Лабораторная работа 2
ПЛАНИРОВАНИЕ ПРОЦЕССОВ

Цель лабораторной работы — изучение механизмов планирования процессов, разработка алгоритма приоритетного планирования и внедрение разработанного алгоритма в учебную операционную систему Pintos.

Основные рабочие файлы: threads/thread.c, threads/synch.c

Теоретические сведения

Система планирования в ОС Pintos является частью системы управления процессами и отвечает за распределение разделяемых ресурсов, таких как процессорное время. Исходные коды системы планирования, аналогично предыдущей лабораторной работе, расположены в директории threads. Помимо функций работы с процессами, ознакомление с которыми произошло при внесении модификаций в реализацию системного таймера, при выполнении данной лабораторной работы необходимо изучить предложенные системой Pintos примитивы синхронизациии механизмы переключения контекста процессов (табл. 2.1). Рабочей директорией данной лабораторной работы по-прежнему является каталог src/threads.

Таблица 2.1. Исходные коды директории src/threads

switch.S switch.h	Реализация механизма переключения контекста процессов (написана на языке ассемблера с подробными комментариями).
synch.c synch.h	Примитивы синхронизации Pintos: семафоры, замки, мониторы.

Также более подробно стоит рассмотреть функции диспетчеризации процессов, расположенные в src/threads/thread.c.

Алгоритмы планирования

First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served. Все процессы, находящиеся в состоянии готовность, выстроены в очередь: когда процесс переходит в состояние «готовность», он помещается в конец этой очереди, выбор нового процесса для исполнения осуществляется из начала очереди. Такой алгоритм выбора процессов на исполнение осуществляет невытесняющее планирование. Выбранный процесс использует процессор до завершения своей работы (до истечения своего текущего CPU burst). После этого для выполнения выбирается новый процесс из начала очереди.

Преимуществом алгоритма является простота реализации. Недостатком алгоритма является зависимость среднего времени ожидания и среднего полного времени выполнения от порядка расположения процессов в очереди. При наличии длительного процесса короткие процессы будут длительное время находиться в состоянии «готовность» и ждать своей очереди. Поэтому алгоритм FCFS практически неприменим для систем разделения времени

Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (RR). RR реализован в режиме вытесняющего планирования. 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. Для очередного исполнения выбирается процесс из начала очереди и устанавливается таймер для генерации прерывания по истечении кванта времени. Если время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени, процесс освобождает процессор до истечения кванта времени. На исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта. Если продолжительность остатка текущего CPU burst процесса больше отведенного кванта времени, то по истечении кванта времени процесс прерывается таймером, помещается в конец очереди процессов, находящихся в состоянии «готовность». Процессор выделяется процессу, находящемуся в начале очереди.

При больших величинах кванта времени, процесс завершает свой CPU burst до его окончания, алгоритм RR вырождается в алгоритм FCFS. При малых величинах кванта времени каждый из n процессов работает с производительностью ~ 1/n от производительности реального процессора. Это справедливо при пренебрежении времени переключения контекста процессов. Частое переключение контекста увеличивает накладные расходы системы на переключение. Производительность системы резко падает.

Shortest-Job-First (SJF)

Эффективность работы алгоритмов FCFS и RR зависит от порядка расположения в очереди процессов, находящихся в состоянии «готовность». Их производительность возрастает при расположении коротких процессов в начале очереди. Алгоритм Shortest Job First (SJF) ("кратчайшая работа первой") использует данное свойство. Для исполнения выбирается процесс с минимальной длительностью CPU burst при известном времени CPU burst процессов, находящихся в состоянии «готовность». Если таких процессов несколько для выбора можно использовать алгоритм FCFS. Квантование времени при этом не происходит.

SJF-алгоритм может быть как вытесняющим, так и невытесняющим. При невытесняющем алгоритме процессор предоставляется избранному процессу на необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса.

Базовая система планирования

В каждый момент времени в ядре операционной системы Pintos может выполняться только один процесс, а остальные, если они существуют, становятся неактивными. Планировщик принимает решение, какому процессу выделить процессор. Если в данный моментв системе нет готовых к выполнению процессов, то планировщик выбирает специальный "пустой" процесс, который релизован в функции idle(). Примитивы синхронизации могут вносить изменения в порядок выполнения процессов, например, когда один процесс ждет результатов выполнения другого процесса или освобождения некоторых ресурсов.

Механизм переключения контекста процессов реализован в threads/switch.S на языке ассемблера. Этот код не является обязательным к изучению, но необходимо понимать, что именно там происходит сохранение состояния текущего процесса и восстановление состояния процесса, который будет выполняться следующим. Функция switch_threads() помещает состояние регистров и указателя вершины стека в соответствующее поле структуры thread текущего процесса и восстанавливает эти данные у следующего процесса.

Функция schedule() отвечает за планирование процессов. Она расположена в threads/thread.c и может быть вызывана только из трех функций: thread_block() - заблокировать процесс, thread_exit() - завершить процесс и thread_yield() - освободить процессор от выполнения текущего потока и перейти к выполнению какого-либо другого. Перед тем, как обратиться к функции schedule(), эти функции отключают системные прерывания (или убеждаются, что они уже отключены) и меняют состояние процесса с THREAD_RUNNING на какое-либо другое.

Функция schedule(), несмотря на свой малый размер, выполняет ряд важнейших задач:

```
static void schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    if (cur != next)
        prev = switch_threads (cur, next);
    thread_schedule_tail (prev);
}
```

Она записывает текущий процесс в локальную переменную cur, определяет процесс, который будет выполняться следующим в локальной переменной next, и затем вызывает фунцию switch_threads(), чтобы осуществить переключение контекста. Остальная часть планировщика реализована в функции thread_schedule_tail(), которая помечает новый процесс как выполняющийся.

В ОС Pintos каждый процесс имеет небольшой стек размером 4 Кб. Ядро старается защищать его от переполнения, но не всегда справляется к этой задачей. Поэтому объявление больших структур данных и нестатических локальных переменных, например int buf[1000], может привести к ошибкам времени выполнения.

Синхронизация

Надлежащая синхронизация является важнейшим компонентом любой системы управления процессами. В общем случае, любая проблема сихронизации процессов может быть решена отключением системных прерываний. В базовой реализации ОС Pintos задача синхронизации процессов решена именно таким образом, но для современных операционных систем данный подход является неприемлемым.

Когда прерывания выключены, никакой другой процесс не сможет вытеснить выполняющийся процесс. В случае, когда прерывания включены, выполняющийся процесс может быть вытеснен другим в любой момент. Функции для работы с прерываниями, расположенные в threads/interrupt.h, представлены в табл. 2.2.

Таблица 2.2. Функции работы с прерываниями

Объект:	**Назначение:**
enum intr_level	Переменная перечислимого типа указывает на состояние прерываний
enum intr_level intr_get_level (void)	Функция возвращает текущее состояние механизма прерываний (включены, выключены)
enum intr_level intr_enable (void)	Функция включает прерывания, возвращает предыдущее состояние механизма прерываний.
enum intr_level intr_disable (void)	Функция отключает прерывания, возвращает предыдущее состояние механизма прерываний

Для решения проблем синхронизации используются такие примитивы, как семафоры, замки и мониторы. Примитивы синхронизации ОС Pintos представлены в threads/synch.c. Подробное описание семафоров представлено в лабораторной работе 3.

Замки

Замок является аналогом семафора с заданным значением value, равным 1. Эквивалентом операции "V" является операция "release" (освободить), а эквивалентом "P" - операция "acquire" (овладеть). По сравнению с семафором замок имеет одно важное ограничение: только процесс, который выполнил операцию "acquire"(и называется обладателем замка), имеет право выполнить операцию "release".

Реализация замков в ОС Pintos не является рекурсивной, т.е. при попытке процессом, который в данный момент обладает замком и находится внутри критической секции, повторно выполнить операцию "acquire", возникает исключение времени выполнения.

Реализация замков для ОС Pintos, расположенная в threads/synch.h, описана в табл. 2.3.

Таблица 2.3. Замки

Объект:	**Назначение:**

struct lock	Структура описывает основные параметры замка:
	struct lock
	{
	struct thread *holder; //процесс-обладательзамка
	struct semaphore semaphore;
	};

void lock_init (struct lock *lock)	Функция инициализирует новый замок. Процесс, создающий замок, по умолчанию не является его обладателем
void lock_acquire (struct lock *lock)	Функция выполняет операцию "acquire" над данным замком, ожидая операцию "release" от предыдущего обладателя замка, если это требуется. Процесс, выполнивший эту операцию, становится обладателем замка
bool lock_try_acquire (struct lock *lock)	Функция выполняет операцию "acquire" без ожидания. Возвращает значение "true", если операция выполнена успешно, или "false", если у замка уже есть обладатель
void lock_release (struct lock *lock)	Функция выполняет операцию "release" над замком, если текущий процесс является его обладателем
bool lock_held_by_current_thread (const struct lock *lock)	Функция возвращаетзначение "true", если текущий процесс является обладателем замка, и "false" в противном случае

Мониторы

Монитор — это логическая конструкция, представляющая собой структуру, которая содержит закрытые переменные (разделяемые ресурсы) и функции для работы с данными переменными. Мониторы решают проблему синхронизации, используя замки и переменные-условия. Аналогично семафорам, перед тем, как получить доступ к разделяемым ресурсам, процесс овладевает замком данного монитора и становится единственным процессом, который имеет право работать с ресурсами. Когда процесс завершает работу с разделяемыми ресурсами, он освобождает замок. Отличие монитора от семафоров заключается в том, что программисту не нужно вручную вызывать функции "release" и "acquire", что снижает количество ошибок программирования и тем самым исключает возможность возникновения взаимных блокировок.

Использование в мониторах опциональных переменных-условий позволяет процессу, который находится внутри монитора, ожидать выполнения некоторого внешнего условия. Каждая переменная связана с некоторым условием, например, освобождением некоторого буфера или завершением работы некоторого процесса. Когда процессу, находящемуся внутри монитора, требуется выполнение некоторого условия, он вызывает функцию "wait" (т.е. ждет соответствующую ему переменную-условие), блокируется и позволяет другим процессам войти в монитор. Когда действия некоторого процесса, находящегося в мониторе, приведут к наступлению ожидаемого события (например, процесс освободит указанный буфер), процесс выполняет операцию "signal", тем самым сигнализируя заблокированному процессу о наступлении ожидаемого им события. Далее блокированный процесс пробуждается и продолжает свою работу внутри монитора.

Реализация мониторов для ОС Pintos, расположенная в threads/synch.h, описана в табл. 2.4.

Таблица 2.4. Мониторы

Объект:	**Назначение:**

struct condition	Структура описывает переменную-условие:
	structcondition
	{
	struct list waiters; //список ожидающих процессов
	};

void cond_init (struct condition *cond)	Функция инициализирует переменную cond как новую переменную-условие
void cond_wait (struct condition *cond, struct lock *lock	Функция выполняетоперацию"release" над замком монитора и ожидает, когда переменная-условие получит сигнал о выполнении условия. Замок монитора должен быть закрыт до выполнения данной функции
void cond_signal (struct condition *cond, struct lock *lock)	Если несколько процессов ожидают переменную-условие, функция пробуждает один из них
void cond_broadcast (struct condition *cond, struct lock *lock)	Если несколько процессов ожидают переменую-условие, пробуждает все.

Порядок выполнения работы

1. Проанализировать текущий алгоритм работы планировщика ОС Pintos. Для этого необходимо изучить исходные коды системы планирования и синхронизации процессов, расположенные в threads/thread.c и threads/synch.c.

2. Изобразить диаграмму исполнения процессов в соответствии с полученным вариантом для текущего алгоритма планирования ОС Pintos (до внесения ваших изменений в код планировщика). Алгоритм ОС Pintos не учитывает приоритет и CPU Burst процессов, выполняя все процессы по-очереди.

3. 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.

4. Изобразить блок-схему нового, разработанного алгоритма планирования.

6. Реализовать новый алгоритм планирования. Для этого нужно внести соответствующие изменения в код существующей системы планирования (threads/thread.c и threads/synch.c).

5. Составить таблицу (привести в отчете), в которой указать функции, которые участвуют в планировании процессов (аргументы, которые принимают эти функции, и возвращаемые ими значения), а также связь этих функций друг с другом.

6. Построить диаграмму исполнения процессов для вашего варианта задания, соответствующую новому разработанному алгоритму планирования. Каждый из процессов выполняется указанное количество «тиков» (столбец CPU Burst), а затем завершается. Провести сравнение диаграммы с диаграммой из п. 2, выделить и пояснить изменения.

7. Реализовать механизм разделения приоритета. Необходимо смоделировать следующую абстрактную ситуацию: Пусть некоторый процесс с высоким приоритетом В ждет входа в критическую секцию, которая в данный момент занята процессом с низким приоритетом Н. Тем временем в очереди готовых к выполнению процессов находится процесс со средним приоритетом С. Естественно, процессор будет выбирать процесс С из очереди готовых и, таким образом, процесс Н не получит процессорного времени и не освободит критическую секцию для процесса с высоким приоритетом В. Решением этой проблемы является механизм разделения приоритета: процесс с высоким приоритетом В должен «поделиться» своим приоритетом с процессом Н и «забрать» его обратно, когда процесс Н освободит критическую секцию. Необходимо смоделировать другие подобные возможные ситуации, в которых может потребоваться разделение приоритета, и внести изменения в примитивы синхронизации, размещенные в threads/synch.h и threads/synch.c.

8. Проверить разработанный алгоритм планирования на наличие «тупиков» синхронизации и устранить их. Осуществить оценку разработанного решения с помощью следующих тестов:

- alarm-priority
- priority-change
- priority-fifo
- priority-preempt
- priority-sema
- priority-condvar

Тестирование механизма разделения приоритета:

- priority-donate-one
- priority-donate-lower
- priority-donate-multiple
- priority-donate-multiple2
- priority-donate-sema
- priority-donate-nest
- priority-donate-chain

9*. Реализованный вами новый планировщик ОС Pintos учитывает приоритет процессов, но не учитывает значение CPU Burst и работает по алгоритму Round Robin. Необходимо разработать тест test-new-alg в файле tests/threads/test-new-alg.c, в котором были бы созданы процессы в соответствие с вашим вариантом задания, и при этом каждому из процессов при создании был бы указан приоритет и CPU Burst — количество тиков, которые он должен отработать и после этого завершиться. При этом переключение процессов должно осуществляться по указанному в варианте алгоритму (это может быть не Round Robin). Тест должен:
— увеличить приоритет текущего процесса (свой) до максимального (чтобы другие создаваемые им процессы не могли начать работать). Если это необходимо для вашего варинта, увеличьте значение PRI_MAX до 64 в коде ОС Pintos;
— создать несколько процессов, указывая при этом приоритет и CPU burst каждого создаваемого процесса (в соответствии с вариантом задания); для решения этой задачи потребуется реализовать дополнительную функцию, подобную thread_create, но принимающую на вход еще один дополнительный параметр — значение cpu_burst;
— понизить свой приоритет до минимального (при этом созданные процессы начнут работать);
— каждый из созданных процессов должен выполняться в бесконечном цикле периодически выводя при этом свое имя и диагностическую информацию: сколько тиков отработал процесс;
— главная функция теста ожидает завершения всех запущенных процессов, и на этом тест завершается; поскольку главный процесс теста имеет низший приоритет, то каких-либо дополнительных действий по ожиданию завершения созданных процессов требоваться не должно.

Созданные процессы должны переключаться по указанному в вашем варианте задания алгоритму планирования, с учетом приоритета. Для реализации этого функционала потребуется, вероятно, внести некоторые изменения в системный алгоритм планирования. Если в вашем варианте указан алгоритм, отличный от Round Robin, то эти изменения могут повлечь некорректную работу предусмотренных в ОС Pintos тестов. Для сохранения работоспособности можно предусмотреть некоторую глобальную переменную, изменяющую тип глобального алгоритма планирования. Ваш тест test-new-alg при запуске может изменить ее значение (напрямую, или с помощью специальной функции) — тем самым изменить поведение планировщика со стандартного (Round Robin) на другой

(SJF, FCFS и т.д.), соответствующий вашему алгоритму. В коде функций планировщика необходимо проверять значение этой глобальной переменной: для стандартных тестов ОС Pintos эта переменная останется в режиме RR, для вашего теста — в режиме алгоритма вашего варианта задания.

Проанализируйте результат, полученный при запуске теста, сделайте выводы.

Содержание отчета

В отчете необходимо указать следующие данные:

1. Диаграммы исполнения процессов:
   — для исходного алгоритма планирования pintos;
   — для алгоритма, разработанного в процессе выполнения работы.
2. Сравнительный анализ диаграмм.
3. Блок-схема нового разработанного алгоритма планирования.
4. Описание функций, которые применяются для организации планирования: их назначение, описание аргументов, возвращаемого значения, связей друг с другом.
5. Исходные коды системы планирования Pintos с внесенными модификациями и комментариями.
6. Описание тестовых задач: имя теста и описание действий, которые в нем совершаются; полученные при запуске результаты, анализ полученных результатов.
7. Описание теста test-new-alg, полученный вывод и анализ полученного вывода.