

Основы системного программирования

Потоки и синхронизация

Гирик Алексей Валерьевич

Университет ИТМО
2021

Материалы курса

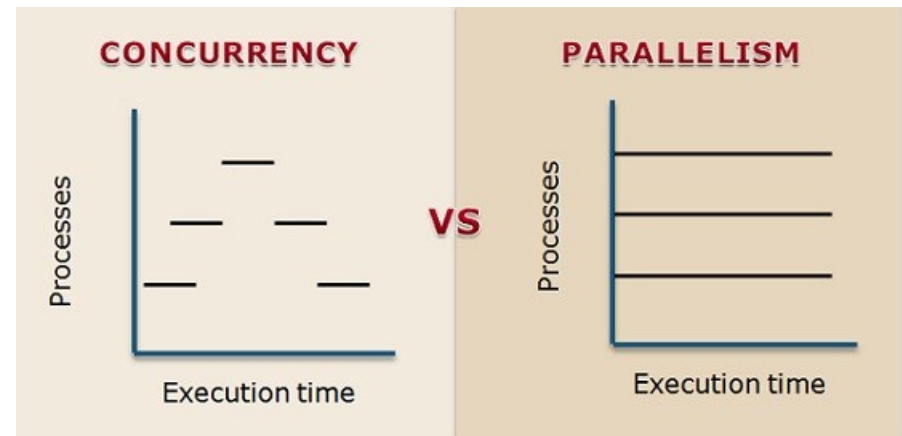
- Презентации, материалы к лекциям, литература, задания на лабораторные работы
 - shorturl.at/gjABL



Многопоточное программирование с помощью библиотеки pthread

Термины

- Параллелизм (parallelism)
- Конкурентность (concurrency)
- Многозадачность (multitasking)
- Многопроцессность (multiprocessing)
- **Многопоточность** (multithreading)
- Асинхронная обработка (asynchronous processing)



Группы процессов, сессии, задания

- Linux – многопользовательская многозадачная ОС общего назначения
 - задача (*task*)
 - программа (*program*)
 - процесс (*process*)
 - **поток (*thread*)**
 - задание (*job*)
 - сессия (*session*)
 - группы
 - процессов (*process groups*)
 - **потоков (*thread groups*)**

Потоки

- Потоки можно рассматривать как минимум с двух точек зрения:
 - поток – объект диспетчеризации, средство занять все ядра процессора и обеспечить реальный параллелизм

$$A = 1 / (S + (1 - S)/N),$$

где A – коэффициент ускорения от параллельной обработки,

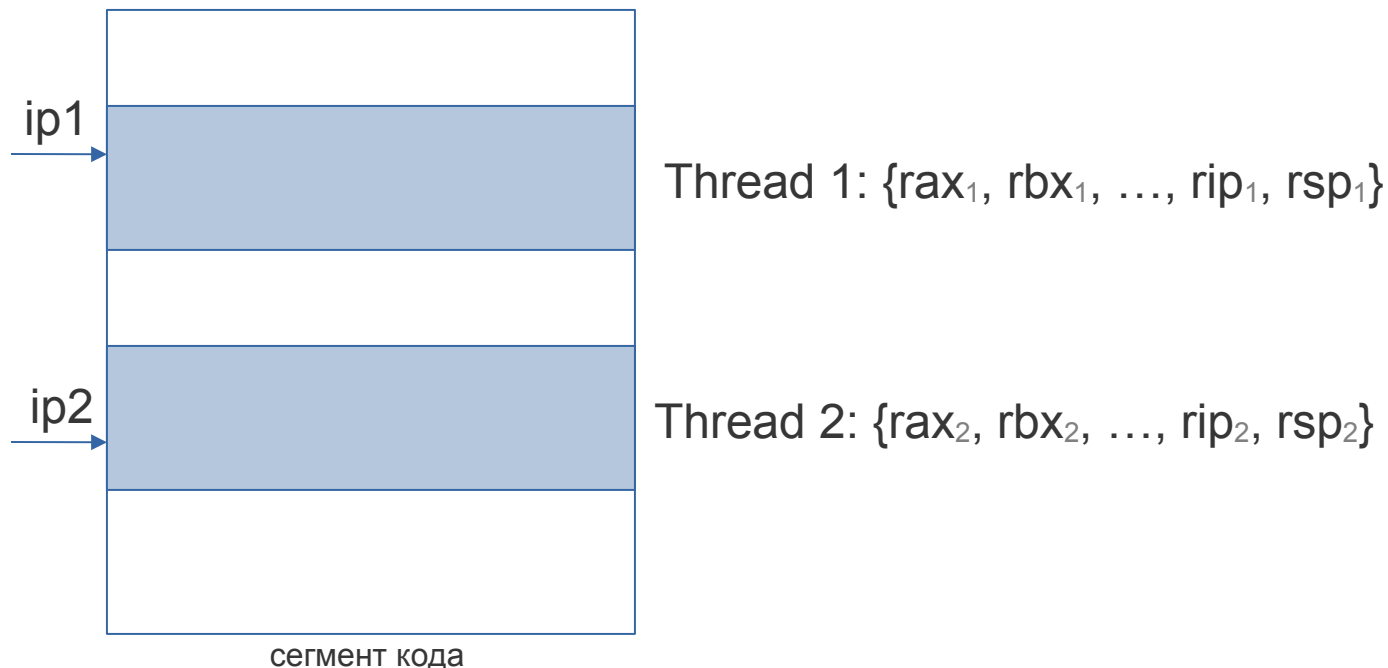
N – количество потоков,

S – доля последовательно выполняемого кода.

- поток – средство логического разделения кода и средство изоляции частей программы

Что такое поток

- Если говорить упрощенно, поток – это состояние регистров процессора (адрес в сегменте кода + остальные регистры)
 - поток – “виртуальный процессор”



Сравнение процессов и потоков

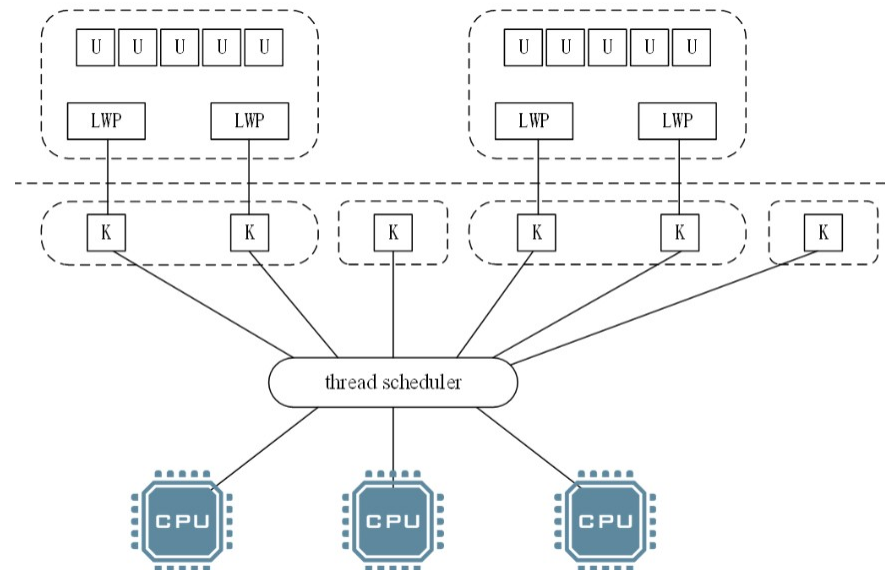
- время, необходимое для создания
- время переключения контекста
 - каталог страниц
 - TLB
 - в некоторых процессорах (ARM) TLB содержит также PID
- общность ресурсов
 - самый главный ресурс – память (адресное пространство)
- изоляция
 - обмен данными
 - действия при возникновении ошибки/исключения

Выводы из сравнения процессов и потоков

- процесс – “контейнер для ресурсов”
 - потоки в каком-то смысле тоже “ресурсы”
- в каждом процессе есть как минимум один поток
 - поток – это поток **выполнения**, “виртуальный процессор”
 - все потоки процесса работают в одном и том же адресном пространстве
- поток – более легковесная сущность, чем процесс

Процессы и потоки

- В настоящее время в Linux потоки пользовательского режима переключаются в режим ядра и обратно с помощью системных вызовов
- Могут быть потоки, работающие только в режиме ядра



Реализации потоков в Linux

- LinuxThreads
 - до версии ядра 2.6
- NPTL (Native POSIX Thread Library)
 - начиная с 2.6
 - и потоки, и процессы в Linux создаются с помощью системного вызова `clone()`, но с разными флагами
 - строго говоря, возможно создание потоков без использования библиотеки `pthread` (используя `clone()`), однако это как минимум неудобно

Создание потоков

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

Завершение потока

- Поток завершается, если:
 - потоковая функция делает return
 - поток вызывает pthread_exit() или exit()
 - поток отменяется вызовом pthread_cancel()
 - завершается основной поток, вызывая exit() или возвращаясь из main()
 - если завершение происходит с помощью pthread_exit(), то дочерние потоки продолжают работу
- Если поток вызывает exit(), _exit() или _Exit(), то завершается весь процесс!

Присоединение потока

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread,  
                 void **retval); [i]
```

Если не присоединять завершённые потоки, в программе будут утечки памяти!

Отсоединение потока

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t thread);
```

Присоединение потока с ожиданием

```
#define _GNU_SOURCE
#include <pthread.h>
```

```
int pthread_tryjoin_np(pthread_t thread,
    void **retval);
```

```
int pthread_timedjoin_np(pthread_t thread,
    void **retval,
    const struct timespec *abstime);
```


Отмена потока

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

```
void pthread_testcancel(void);
```

Потоки и сигналы

```
#include <signal.h>

int pthread_sigmask(int how,
                    const sigset_t *set,
                    sigset_t *oldset);
```

Отправка сигнала потоку

```
#include <signal.h>
```

```
int pthread_kill(pthread_t thread,  
    int sig);
```

Функции очистки завершения потока

```
#include <pthread.h>
```

```
void pthread_cleanup_push(  
    void (*routine)(void *),  
    void *arg);
```

```
void pthread_cleanup_pop(int execute);
```

Синхронизация потоков

Синхронизация потоков

- асинхронный доступ к данным
 - чтение неизменяемых данных не вызывает проблем
 - одновременный доступ на чтение и запись может приводить к нарушению инвариантов
- потокобезопасность (thread safety)
 - повторная входимость функций (reentrancy)

Средства синхронизации потоков

- средства синхронизации в библиотеке pthread
 - циклические блокировки (spinlocks)
 - мьютексы (mutexes)
 - условные переменные (condition variables)
 - блокировки чтения-записи (read-write locks)
 - барьеры (barriers)

Циклическая блокировка

```
#include <pthread.h>

int pthread_spin_init(
    pthread_spinlock_t *lock,
    int pshared);

int pthread_spin_destroy(
    pthread_spinlock_t *lock);
```


Циклическая блокировка

```
#include <pthread.h>

int pthread_spin_lock(
    pthread_spinlock_t *lock);

int pthread_spin_trylock(
    pthread_spinlock_t *lock);

int pthread_spin_unlock(
    pthread_spinlock_t *lock);
```

Мьютексы

```
#include <pthread.h>
```

```
pthread_mutex_t mutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_init(  
    pthread_mutex_t *mutex,  
    pthread_mutexattr_t *attr);
```

```
int pthread_mutex_destroy(  
    pthread_mutex_t *mutex);
```

Захват и освобождение мьютекса

```
#include <pthread.h>
```

```
int pthread_mutex_lock(  
    pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(  
    pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(  
    pthread_mutex_t *mutex);
```

Иерархии блокировок и тупики

- при наличии в программе нескольких мьютексов возможны тупики (deadlocks)
 - иерархия блокировок должна быть выстроена таким образом, чтобы тупики исключались
 - альтернативный способ – откат (backoff)

Гранулярность блокировок

- гранулярность блокировок (locking granularity) определяется длиной критических секций кода (длиной участков кода, защищенных примитивами синхронизации)
 - желательно стремиться к тому, чтобы уменьшить гранулярность блокировок
 - lock chaining – подход, при котором критические секции перекрываются и при входе во вложенную критическую секцию примитив синхронизации охватывающей секции освобождается

Условные переменные

```
#include <pthread.h>
```

```
pthread_cond_t cond =  
    PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(  
    pthread_cond_t *cond,  
    pthread_condattr_t *condattr);
```

```
int pthread_cond_destroy(  
    pthread_cond_t *cond);
```

Ожидание условной переменной

```
#include <pthread.h>
```

```
int pthread_cond_wait(  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex,  
    struct timespec *expiration);
```

Блокировки чтения-записи

```
#include <pthread.h>
```

```
pthread_rwlock_t rwlock =  
    PTHREAD_RWLOCK_INITIALIZER;
```

```
int pthread_rwlock_init(  
    pthread_rwlock_t *rwlock,  
    const pthread_rwlockattr_t *attr);
```

```
int pthread_rwlock_destroy(  
    pthread_rwlock_t *rwlock);
```


Блокировки чтения-записи

```
#include <pthread.h>
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Выводы по процессам и потокам

- в каких случаях использовать процессы, а в каких – потоки?
- как уменьшить накладные расходы на создание потоков/процессов?

Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
 - shorturl.at/gjABL



Задание на дом

- читать
 - Стивенс, Раго. UNIX. Профессиональное программирование
 - главы 11, 12
 - Лав. Linux: Системное программирование
 - глава 7

