Лекция 5: Потоки (Threads)

Алексей Линёв
Александр Мощук

somes slides are adapted from the OS course at the University of Washington

Объявления

- Домашнее задание №1
  - сдача сегодня (сейчас!)
- Проект №1
  - сдача в среду, 18:00...
- Расписание:
  - Семинар по проекту №1 сегодня
    - После лекции...
  - Лекция №6 переносится со вторника на среду
    - Вторник – только работа над проектом №1
  - Лекция №7 переносится с четверга на пятницу
  - Обновленное расписание на web-странице

Из чего состоит процесс?

- Процесс включает
  - адресное пространство
  - код выполняющейся программы
  - данные выполняющейся программы
  - стек и указатель на его вершину (SP)
    - позволяет отследить историю вызовов процедур
  - программный счетчик, указывающий на следующую инструкцию (PC)
  - регистры общего назначения и их значения
  - множество ресурсов ОС
    - открытые файлы, сетевые соединения, звуковые каналы,…
- Множество разноплановых концепций объединены в одном понятии!
- Сегодня:
  - потоки/нити исполнения (threads)

Параллелизм

- Представьте себе web-сервер, который должен обрабатывать несколько запросов одновременно
  - Во время ожидания подтверждения покупки от сервера центра обработки данных по кредитным карточкам, web-сервер может считывать с диска данные, запрошенные вторым клиентом, и формировать ответ для третьего клиента на основании кэшированной информации
- Представьте web-браузер, который хочет выполнить одновременно несколько запросов
  - Домашняя страница нашего курса содержит десятки html команд "src=…", каждая из которых займет какое-то время для загрузки. Было бы хорошо скачать все их параллельно…
- Представьте параллельную программу, выполняющуюся на многопроцессорной машине, которая хочет использовать «аппаратную параллельность»
  - Например, при умножении матриц можно разбить результирующую матрицу на k регионов и вычислять значения из каждого региона на k процессорах параллельно

Что нам требуется?

- В каждом из этих примеров параллельности (web-сервер, web-клиент, параллельная программа)
  - Выполняется один и тот же код
  - Выполняется доступ к одним и тем же данным
  - Используется один и тот же уровень привилегий
  - Используются одно и те же множество ресурсов (открытые файлы, сетевые соединения,…)
- Но – используются различные аппаратные контексты!
  - стек и указатель на вершины стека (SP)
    - содержит историю вызовов процедур
  - программный счетчик, указывающий на следующую инструкцию (PC)
  - регистры общего назначения и их значения

Как этого достичь?

- Если использовать процессы в том виде, как мы их знаем:
  - создаем несколько процессов, используя fork()
  - в каждом отображаем на адресное пространство одну и ту же физическую память для разделения данных
    - например, посредством использования системного вызова shmget()
- Такой способ крайне неэффективен
  - память: требуется размещать УБП, таблицы страниц и т.д.
  - время: создание внутренних структур ОС, обработка fork(), копирование адресных пространств и т.д.
- Некоторые другие неудачные альтернативы
  - полностью независимые процессы - web-сервера
  - вручную запрограммированная асинхронная работа web-клиента, использующая неблокирующий ввод-вывод

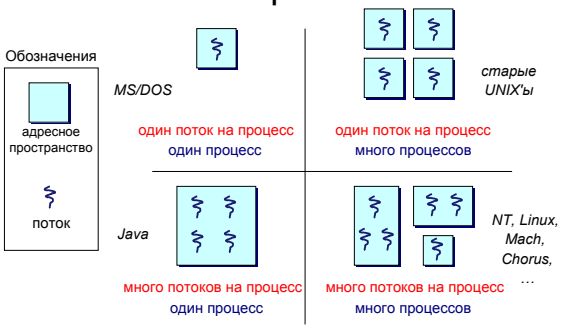
Как сделать все это лучше?

- Основная идея
  - разделить понятие процесса (адресное пространство и пр.)
  - … и понятие «потока управления» / «потока исполнения» (состояние выполнения, аппаратный контекст: PC, SP и т.д.)
- Эта исполняющаяся сущность обычно называется *потоком*/*нитью* (thread), иногда – *облегченным процессом* (lightweight process)

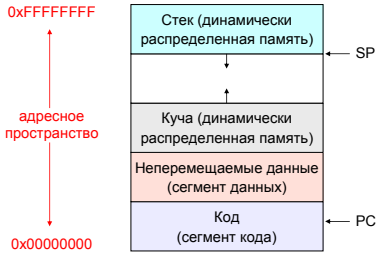
Процессы и потоки

- Большинство современных ОС (Mach, Chorus, NT, современные UNIX'ы) поддерживают два вида сущностей:
  - процесс – объект, определяющий адресное пространство, атрибуты и выделенные ресурсы (открытые файлы и пр.)
  - поток – объект, представляющий последовательный поток исполнения команд в рамках процесса
- Поток привязан к конкретному процессу
  - процесс может иметь множество исполняющихся внутри него потоков
  - разделение данных между потоками реализуется автоматически – они просто используют одно адресное пространство!
  - создание потока – также очень дешевая операция!
- Именно потоки становятся объектами планирования
  - а процессы – это всего лишь контейнеры, в которых выполняются потоки

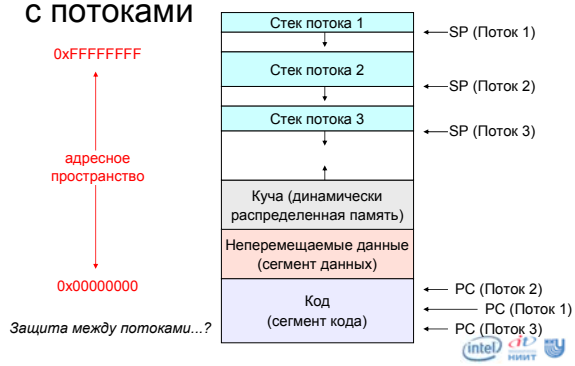
Возможные варианты



Адресное пространство процесса (раньше)



Адресное пространство процесса с потоками



Защита между потоками…?

Разложение на процессы и потоки

- Параллелизм (многопоточность) полезна при
  - параллельной обработке событий (например, в web-серверах и клиентах)
  - разработке параллельных программ (например, умножения матриц)
  - улучшении структуры программы (например, Java-программы)
- Многопоточность полезна даже на однопроцессорных системах
  - даже если за раз может выполняться только один поток
    - Может ли многопоточная программа выполняться быстрее однопоточной в этой ситуации?
- Поддержка многопоточности, то есть разделение понятий процесса (адресное пространство, ресурсы и т.д.) и потока управления (состояние выполнения) – большое достижение
  - параллельное исполнение не требует создания новых процессов
  - "быстрее / лучше / проще"

Интерфейс работы с потоками

- POSIX `pthreads` API:
  - `t = pthread_create(attributes, start_procedure)`
    - создание нового потока управления
    - новый поток начинает выполнение функции start_procedure
  - `pthread_exit()`
    - завершение вызвавшего потока
  - `pthread_wait(t)`
    - ожидание завершения указанного потока
  - `pthread_cond_wait(condition_variable)`
    - вызывающий поток переходит в состояние ожидания до наступления события `condition_variable`
  - `pthread_signal(condition_variable)`
    - пробуждение потока, ожидающего наступления события

“Откуда появляются потоки?”

- Естественный ответ: за создание/управление потоками ответственно ядро ОС
  - вызов ядра, создающий потоки, выполняет следующие действия:
    - формирует в адресном пространстве процесса стек для нового потока
    - создает и инициализирует Управляющий Блок Потока (Дескриптор Потока) – Thread Control Block, TCB
      - указатель на вершину стека, программный счетчик, значения регистров
    - помещает поток в очередь готовых к выполнению
  - мы будем называть такие потоки *потоками ядра* (kernel threads)

“Откуда появляются потоки?”

- Потоки также могут управляться на уровне пользователя (то есть, полностью внутри процесса)
  - Английский термин: user-level threads
  - управление потоками производится библиотекой, прилинкованной к программе
    - поскольку потоки разделяют одно адресное пространство, менеджер потоков не должен управлять адресными пространствами (что возможно только в режиме ядра)
    - потоки различаются (в первом приближении) только аппаратным контекстом (PC, SP, значения регистров), что можно изменять и в режиме пользователя
    - библиотека поддержки многопоточности мультиплексирует потоки пользовательского уровня “над ядром”

Потоки ядра / kernel threads

- Итак, теперь ОС управляет потоками *и* процессами
  - все операции над потоками реализованы в ядре
  - ОС планирует выполнение всех потоков в системе
    - если один из потоков процесса блокируется (например, при выполнении ввода-вывода), ОС знает об этом и может выполнять другие потоки того же процесса
    - таким образом, возможно перекрытие вычислений и ввода-вывода даже *в рамках одного процесса*
- Работа с потоками в ядре проще и быстрее, чем работа с процессами
  - структура, которую надо выделять и заполнять, существенно меньше
- Но использование потоков все-таки слишком дорого для их использования в мелочах (например, создание потока - намного более затратное действие, чем вызов функции)
  - операции над потоками – это системные вызовы
    - переходы в ядро
    - проверка аргументов
  - необходимо в ядре сохранять кое-какие данные для каждого потока

Потоки пользовательского уровня

- Чтобы сделать потоки простыми и быстрыми – реализуйте их на пользовательском уровне
  - они будут полностью управляться библиотекой пользовательского уровня
- Потоки пользовательского уровня малы и быстры
  - каждый поток представлен просто программным счетчиком, значениями регистров, стеком и простым дескриптором потока (ДПт = thread control block = TCB)
  - создание потока, переключение между потоками и синхронизация выполнения потоков выполняется посредством вызова процедур
    - нет необходимости вызывать ядро!
  - в результате, операции над потоками могут быть в 10-100 раз быстрее, чем операции с потоками ядра

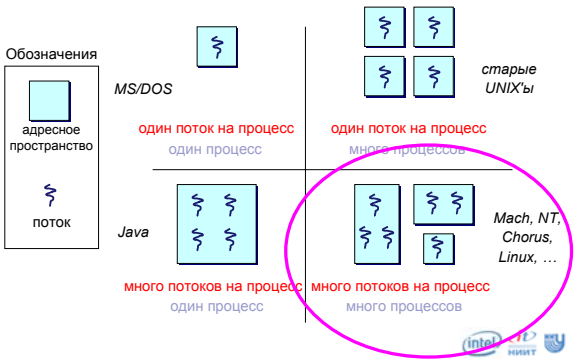
Пример оценки производительности

- Pentium 700MHz, Linux 2.2.16
  - Процессы
    - **`fork/exit`**: 251 мкс
  - Потоки ядра
    - **`pthread_create()/pthread_join()`**: 94 мкс
      (в 2.5 раза быстрее)
  - Потоки пользовательского уровня:
    - **`pthread_create()/pthread_join()`**: 4.5 мкс
      (еще в 20 раз быстрее)

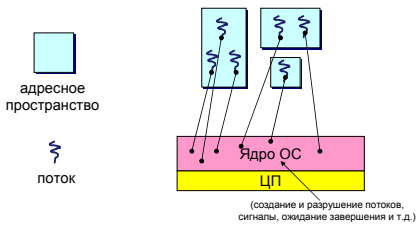
Пример оценки производительности

- Pentium 700MHz, Linux 2.2.16
- DEC SRC Firefly, Ultrix, 1989
  - Процессы
    - `fork/exit`: 251 мкс / 11,300 мкс
  - Потоки ядра
    - `pthread_create()/pthread_join()`: 94 мкс / 948 мкс **(в 12 раз быстрее)**
  - Потоки пользовательского уровня
    - `pthread_create()/pthread_join()`: 4.5мкс / 34 мкс **(еще в 28 раз быстрее)**

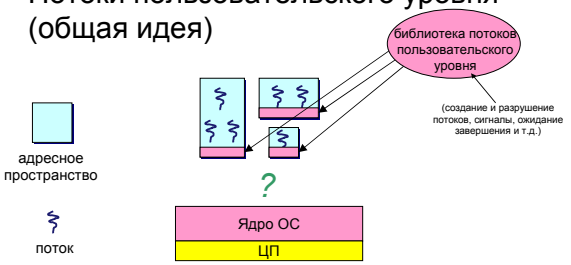
Возможные варианты



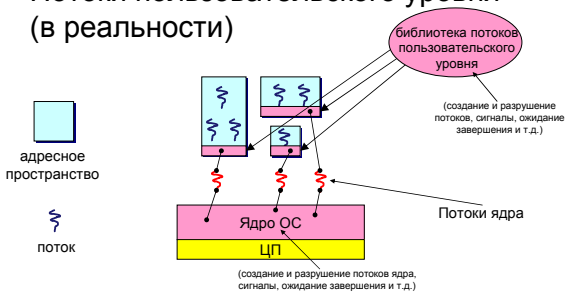
Потоки ядра



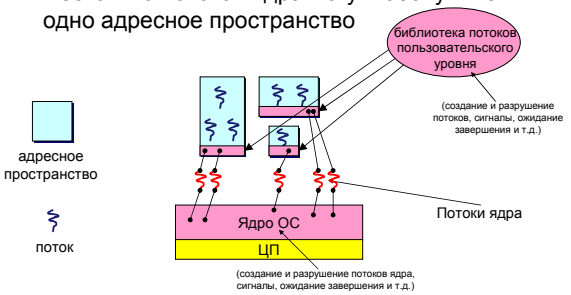
Потоки пользовательского уровня (общая идея)



Потоки пользовательского уровня (в реальности)



Несколько потоков ядра могут "обслуживать" одно адресное пространство



Реализация потоков пользовательского уровня

- Ядро предполагает, что пользовательский процесс – это просто обычный процесс, исполняющий свой код
  - А этот код содержит библиотеку поддержки исполнения потоков и собственный планировщик выполнения потоков
- Планировщик выполнения потоков определяет очередность предоставления ЦП потокам
  - он использует очереди для потоков в различных состояниях: выполняющихся, готовых к выполнению, ожидающих
    - в точности так же, как при управлении процессами в ОС
    - но реализованное в виде библиотеки пользовательского уровня

Как избежать узурпации процессора потоком пользовательского уровня?

- Стратегия 1: призвать всех к сотрудничеству
  - каждый поток добровольно отдает ЦП, вызывая **`yield()`**
  - **`yield()`** вызывает планировщик, который переключает контекст исполнения и запускает другой поток, готовый к выполнению
  - что произойдет, если поток никогда не вызывает **`yield()`** ?
- Стратегия 2: использовать *вытеснение* (*preemption*)
  - планировщик запрашивает у ОС доставку ему прерываний от таймера с указанным периодом
    - обычно доставка реализуется в виде посылки сигнала UNIX (`man signal`)
    - сигналы очень похожи на программные прерывания, но сигналы приходят к пользовательским приложениям от ОС, а прерывания – от аппаратного обеспечения к ОС
  - при каждом прерывании планировщик получает управление и переключает контекст исполнения по своему усмотрению

Переключение контекста потока

- Очень просто для потоков пользовательского уровня
  - сохранить контекст выполняющегося потока
    - состояние аппаратного обеспечения помещается на вершину стека выполняющегося потока
  - поменять указатель стека
    - указатель стека (SP) устанавливается на стек потока, выбранного для исполнения
  - восстановить контекст следующего потока
    - состояние аппаратного обеспечения восстанавливается с вершины стека потока, выбранного для исполнения
  - выполнить возврат (return) уже в новом потоке
    - исполнение продолжится со следующей инструкции выбранного потока
- Переключение контекста нужно программировать на ассемблере
  - переключение работает на уровне соглашений о вызове процедур
    - таким образом, оно не может реализовано через вызовы процедур

Что произойдет, если поток попытается выполнить ввод-вывод ?

- Поток ядра, обеспечивающий выполнение потоков пользовательского уровня, перейдет в состояние ожидания и будет "потерян" на время выполнения синхронной операции ввода-вывода!
- Можно использовать для каждого потока пользовательского уровня поток ядра
  - нет реального отличия от потоков ядра, но операции общего назначения (например, синхронизация) будут быстрыми
- Можно иметь ограниченный набор потоков ядра, обеспечивающих все потоки пользовательского уровня одного процесса
  - ядро будет планировать исполнение потоков ядра, не обращая внимания на то, что происходит на пользовательском уровне

Что произойдет, если ядро вытеснит поток, заблокировавший какой-либо ресурс?

- Другие потоки не смогут войти в критическую секцию и будут заблокированы (перейдут в состояние ожидания)
  - это компромисс, как и во всех остальных случаях
- Решение этой проблемы требует координации действий ядра и менеджера потоков пользовательского уровня
  - "активация планировщика" (“scheduler activations”)

Выводы

- Использование нескольких потоков в одном адресном пространстве действительно удобно
- Потоки ядра намного более эффективны чем процессы, но они все равно недостаточно дешевы
  - все операции требуют вызовов ядра и проверки параметров
- Потоки пользовательского уровня
  - быстры
  - отлично подходят для операций общего назначения
    - создание, синхронизация, уничтожение
  - в частных случаях испытывают затруднения из-за неосведомленности ядра
    - синхронный ввод-вывод
    - вытеснение потока – владельца заблокированного ресурса

Вопросник по проекту №1

1. Оцените свою работу по реализации шелла (0-5):
   - 0 = не начинал(а)
   - 1-3 = в процессе разработки
   - 4 = все написано, есть баги, отлаживаю
   - 5 = все работает, закончил(а)
2. Оцените свою работу по добавлению системного вызова (0-5)
3. Работает ли у Вас в шелле выполнение команд через fork / execvp?
4. Работает ли у Вас в шелле команда “. *filename*”?
5. Пробовали ли Вы компилировать ядро?
6. Разобрались ли Вы, как добавляется новый системный вызов?
7. Пробовали ли Вы тестировать Ваше ядро в VMware?
8. Сколько часов Вы уже потратили (примерно) на выполнение проекта №1?
9. С Вашей точки зрения, сколько часов Вам еще понадобится, чтобы закончить проект №1?