Потоки

Операционные системы 2011/12

Татьяна Романова

24 сентября 2011 г.

План на сегодня

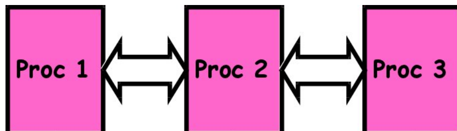
- Чуть-чуть про IPC.
- Потоки.
- Примеры многопоточных приложений.
- Управление потоками.
- Возможные варианты реализации (потоки ядра, легковесные процессы, потоки в пространстве пользователя).
- Примеры реализаций.

Материалы

- CS 162, лекции 3, 4, 5 (видео лекции на YouTube).
- Ю. Вахалия, UNIX изнутри, глава 3.
- Э. Таненбаум, Современные операционные системы, глава 2 (от потоков до межпроцессного взаимодействия)
- 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.

Потоки и процессы

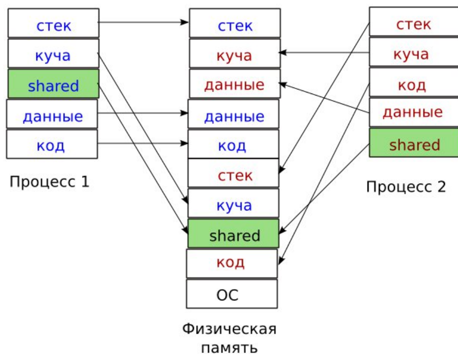
Ограничения процессов



Проблемы:

- Создавать новый процесс дорого.
- Требуется переключение контекста.
- Все операции через системные вызовы.
- У каждого процесса своё адресное пространство.

Межпроцессное взаимодействие



- Сигналы
- Файлы и каналы (pipe)
- Передача сообщений (в т. ч. по сети)
- Разделяемая память, отображение файлов в память (mmap)

Понятие потока

Процесс:

ресурсы

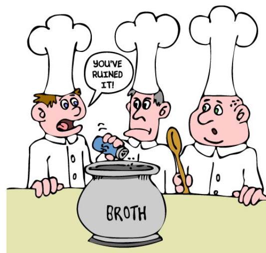
- адресное пространство
- открытые файлы
- информация о привилегиях
- обработчики сигналов

поток выполнения

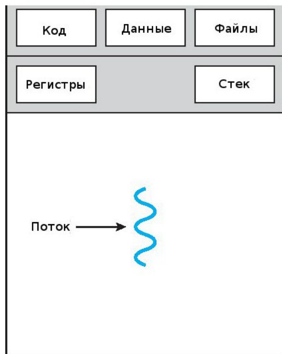
- счетчик команд
- регистры с переменными
- стек
- текущее состояниее

Поток

- минимальная единица планирования
- разделяет с другими потоками данные и код
- реализации могут различаться



Многопоточные программы



- Потоки могут выполняться одновременно.
- Отдельное адресное пространство $\Longrightarrow$ защита других процессов и системы.

Добро или зло?

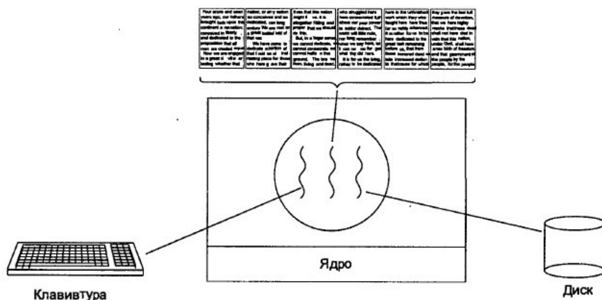
Преимущества:

1. Один поток заблокирован, другой работает (верно и для процессов)
2. Разделяют ресурсы
   - проще огранизовать взаимодействие
   - дешевле создавать и уничтожать
   - дешевле переключаться между потоками (почему?)
3. Работа на многопроцессорных машинах

Недостатки:

1. Оступствие защиты потоков друг от друга
2. Трудности при реализации
   - Какой поток принимает сигналы?
   - Как разместить стек?
   - Не совсем глобальные данные (errno)
3. Необходимость синхронизации

Пример: текстовый редактор



- Один поток взаимодействует с пользователем
- Второй в фоновом режиме форматирует документ
- Третий периодически сохраняет данные на диск

Пример: веб-сервер

- Поток диспетчер принимает запросы и выстраивает их в очередь.
- Если есть свободный рабочий поток — связать его с запросом.
- Рабочий поток: поискать страницу в кэше, если есть, отдать из кэша, если нет, скачать и положить в кэш.

Диспетчер:

```
while(1) {
    get_next_request();
    start_worker();
}
```

Рабочий поток:

```
while(1) {
    wait_for_work();
    look_for_page_in_cache();
    if (not_found())
        wget_and_store();
    return_page();
}
```

Нужна синхронизация потоков при работе с кэшем.
Альтернатива — асинхронный ввод-вывод и модель конечного автомата.

Управление потоками

Ресурсы потока

Общие для всех потоков:

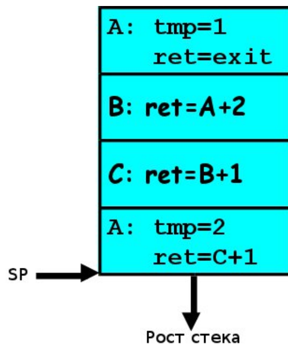
- Данные (глобальные переменные, куча)
- Ввод/вывод (файловая система, сетевые соединения)

У каждого потока свои:

- Блок управления потоком (thread control block, TCB): специальные (PC, SP) и общие регистры процессора, состояние процесса, приоритет, информация планировщика.
- Стек

Пример использования стека

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



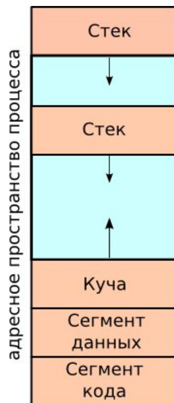
- В стеке хранятся локальные переменные (нельзя возвращать адрес!).
- Сохранение в стеке адреса возврата дает возможность использовать рекурсию.

Стек потока

Программа с двумя потоками:

```
main() {
    create_thread(get_primes("primes.txt"));
    create_thread(hello("hello.txt"));
}
```

- Как расположены стеки потоков?
- Какого размера должен быть стек?
- Может ли один поток испортить стек или TCB другого?
- Можно ли защититься от переполнения стека?



Когда переключаться?

1. Поток сам отдает процессор, переходя в состояние сна или вызывая yield
   - не нужно беспокоиться о синхронизации
   - есть риск никогда не получить управление

   ```
   get_primes() {
       while(1) {
           get_next_prime();
           yield();
       }
   }
   ```

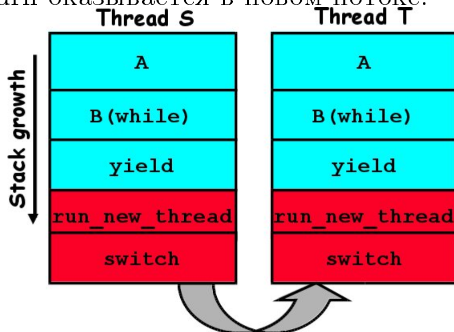
2. Прерывание (например, по таймеру)
   - вектор прерываний (адреса процедур-обработчиков)
   - аппаратно: сохранение PC и состояния в стеке, переход по адресу из вектора прерываний
   - обработчик: завершение сохранения, переустановка SP на свой стек, обработка прерывания
   - восстановление состояния потока или переход к следующему

Как происходит переключение?

```
A() {
    B();
}
B() {
    while(1) {
        yield();
    }
}
```

Запускаем 2 потока: S и T. Каждый начинает выполнять процедуру A().

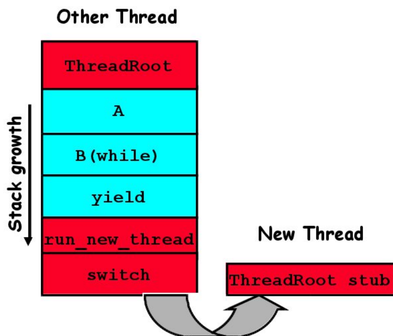
Процедура switch сохраняет регистры старого треда и загружает регистры нового $\Longrightarrow$ с изменением SP и вызовом return оказывается в новом потоке!



Зачем переходить из режима задачи в режим ядра?

Создание

- create_thread(fn_ptr, args_ptr, stack_size);
- Сигнатура функции зависит от используемой библиотеки.
- Проверка аргументов в режиме ядра.
- Выделение TCB, стека и добавление в очередь планировщика, адрес возврата — процедура tread_root();




```
thread_root() {
    do_startup(); //статистика
    user_mode_switch();
    call fn_ptr(args_ptr);
    thread_finish();
}
```

Завершение

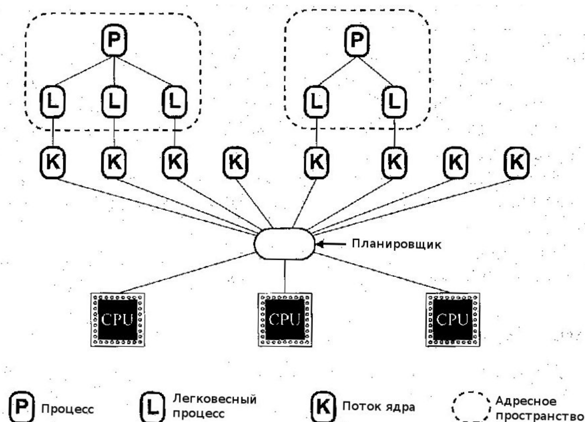
- thread_finish() выполняется в режиме ядра
- thread_join(Thread t) — ожидание завершения потока t (почти как wait)
- При завершении — разбудить всех, кто сделал join текущему потоку
- Не может освободить стек, в котором работает. Но следующий поток — может.

Реализации потоков

Потоки ядра

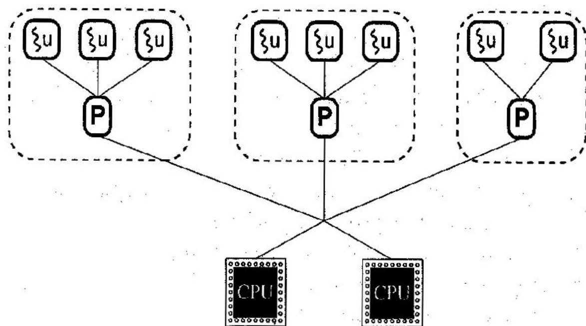
- Не связаны с прикладным процессом.
- Создаются, работают и уничтожаются внутри ядра.
- Используют только стек ядра.
- Быстрое переключение — нет необходимости переходить в режим задачи или переключать адресное пространство.
- Используются для работы системных процессов-демонов.

Легковесные процессы (LWP)



- Реализованы поверх потоков ядра
- Планируются ядром $\Longrightarrow$ могут паралеллиться 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. (Потоки 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 (ntpl).
- 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.

Заключение

Итоги лекции

- Существуют задачи, где необходимо межпроцессное взаимодействие.
- Процесс можно разделить на адресное пространство и потоки выполнения.
- Потоки обладают собственными регистрами и стеком и разделяют все остальное.
- Переключение может быть вытесняющим (по таймеру) и невытесняющим (yield).
- Потоки могут быть реализованы в пространстве ядра и в пространстве пользователя, оба подхода имеют приемущества и недостатки.
- При вытесняющем планировании необходима синхронизация!