

# Многопоточность

Лекция №17 по АКОС  
2019-2020

# Потоки

- Thread (ака поток, нить, легковесный процесс) - единица планирования времени процессора
- У каждого потока свой стек и состояние процессора
- В одном процессе может быть много потоков, - все они разделяют общее адресное пространство

# Главный поток

- При запуске процесса работает один поток, выполняющий функцию `_start`
- Главный поток может запускать другие потоки
- Произвольные потоки могут запускать потоки
- Все потоки равнозначны; в отличие от процессов нет иерархии "родитель-ребенок"

# exit

- Системный вызов `_exit` завершает работу процесса
- Системный вызов `pthread_exit` завершает работу текущего потока
- Можно завершить главный поток, но оставить работать остальные

*Демонстрация: `detached-threads.c`*

# Стек

## Память процесса:

0x7fffffffffff – 8 Mb	<b>Стек</b> ↓
0x03106000	↑ <b>Куча</b>
	<b>.bss</b>
	<b>.data</b>
	<b>.rodata</b>
	<b>.text</b>
0x00400000 0x00000000	<b>Заполнено нулями</b>

## Память потока:

0x7fffffffffff – 8 Mb	<b>Стек</b> ↓
--------------------------	------------------

0x7ffffff7ffffff – 8 Mb	<b>Стек</b> ↓
----------------------------	------------------

0x7ffffeffffff – 8 Mb	<b>Стек</b> ↓
--------------------------	------------------

Области стека могут чередоваться с динамически загружаемыми библиотеками.

# Стек

- По умолчанию размер стека нового потока совпадает с размером стека главного потока
- В POSIX можно явно указывать размер стека и его размещение
- Ниже стека находится Guard Page
- **Никакой изоляции: любой поток может обращаться к памяти другого потока**

# Процесс v.s. поток

- С точки зрения планировщика заданий, процесс и поток - это одно и то же
- Системный вызов `sched_yield` работает с текущим потоком; в случае однопоточной программы - с главным потоком
- Ограничение на "количество процессов" в системе Linux - это ограничение на суммарное количество потоков

# Процесс v.s. поток

## Процессы

- Полная изоляция друг от друга
- Падение одного процесса не влияет на работоспособность остальных
- Процессы могут иметь разные привилегии
- Взаимодействие - только через IPC

## Потоки

- Все находится в одном адресном пространстве
- Сигнал (в том числе вследствие дефектов ПО) убивает все потоки, а не только проблемный
- Все потоки работают под одним пользователем
- Простое взаимодействие



# Атрибуты различных потоков

## Общие

- Process ID
- Parent process ID
- Group ID/Session ID
- Терминал
- UID/GID
- Открытые дескрипторы
- Блокировки файлов
- Обработчики сигналов
- umask, cwd, root dir
- Лимиты

## Различающиеся

- Thread ID
- Стек потока
- Стек обработчика сигналов
- Приоритет выполнения
- Маска сигналов  
pthread\_sigmask
- Значение errno

# <errno.h>

- С точки зрения программиста - "глобальная переменная"
- Реализация - через макрос

```
/* The error code set by various library  
functions.  */
```

```
extern int
```

```
    *__errno_location (void)
```

```
    __THROW __attribute_const__;
```

```
# define errno (*__errno_location ())
```

# Реализации многопоточности

- POSIX (Linux, \*BSD, Mac)  
    <pthread.h>  
    pthread\_create(...)
- WinAPI (Windows)  
    <Windows.h>  
    CreateThread(...)

# WinAPI Threads

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,  
    SIZE_T                    dwStackSize,  
    LPTHREAD_START_ROUTINE    lpStartAddress,  
    LPVOID                    lpParameter,  
    DWORD                     dwCreationFlags,  
    LPDWORD                   lpThreadId  
);
```

Минимальный размер стека – 64Kб

# Стандарт C++11

- Универсальная реализация для всех платформ  
`<thread>`  
`std::thread(...)`
- Не поддерживает тонкую настройку параметров потока, например размер стека

# Стандарт C11

- Стандарт определяет интерфейс `<threads.h>`

```
void call_once(  
    once_flag *flag,  
    void (*func)(void)  
);
```

- **Поддержка потоков C11 не реализована ни в одном компиляторе**

# Запуск процессов

- Все потоки равноценны → можно вызвать `fork()` из любого потока
- При создании процесса создается копия всей памяти, включая стеки всех потоков
- В новом процессе будет существовать **только один поток**, который продолжает выполнение потока, вызвавшего `fork()`

# Запуск процессов

- Могут остаться заблокированные мьютексы/семафоры → deadlock
- Если предполагается использование `fork`, то необходимо зарегистрировать с помощью `pthread_atfork` функции, которые принудительно освободят блокировки



# Выполнение `exec`

- Вызов `exec` полностью заменяет адресное пространство процесса
- После вызова `exec` остается только один поток
- Комбинация `fork+exec` относительно безопасна

# Маски сигналов

(у каждого thread'a - своя)

Какие сигналы будут обработаны →  
Маска [заблокированных] сигналов →

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

**Process 3**  
`kill(SIGFPE, PID)`

0	0	0	0	0	1	0
1	0	1	1	1	1	1
SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
0	1	0	0	0	1	0

**Маски сигналов,**  
в отличие от маски ожидающих доставки,  
**наследуются при fork**

# Обработка сигналов

- Маска сигналов, ожидающих доставки - это свойство **процесса**, а не отдельного потока
- Обработчик сигнала находится в общем адресном пространстве для всех потоков
- Выполнение обработчика сигнала - в **произвольном** потоке
- Для отдельных потоков можно блокировать доставку сигналов (pthread\_sigmask)

# Принудительное завершение

- Антипаттерн проектирования. Следует избегать его использования
- Два способа принудительного завершения потоков:
  - умеренной жестокости: поток может быть остановлен в cancelation point - большинство системных вызовов
  - садомазохистский: поток прибивается принудительно, даже если там `while (1) {}`. Но это происходит не мгновенно, а когда планировщик заданий выберет отмененный поток.

# Принудительное завершение

- Функция `pthread_cancel` "завершает" работу указанного потока: отправляет запрос на остановку
- Поток может запретить обработку таких запросов:  
`pthread_setcancelstate(  
                    PTHREAD_CANCEL_DISABLE)`

# Принудительное завершение

- Два способа завершения:
  - в точке останова (см. список Cancellation Points в `man 7 pthreads`)
  - через посылку сигнала
- Способ завершения определяется `pthread_setcanceltype`

# Принудительное завершение

- При завершении могут остаться:
  - заблокированные семафоры и мьютексы
  - незакрытые файловые дескрипторы
  - потерянные указатели на память в куче
  - просто недоделанная работа
- Либо запрещаем остановку глобально, либо регистрируем "аварийную" функцию очистки для каждого ресурса:  
`pthread_cleanup_push`

# Thread-Local Storage

- Глобальные переменные доступны всем потокам
- Глобальные переменные - зло, но иногда бывают необходимы
- Ключевые слова `_Thread_local` (C11) и `thread_local` (C++) объявляют переменные уникальными для каждого потока
- Значения TLS-переменных не изолированы от других потоков



# Глобальные переменные

- Доступ к глобальным переменным из разных потоком приводит к data race
- Значения глобальных переменных могут меняться косвенно - через вызов функций стандартной библиотеке
- Многие функции помечены как Thread-Safe: они реализуют блокировки (ценой потери производительности)
- Пример: `getc` v.s. `getc_unlocked`

# Как бороться с Data Race

- Атомарные переменные - выровнены по границе машинного слова

`<stdatomic.h>` (C11)

```
typedef _Atomic _Bool atomic_bool;  
typedef _Atomic char atomic_char;  
typedef _Atomic signed char atomic_schar;  
typedef _Atomic unsigned char atomic_uchar;  
typedef _Atomic short atomic_short;  
typedef _Atomic unsigned short atomic_ushort;  
typedef _Atomic int atomic_int;
```

Необходимо использовать специальные атомарные операции

- Принудительные барьеры: мьютексы и семафоры

# Проблемы проектирования

- Накоплена большая кодовая база для однопоточного выполнения
- Самый простой способ использовать такой код - ставить блокировки везде
- Этот способ - самый неэффективный

# Пример: Python

- Запуск потока - это `pthread_create` + инициализация структуры `PyThreadState`
  - Всё честно: каждый поток имеет свой стек и выполняется как отдельная задача
- НО
- Используется разделяемый экземпляр `PyInterpreterState`

**Global Interpreter Lock (GIL)**

# Пример: Python. Как бороться

- Не использовать Python
- Для распараллеливания использовать модуль `multiprocessing` вместо `multithreading`
- Параллельную часть вынести в Си-код, который не трогает `PyInterpreterState`

```
Py_BEGIN_ALLOW_THREADS    // освобождение GIL  
..... // какой-то Си-код, не использующий Py API  
Py_END_ALLOW_THREADS      // захват GIL
```

# Потоки внутри ядра

- Цель - распараллелить задачи внутри ядра между разными ядрами процессора
- Выполняются в адресном пространстве ядра
- В Linux всем потокам внутри ядра назначаются фиктивные ProcessID, которые процессами не являются
- Примеры:
  - kswapd - управляет разделом подкачки
  - kworker - [как правило] операции ввода-вывода

# Giant Kernel Lock

- Глобальная блокировка, которая приводит к тому, что выполняется только один поток
- Процессы могут блокировать работу других процессов даже в многоядерных системах, вызывая системные вызовы
- Проблема - в Legасу-коде отдельных драйверов или подсистем
- В BSD проблема была окончательно устранена в 2009 году (FreeBSD 8.0)
- В Linux проблема была окончательно устранена в 2011 году (2.6.39)

# BeOS / HaikuOS

<https://www.haiku-os.org/>

- Система, изначально спроектированная для использования на SMP
- Каждая подсистема ядра работает в отдельном потоке

