Реализация потоков. Синхронизация

Операционные системы 2011/12

Татьяна Романова

1 октября 2011 г.

План на сегодня

- Потоки ядра, легковесные процессы, потоки в пространстве пользователя.
- Примеры реализаций многопоточности.
- Необходимость синхронизации.
- Примеры проблем, варианты решений.
- Объекты синхронизации: спин-блокировка.

Материалы

- CS 162, лекции 5, 6 (видео лекции на YouTube).
- Э. Таненбаум, Современные операционные системы, глава 2 (межпроцессное взаимодействие)
- http://en.wikipedia.org/wiki/Thread_(computer_science)
- Anderson et al. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, 1991.
- Drepper, Molnar. The Native POSIX Thread Library for Linux, 2005.

Свойства потоков

Свойственно и процессам:

- Переключение потоков создает видимость одновременного выполнения.
- Параллелятся на несколько процессоров (при определенной реализации).
- Один поток блокируется, другой продолжает работу.
- С каждым потоком связано его текущее сосотояние (заблокирован, готов к вылонению, работает), набор регистров и стек.

В отличие от процессов:

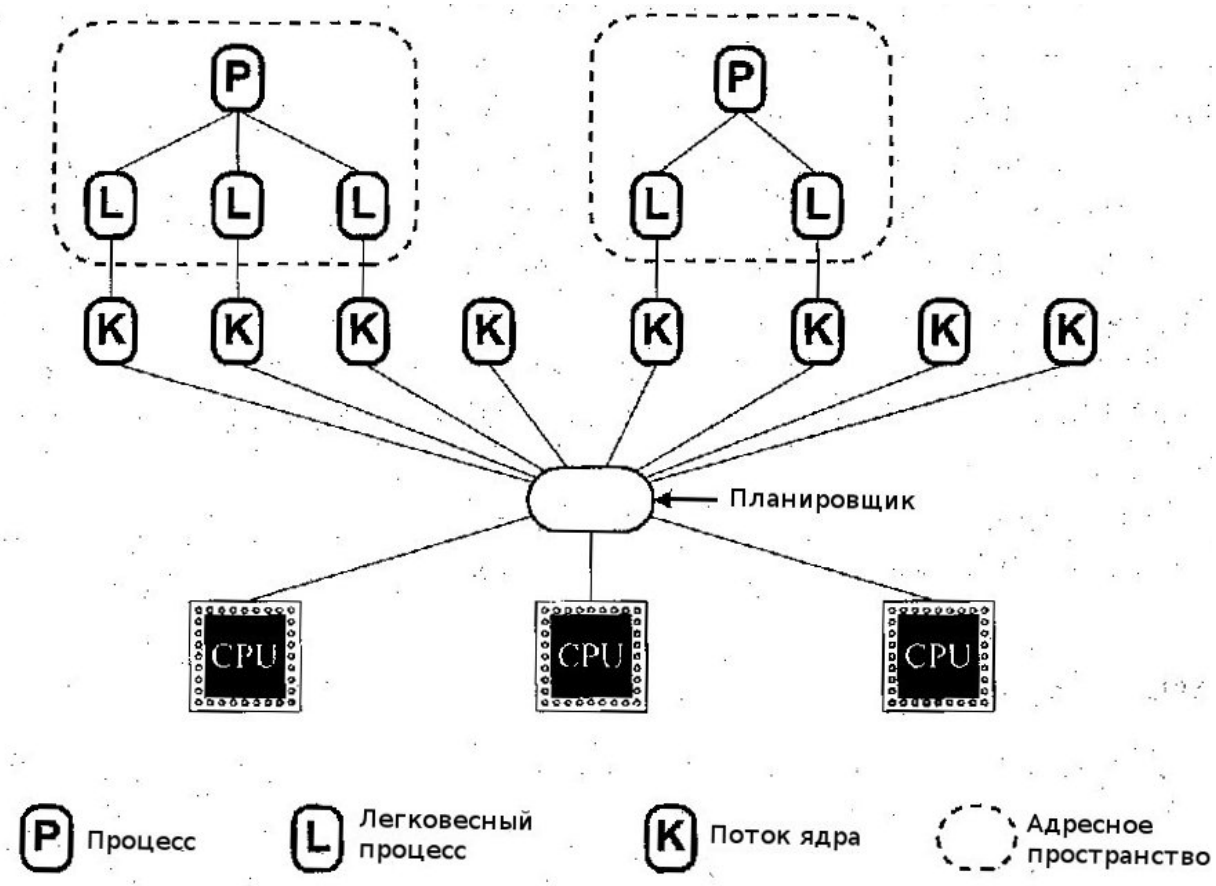
- Более быстрое переключение.
- Тратят меньше ресурсов ядра $\Longrightarrow$ можно создавать больше потоков.
- Разделение ресурсов упрощает взаимодействие и дает возможность делать самые разные ошибки.

Реализации потоков

Потоки ядра

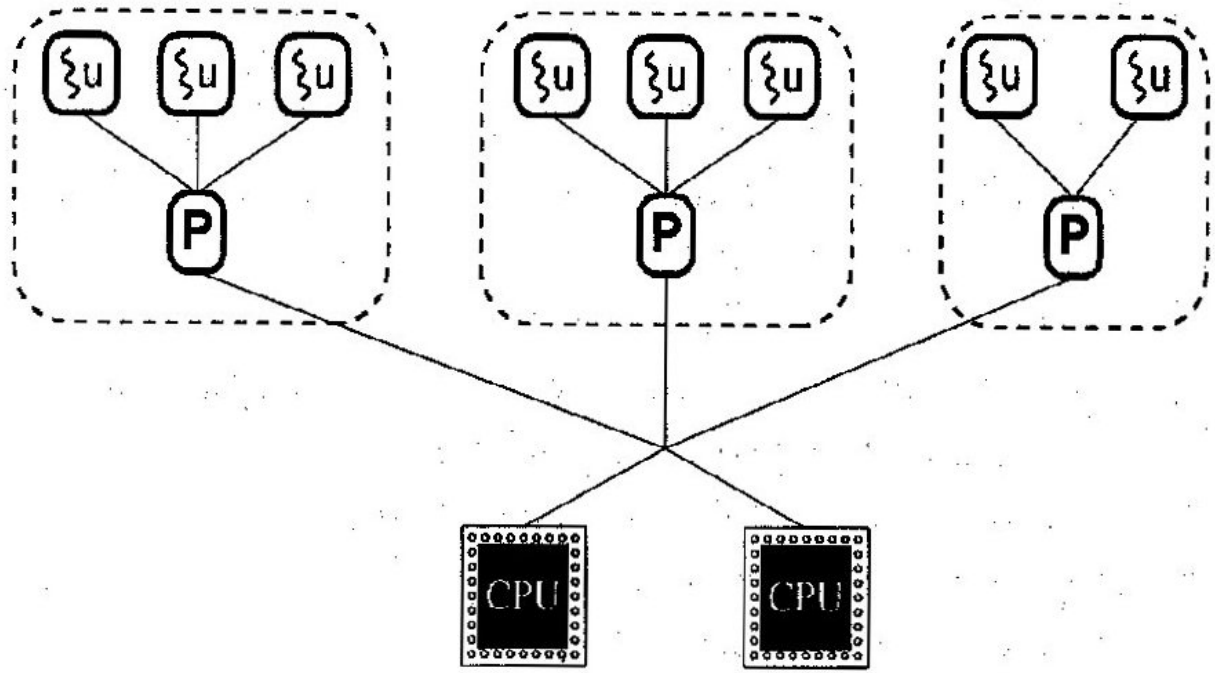
- Не связаны с прикладным процессом.
- Создаются, работают и уничтожаются внутри ядра.
- Используют только стек ядра.
- Быстрое переключение — нет необходимости переходить в режим задачи или переключать адресное пространство.
- Используются для работы системных процессов-демонов.

Легковесные процессы (LWP)



- Реализованы поверх потоков ядра
- Планируются ядром $\Longrightarrow$ могут параллелиться на процессоры.
- Создание, переключение и синхронизация — через системные вызовы.

Потоки в пространстве пользователя



- Библиотека потоков сама размещает их в рамках одного процесса.
- Она же занимается планированием.
- Для ОС это — один процесс.
- Быстро создаются, легко переключаются.
- Могут использовать невытесняющее планирование (yield), что снижает расходы на синхнонизацию.

Модели 1:1, N:1, M:N

1. 1:1, потоки в пространстве ядра.
   На каждый поток выделяется поток ядра. Самая простая реализация. Необходима поддержка со стороны ядра ОС. (Win32, Solaris, NetBSD, FreeBSD, DragonFly BSD, Linux).
2. N:1, потоки в пространстве пользователя.
   На несколько потоков пользователя один поток ядра. Не могут использовать многопроцессорность, могут блокироваться, если не использовать асинхронные операции. (Прикладная библиотека потоков в Plan9, GNU Portable threads)
3. M:N, комбинированный подход.
   Пользовательские потоки мультиплескируются на несколько потоков ядра. Сложная реализация, возможны проблемы с планированием (инверсия приоритетов) (FreeBSD 5.0, NetBSD с 2 по 4, SunOS 5.2-5.8).

Активации планировщика

- 1991 год, Андресон и др., использование преимуществ подходов N:1 и 1:1.
- Ядро отвечает за выделение процессоров
- Прикладная библиотка отвечает за планирование
- При изменении конфигурации (добавление процессора, блокирующий вызов одного из потоков и т. п.) ядро делает обратный вызов (upcall) в прикладную библиотеку и передает туда новую активацию планировщика — сформированный контекст, в котором может быть запущен пользовательский поток.

Пример реализации: Linux

- До ядра версии 2.6 использовалась LinuxTreads, которая не очень соответствовала стандарту POSIX, после Native POSIX Thread Library (nptl).
- Нет отдельной концепции «легковесных процессов» (в Solaris есть).
- Процессы и потоки создаются одним и тем же системным вызовом clone с разными параметрами, ядром управляются одинаково.
- В параметрах может быть указано, какие ресурсы нужно разделять.

Интерфейс pthreads

- Ничего не говорит о том, как должны быть реализованы потоки (в ядре, в простанстве пользователя, комбинировано)
- Это хорошо: можно писать переносимый код.
- И плохо: нет гарантии, что код будет выполняться определенным образом, приходится рассчитывать на худшее.
- Предоставляет вызовы pthread_create, pthread_join, pthread_exit, а также механизмы синхронизации потоков.

Синхронизация

Планировщик против программиста

Однопоточная программа:

- Выходные данные однозначно определяются входными
- Результат не завсист от момента запуска
- Ошибки повторяются, программы легко отлаживаются

Многопоточные приложения:

- Порядок запуска и время работы потоков могут быть любыми.
- Если в коде есть ошибка, планировщик рано или поздно ее надет.
- При наличии ошибки получаем «странное» поведение, не всегда помогающее локализовать ошибку.
- Повторить ошибку будет практически невозможно

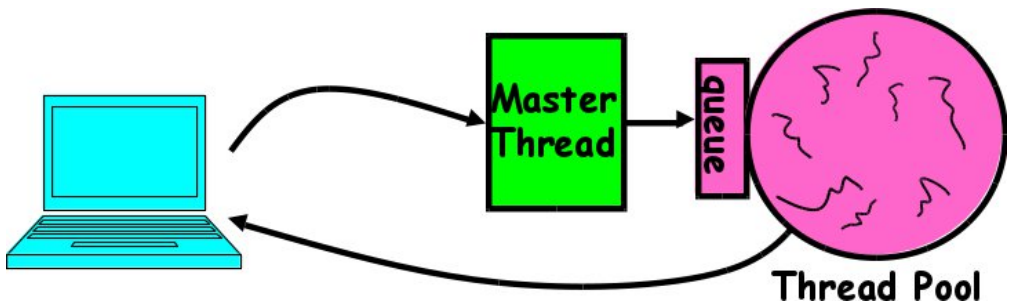
Зачем нужно взаимодействие?

1. Разделение «дорогих» ресурсов
   - много пользователей у одной файловой системы
   - много банкоматов обслуживают один счет в банке
2. Ускорение
   - параллельно: обработка клиентских соединений и чтение ответов с диска
   - работа одного приложения на нескольких процессорах
3. Хорошая программная модель
   - каждый поток отвечает за свою часть работы
   - проще проектировать и расширять

Пример: веб-сервер

Варианты реализации:

- Один поток, последовательная обработка запросов
- На каждый поток по запросу
- Пул (ограниченный набор) потоков



Диспетчер получает запросы и ставит их в очередь. При необходимости «будит» один из рабочих потоков.
Каждый рабочий поток выбирает из непустой очереди запрос и обрабатывает. Если очередь пуста, поток «засыпает».

Пример: банкомат

```
bank_server() {
    while(1) {
        get_request();
        process_request();
    }
}
process_request(r) {
    if (r.type == DEPOSIT) deposit(r.accId, r.amount);
    elif (r.type == WITHDRAW) ...
}
deposit(bal, amount) {
    acc = get_account(accId); // Ввод-вывод
    acc.balance += amount;
    save_account(acc); // Ввод-вывод
}
```

Пример: банкомат

Для ускорения каждый process_request может работать в отдельном потоке.

Два параллельных пополнения одного счета:

Поток 1	Поток 2
load r1, acc.balance	
	load r1, acc.balance
	add r1, amount2
	store r1, acc.balance
add r1, amount1	
store r1, acc.balance	

Новые понятия

Атомарная операция — последовательность инструкций, которые выполняются не прерываясь. Результат: выполнение всех инструкций или невыполнение ни одной.

- Ядро ОС Linux поддерживает атомарные операции с целыми числами и битовые атомарные операции.

Гонка за ресурсами (или состояние состязания) — ситуации, в которых результат работы программы зависит от порядка выполнения потоков.

Критическая область — часть программы, где есть обращение к совместно используемым ресурсам, которое может привести к состязанию.

Взаимное исключение — запрет одновременного нахождения потоков в критической области.

Задачка про «слишком много молока»

Считая атомарными операциями только load и store (загрузка в регистр и сохранение в память), попробуем решить задачу синхронизации:

	Сосед A	Сосед B
15:00	Увидел, что молока нет	
15:10	Ушел в магазин	Увидел, что молока нет
15:20	Купил молоко	Ушел в магазин
15:30	Положил в холодильник	Купил молоко
15:40		Положил в холодильник

Требования:

- Молоко покупает только один
- Кто-то покупает молоко, когда его нет

Решение №1: запрещение прерываний

- Запрещение прерываний — атомарная операция
- Поток A запрещает все прерывания (в т. ч. по таймеру) при входе в критическую область и разрешает снова при выходе.
- Полезно в основном на однопроцессорных машинах.
- Должно быть недоступно в режиме задачи.
- В режиме ядра — возможно, но на очень которкое время.

Решение №1: запрещение прерываний

- Запрещение прерываний — атомарная операция
- Поток A запрещает все прерывания (в т. ч. по таймеру) при входе в критическую область и разрешает снова при выходе.
- Полезно в основном на однопроцессорных машинах.
- Должно быть недоступно в режиме задачи.
- В режиме ядра — возможно, но на очень которкое время.

Аналогия для решения задачи про молоко: замок на холодильник.

Решение №2: переменные блокировки

```
if (no milk) {
      if (no note) {
            leave note
            buy milk
            remove note
      }
}
```

При входе в критическую область поток проверяет значение переменной блокировки. Если оно равно 0, устанавливает его равным 1 и входит в критическую область.

Решение №2: переменные блокировки

```
if (no milk) {
    if (no note) {
        leave note
        buy milk
        remove note
    }
}
```

При входе в критическую область поток проверяет значение переменной блокировки. Если оно равно 0, устанавливает его равным 1 и входит в критическую область.

- Те же проблемы, что и у исходной задачи
- Возможно переключение на другой поток между if (no_note) и leave_note()
- Ошибка есть, но заметить ее сложнее

Решение № 3

Именные записки:

Поток A

```
leave note A
if (no note B) {
    if (no milk) {
        buy milk
    }
}
remove note A
```

Поток B

```
leave note B
if (no note A) {
    if (no milk) {
        buy milk
    }
}
remove note B
```

Решение № 3

Именные записки:

Поток A

```
leave note A
if (no note B) {
    if (no milk) {
        buy milk
    }
}
remove note A
```

Поток B

```
leave note B
if (no note A) {
    if (no milk) {
        buy milk
    }
}
remove note B
```

- Не работает: есть шанс что никто не купит молока
- Еще менее вероятный, но все же возможный сценарий

Решение №4: активное ожидание

Поток A

```
leave note A
while (note B) {
        do nothing;
}
if (no milk) {
        buy milk
}
remove note A
```

Поток B

```
leave note B
if (no note A) {
        if (no milk) {
                buy milk
        }
}
remove note B
```

Решение №4: активное ожидание

Поток A

```
leave note A
while (note B) {
    do nothing;
}
if (no milk) {
    buy milk
}
remove note A
```

Поток B

```
leave note B
if (no note A) {
    if (no milk) {
        buy milk
    }
}
remove note B
```

Работает, но:

- Несимметричная работа потоков
- Неочевидная работа механизма
- Плохо масштабируется
- Поток A тратит процессор, ничего не делая и мешая работать потоку B

Необходимый интерфейс

- Нужны атомарные операции:
  - mutex.lock() заставляет процесс ждать, если блокировка захвачена кем-то другим
  - mutex.unlock() снимает блокировку, будит ожидающие потоки
- Решение задачи про молоко выглядит просто:

```
milk_mutex.lock()
if (no milk)
    buy milk
milk_mutex.unlock()
```

- Как реализовать?

Заключение

Итоги лекции

- Потоки могут быть реализованы в пространстве ядра и в пространстве пользователя, оба подхода имеют приемущества и недостатки.
- Для работы большинства многопоточных приложений требуется синхронизация.
- Найти ошибку в многопоточном коде довольно сложно.
- Для обеспечения синхронизации нужны атомарные операции.
- Наличие только атомарных load и store позволяет синхронизировать 2 потока, но не очень хорошим способом.