curr->semaphore.waiters); ready = list\_next(ready)){

wait\_th = list\_entry(ready, struct thread, elem);

if (wait\_th->priority\_before\_donate > max)

max = wait\_th->priority\_before\_donate;

}

}

cur\_thread->priority\_before\_donate = max;

}

else

cur\_thread->priority\_before\_donate = cur\_thread->priority;

lock->holder = NULL;

intr\_set\_level(last);

sema\_up(&lock->semaphore);

}

/\* Returns true if the current thread holds LOCK, false

otherwise. (Note that testing whether some other thread holds

a lock would be racy.) \*/

bool

lock\_held\_by\_current\_thread (const struct lock \*lock)

{

ASSERT (lock != NULL);

return lock->holder == thread\_current ();

}

/\* One semaphore in a list. \*/

struct semaphore\_elem

{

struct list\_elem elem; /\* List element. \*/

struct semaphore semaphore; /\* This semaphore. \*/

};

bool sema\_cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED){

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

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

return (first->semaphore.priority > second->semaphore.priority);

}

/\* Initializes condition variable COND. A condition variable

allows one piece of code to signal a condition and cooperating

code to receive the signal and act upon it. \*/

void

cond\_init (struct condition \*cond)

{

ASSERT (cond != NULL);

list\_init (&cond->waiters);

}

/\* Atomically releases LOCK and waits for COND to be signaled by

some other piece of code. After COND is signaled, LOCK is

reacquired before returning. LOCK must be held before calling

this function.

The monitor implemented by this function is "Mesa" style, not

"Hoare" style, that is, sending and receiving a signal are not

an atomic operation. Thus, typically the caller must recheck

the condition after the wait completes and, if necessary, wait

again.

A given condition variable is associated with only a single

lock, but one lock may be associated with any number of

condition variables. That is, there is a one-to-many mapping

from locks to condition variables.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but interrupts will be turned back on if

we need to sleep. \*/

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.priority = thread\_current()->priority\_before\_donate;

list\_insert\_ordered(&cond->waiters, &waiter.elem, &sema\_cmp\_priority, NULL);

lock\_release (lock);

sema\_down (&waiter.semaphore);

lock\_acquire (lock);

}

/\* If any threads are waiting on COND (protected by LOCK), then

this function signals one of them to wake up from its wait.

LOCK must be held before calling this function.

An interrupt handler cannot acquire a lock, so it does not

make sense to try to signal a condition variable within an

interrupt handler. \*/

void

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

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

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

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

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

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

}

}

/\* Wakes up all threads, if any, waiting on COND (protected by

LOCK). LOCK must be held before calling this function.

An interrupt handler cannot acquire a lock, so it does not

make sense to try to signal a condition variable within an

interrupt handler. \*/

void

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

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

while (!list\_empty (&cond->waiters))

cond\_signal (cond, lock);

}

ПРИЛОЖЕНИЕ г

Листинг программы «synch.h»

#ifndef THREADS\_SYNCH\_H

#define THREADS\_SYNCH\_H

#include <list.h>

#include <stdbool.h>

/\* A counting semaphore. \*/

struct semaphore

{

unsigned value; /\* Current value. \*/

struct list waiters; /\* List of waiting threads. \*/

int priority;

};

void sema\_init (struct semaphore \*, unsigned value);

void sema\_down (struct semaphore \*);

bool sema\_try\_down (struct semaphore \*);

void sema\_up (struct semaphore \*);

void sema\_self\_test (void);

/\* Lock. \*/

struct lock

{

struct list\_elem locked;

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

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

};

void lock\_init (struct lock \*);

void lock\_acquire (struct lock \*);

bool lock\_try\_acquire (struct lock \*);

void lock\_release (struct lock \*);

bool lock\_held\_by\_current\_thread (const struct lock \*);

/\* Condition variable. \*/

struct condition

{

struct list waiters; /\* List of waiting threads. \*/

};

void cond\_init (struct condition \*);

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

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

void cond\_broadcast (struct condition \*, struct lock \*);

bool sema\_cmp\_priority(const struct list\_elem \*, const struct list\_elem \*, void \*);

bool thread\_compare\_priority(const struct list\_elem\*, const struct list\_elem\*, void\*);

/\* Optimization barrier.

The compiler will not reorder operations across an

optimization barrier. See "Optimization Barriers" in the

reference guide for more information.\*/

#define barrier() asm volatile ("" : : : "memory")

#endif /\* threads/synch.h \*/