Системное программирование

Лекция 6

POSIX Threads

Многозадачность

Многозадачность – способность ОС управлять выполнением нескольких задач одновременно.

Кооперативная многозадачность – способ реализации многозадачности, при котором моменты передачи управления от одной задачи к другой определяет сама задача.

Вытесняющая многозадачность – способ реализации многозадачности, при котором момент передачи управления от одной задачи к другой определяет среда выполнения.

Процессы и потоки

Процесс – экземпляр выполняющейся программы.

Процесс соответствует программе в целом. Данные программы также являются частью процесса. За исполнение программы отвечают потоки выполнения процесса.

Поток выполнения * – участок программного кода, который может быть выбран планировщиком для выполнения.

У каждого процесса есть как минимум 1 поток выполнения, который в момент запуска начинает выполнять код программы с начала.

* execution thread, *встречается перевод «нить»*

Реализация потоков

Существует несколько вариантов реализации потоков, различающихся преимущественно по уровню, на котором происходит управление потоками:

- **Реализация на уровне ядра** – потоки имеют отражение в ядре, управлением занимается планировщик ОС.
- **Реализация на уровне пользователя** – ядро занимается только управлением процессов, управлением потоков занимается модуль в пользовательском пространстве.

Некоторые языки программирования комбинируют оба этих способа реализации (goroutine в Go, Task/ThreadPool в C#)

POSIX Threads

Вследствие роста популярности многозадачных систем стандарт POSIX был расширен. Основные функции и структуры для работы с потоками были включены в *POSIX.1c, Threads extensions*. Сам стандарт POSIX Threads, равно как и его реализации, сокращенно зовутся **Pthreads**.

В Pthreads каждый поток имеет собственные:

- идентификатор потока [можно получить функцией `pthread_t pthread_self()`];
- стек*;
- маску сигналов;
- Thread Local Storage*;
- приоритет.

Функции Pthreads возвращают код ошибки напрямую – не через errno.

Замечание: использование Pthreads требует связывания с библиотекой libpthread.so и настройки дополнительных директив препроцессора. Сделать это можно через флаг компилятора *-pthread* или директив CMake:

```
set(THREADS_PREFER_PTHREAD_FLAG ON)
find_package(Threads REQUIRED)
link_libraries(Threads::Threads).
```

Создание потока

Потоки создаются функцией `pthread_create()`:

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

Параметры:

`thread`	–	буфер для идентификатора потока;
`attr`	–	атрибуты создаваемого потока [опционален];
`start_routine`	–	функция потока;
`arg`	–	аргумент для функции потока.

Поток запускается сразу после создания.

Завершение потока

Поток завершается, если:

- завершилось выполнение функции потока;
- вызвана фунцкия `void pthread_exit(void* retval);`
- вызвана функция `pthread_cancel()` и для потока разрешено внешнее завершение работы.

Поток считается завершившимся успешно только в первых 2 случаях, поскольку у него будет возвращаемое значение.

В качестве результата работы потока используется возвращаемое значение функции потока или значение параметра `retval` функции `pthread_exit()`.

Замечание: необработанное исключение в любом из потоков приведет к завершению всего процесса.

Ожидание завершения потока

Дождаться завершения конкретного потока можно функцией `pthread_join()`.

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

Параметры:

`thread` – идентификатор ожидаемого потока;
`retval` – буфер для результата работы потока [опционален].

Функция блокирует вызывающий поток до тех пор, пока целевой поток не завершится.

Если известно, что никто не будет ожидать завершения потока, поток может быть отсоединен функцией `pthread_detach()`:

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

Результат работы потока (пример)

```
void* thread_function(void* arg);
void pthread_exit(void* retval);
int pthread_join(pthread_t thread, void** retval);
```

Значение `void*`, возвращаемое функцией потока, является результатом работы потока, который может быть получен через `pthread_join()`.

Результат работы потока может также быть установлен функцией `pthread_exit(value)`.

Тип `void*` выбран как наиболее общий, поскольку значение можно затем привести к требуемому типу.

Запрещается возвращать указатель на локальные переменные (поскольку стек потока будет уничтожен). Если возвращать нечего – верните NULL.

Внешнее завершение потока (пример)

Запрос на досрочное завершение потока посылается функцией pthread_cancel():

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

Результатом работы завершенного потока является константа PTHREAD_CANCELED.

При этом поток может определить свою реакцию на запрос завершения функциями

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Параметры:

state – разрешение на прием запросов (PTHREAD_CANCEL_ENABLE/DISABLE),
type – способ обработки (PTHREAD_CANCEL_DEFERRED/ASYNCHRONOUS).

Если type==PTHREAD_CANCEL_DEFERRED, то проверка на наличие запроса производится только при вызове функций, помеченных как *cancellation points* – например `void pthread_testcancel()`.

Выполнение действий при завершении (#1)

Для потоков нет аналога функции `atexit()`. Вместо этого используется пара макросов `pthread_cleanup_push/ pthread_cleanup_pop()`:

```
void pthread_cleanup_push(void (*func)(void*), void* arg);
void pthread_cleanup_pop(int execute);
```

`pthread_cleanup_push()` регистрирует функцию-обработчик, которая будет вызвана с заданным аргументом, если внутри кода между push/pop будет вызвана `pthread_exit()` или придет запрос на завершение;

`pthread_cleanup_pop()` дерегистрирует функцию-обработчик и, опционально, выполняет эту функцию, если `execute != 0`.

Thread Local Storage

Thread Local Storage (хранилище локальных данных потока) – структура, в которой хранятся данные, специфичные для потока.

TLS устроен в виде набора пар «ключ-значение».

Доступ к TLS осуществляется в 2 этапа:

1. На первом этапе создается ключ TLS.
2. На втором этапе с ключом ассоциируется указатель либо целочисленное значение, связанное с данными.

Примером переменной в TLS является **errno** – она своя у каждого потока.

Доступ к TLS

Ключи TLS создаются и уничтожаются парой функций

```
int pthread_key_create(pthread_key_t* key,
                       void  (*destr_function) (void*));
int pthread_key_delete(pthread_key_t key);
```

Функция `pthread_key_create()` создаёт новый ключ TLS. В аргументе `destr_function` передается функция-деструктор, которая будет вызвана при завершении потока, если с ключом ассоциировано не-NULL значение.

Функция `pthread_key_delete()` уничтожает ключ без вызова деструктора.

Доступ к TLS

С ключом TLS может быть связано некоторое значение, которое может затем быть получено внутри потока в любой точке. Данное значение по умолчанию равно NULL.

```
int   pthread_setspecific(pthread_key_t key,
                          const void* pointer);
void* pthread_getspecific(pthread_key_t key);
```

Функция `pthread_setspecific()` назначает ключу `key` значение `pointer`.

Функция `pthread_getspecific()` позволяет получить значение по ключу.

Выполнение действий при завершении (#2)

TLS может быть использовано для выполнения заданных действий по завершении работы потока. Для этого необходимо:

1. Создать ключ вызовом функции `pthread_key_create()` с необходимой функцией-обработчиком в качестве параметра `destr_function`.
2. Ассоциировать с ключом ненулевое значение вызовом функции `pthread_setspecific()`. Данное значение будет использовано как аргумент `destr_function`.

Заданная функция вызовется в момент завершения потока.
Если функцию обработчик нужно отменить – достаточно удалить ключ через `pthread_key_delete()`;

Ключевое слово thread_local (C/C++) (пример)

Многие языки программирования позволяют неявно использовать TLS.

В C и C++ переменная может быть помещена в TLS с помощью спецификатора **thread_local**.

```
int x = 0;
thread_local int y = 0;

int foo() {
    x = rand(); //изменение увидят все потоки
    y = rand(); //изменение увидит только текущий поток
}
```

Если переменная является объектом с нетривиальным деструктором, то деструктор будет вызван по завершении потока.

Потоки и сигналы

- Обработчики сигнала могут быть установлены только для всего процесса – нельзя иметь разные обработчики в разных потоках.
- Каждый поток имеет свою маску сигналов, которая устанавливается функцией `pthread_sigmask()`.
- Посылка сигналов между потоками одного процесса выполняется функциями `pthread_kill()/pthread_sigqueue()`.
- Сигнал, посланный процессу, обрабатывается потоком, в котором данный сигнал не заблокирован. Если таких потоков несколько, то поток выбирается случайно. Если сигнал заблокирован во всех потоках, то он заблокирован для всего процесса.

Примечание: результат вызова `sigprocmask()` в многопоточной программе зависит от конкретной ОС, поэтому его не следует использовать

Маска сигналов потока

Нельзя назначить одному и тому же сигналу разные обработчики в разных потоках, но можно указать, на какие сигналы поток будет реагировать, путем установки маски сигналов.

Маска сигналов потока по умолчанию наследуется от потока-создателя.

Маска сигналов отдельного потока может быть изменена функцией `pthread_sigmask()`:

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

Аргументы и поведение функции аналогичны вызову `sigprocmask()`.

Посылка сигналов потоку (пример)

Обмен сигналами между потоками внутри одного процесса производится функциями `pthread_kill/pthread_sigqueue()`.

```
int pthread_kill(pthread_t thread, int sig);
int pthread_sigqueue(pthread_t thread, int sig,
                                  const union sigval value);
```

Данные функции аналогичны вызовам `kill/sigqueue()`, за исключением первого аргумента.

Замечание: в рамках POSIX нет возможности послать сигнал конкретному потоку другого процесса.

Получение сигнала потоком(пример)

Поток может :

- получить заблокированный сигнал вызовом `sigwaitinfo()` или его вариациями;
- обработать сигнал и (опционально) синхронизироваться с помощью `sigsuspend()`.

При этом нужно учитывать, что поток не различает сигналы, полученные от других процессов и сигналы, полученные от потоков этого же процесса.

Если процесс получает сигнал, который разблокирован в нескольких потоках, то он обрабатывается только в одном из них (в каком – неизвестно).

Если несколько потоков ожидают заблокированный сигнал с помощью `sigwaitinfo()`, и сигнал приходит всему процессу, то результат зависит от ОС. На Linux в одном из потоков вызов завершится успешно и вернет номер сигнала, в остальных потоках вызов завершится с ошибкой EINTR.

Потоки и fork

- Если `fork()` был вызван в многопоточной программе, то будет скопирован только вызывающий поток.
- После `fork()` данный поток должен выполнить вызов `exec*()`. В промежутке между `fork()` и `exec*()` поток может вызывать только реентерабельные функции и функции, явно помеченные, как безопасные.
- Для обеспечения корректного состояния ресурсов, занятых потоком, после `fork()` можно зарегистрировать функции-обработчики, выполняющие необходимые действия функцией `pthread_atfork()`:

```
int pthread_atfork(void (*prepare)(void),void (*parent)(void),
                   void (*child)(void));
```

Потоки и exec

В момент вызова exec*()

- все потоки старого процесса, за исключением вызывающего, уничтожаются;
- вызывающий поток становится главным потоком новой программы и начинает исполнять main().

Приоритет потока (Linux only) (пример)

Приоритет потока – значение, определяющее, насколько часто планировщик выбирает поток для продолжения выполнения.

Приоритет в POSIX представляется в виде числа в диапазоне от [-19, 20], где 20 – *наименьший* приоритет.

Нижняя граница приоритета задается пределом RLIMIT_NICE.

Приоритет потока задается системным вызовом `nice()`.

```
int nice(int inc);
```

Параметр `inc` задает величину изменения приоритета.

Вызов возвращает новое значение приоритета или -1. Т.к. -1 может оказаться новым значением приоритета, необходимо проверять errno.

Прикрепление потока к ядру (Linux only) (пример)

В некоторых случаях необходимо прикрепить поток к одному или нескольким ядрам (задать CPU affinity). Прикрепление наследуется при создании потока.

```
int pthread_setaffinity_np(pthread_t thread,
                  size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread,
                        size_t cpusetsize, cpu_set_t *cpuset);
```

Параметры:

`thread` - идентификатор потока;
`cpuset` - набор ядер, к которым прикреплен поток;
`cpusetsize` - общей размер набора ядер [для обычных ПК = `sizeof(cpu_set_t)`].

Для работы с `cpu_set_t` используются макросы `CPU_ZERO()`, `CPU_SET()`, `CPU_CLR()` (*man CPU_SET*).

Уступка процессорного времени

Попросить планировщик ОС передать управление другому потоку можно вызовом `sched_yield()`.

```
int sched_yield();
```

Отличие от `sleep()/nanosleep()` в том, что минимальное время приостановки потока не регламентируется (планировщик может вовсе немедленно перезапустить поток).

Данный системный вызов следует использовать, если становится очевидно, что в данный момент потоку «нечего делать», но потом работа для него появится.