Lab 1

Как создать поток?

Традиционная модель процессов в UNIX поддерживает только один поток управления на процесс. Концептуально это то же, что и модель, основанная на потоках, в случае, когда каждый процесс состоит из одного потока.

Структуры процесса

Виртуальная память процесса содержит следующие структуры:

- текст
- данные
- пользовательский стек
- куча
- динамические сегменты

Текст содержит исполняемый код программы нашего процесса. *Данные* имеют два сегмент - *DATA* (неинициализированные статические и внешние переменные) и *BSS* (неинициализированные статические данные и внешние переменные). *Пользовательский стек* - стек автоматической памяти программы, на котором хранятся локальные переменные, аргументы вызова ф-ций, их возвращаемые значения и другая информация. *Куча* - участок динамической памяти. *Динамические сегменты* - сегменты кода и данных разделяемых библиотек, отображенные на память файлы и др.

Также нужно упомянуть связанную с процессом структуру, которая храниться в памяти ядра - *пользовательская область* процесса или *user area*. В данной структуре храниться информация о процессе, необходимая для его исполнения - *атрибуты процесса*; например, дескрипторы открытых файлов, информация о

системных ресурсах, реакции на сигналы, параметры командной строки и пр.

Также, пользовательская область хранит стек, которым ядро пользуется при выполнении системных вызовов. Если в процессе присутствует, несколько нитей, то на каждую нить в пользовательской области выделяется отдельный стек, чтобы системные вызовы в каждой нити могут выполняться независимо.

- поток - единица планирования в рамках одного процесса.

Реализация многопоточности

Стандарт POSIX допускает различные подходы к реализации многопоточности. Рассмотрим три основных:

- Нити в пределах одного процесса (в пользовательском адресном пр-ве) переключаются собственным *пользовательским планировщиком*.
- Переключение между нитями осуществляется *системным планировщиком*, так же, как и при переключении между процессами.
- Гибридная реализация, при которой каждому процессу выделяется некоторое кол-во системных нитей, при этом процесс также имеет свой собственный пользовательский планировщик в пользовательском адресном пр-ве.

Примечание: в данном контексте *пользовательской нитью* является нить, планируемая *пользовательским планировщиком*, а *системной нитью* - нить, планируемая системным планировщиком соответственно. В большинстве UNIX-системах в т.ч. в Solaris системные нити называются *LWP - Light Weight Process*.

использование пользовательского планировщика

Достоинством данного варианта является то, что он может быть реализован без изменений ядра системы. Когда системный планировщик передает

планирование процессора на какой-то процесс, пользовательский планировщик этого процесса может решить, какой из пользовательских нитей передать управление.

Однако, при практическом применении такого планировщика возникает серьёзная проблема. Если одна из нитей процесса выполнит блокирующийся системный вызов, то тогда блокируется весь процесс. Решение данной проблемы потребует произвести серьезные изменения в ядре операционной системы, что в итоге сводит на нет главное достоинство пользовательского планировщика.

использование системного планировщика

Ядро типичной ОС уже имеет системный планировщик, который переключает управление между процессами. Переделать этот планировщик для того, чтобы он также мог переключать несколько нитей одного процесса относительно нетрудно. Возможны два способа это сделать:

1. Нить является подчиненной, по отношению к процессу сущностью. Идентификатор нити состоит из идентификатора процесса и собственного идентификатора нити, уникального в рамках данного процесса. (А вчём подход то??) Большинство UNIX-систем использует именно такой подход, в т.ч. и Solaris 10.
2. Нить является сущностью того же уровня, что и процесс. ?

Поскольку ОС с данным подходом реализации многопоточности (в частности Solaris 10 и Linux) используют именно системные нити, то каждому потоку POSIX Thread API соответствует отдельный LWP.

При наличии поддержки pthread программа также запускается как процесс, состоящий из одного потока управления. Поведение такой программы ничем не отличается от поведения традиционного процесса, пока она не создаст дополнительные потоки управления. Создание дополнительных потоков производится с помощью функции pthread_create.

```
SYNOPSIS
       #include <pthread.h>

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

       Compile and link with -pthread.
```

```
ERRORS
       EAGAIN Insufficient resources to create another thread.

       EAGAIN A system-imposed limit on the number of threads was encountered.
              There  are  a  number of limits that may trigger this error: the
              RLIMIT_NPROC soft resource limit (set via  setrlimit(2)),  which
              limits  the  number of processes and threads for a real user ID,
              was reached; the kernel's system-wide limit  on  the  number  of
              processes and threads, /proc/sys/kernel/threads-max, was reached
              (see proc(5)); or the maximum  number  of  PIDs,  /proc/sys/ker-
              nel/pid_max, was reached (see proc(5)).

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters speci-
              fied in attr.
```

Вновь созданный поток начинает выполнение с функции start_routine . Эта функция принимает единственный аргумент arg — нетипизированный указатель. Если функции start_routine() потребуется передать значительный объем информации, ее следует сохранить в виде структуры и передать в arg указатель на структуру. При создании нового потока нельзя заранее предположить, кто первым получит управление — вновь созданный поток или поток, вызвавший функцию pthread_create . Новый поток имеет доступ к адресному пространству процесса и наследует от вызывающего потока среду окружения арифметического сопроцессора и маску сигналов, однако набор сигналов, ожидающих обработки, для нового потока очищается.

Завершение процесса системным вызовом `exit(2)` или возвратом из функции `main` приводит к завершению всех нитей процесса. Это поведение описано в стандарте POSIX, поэтому ОС, которые ведут себя иначе (например, старые версии Linux), не соответствуют стандарту. Если вы хотите, чтобы нити вашей программы продолжали исполнение после завершения `main`, следует завершать `main` при помощи вызова `pthread_exit(3C)`.

Если атрибут потока *detachstate* имеет значение PTHREAD_CREATE_JOINABLE, статус выхода, указанный параметром *value_ptr* становится доступным при вызове *pthread_join()* к текущему потоку.

Все обработчики прерывания после вызова *pthread_exit()* изымаются из стека и исполняются в порядке *FILO*. (Про обработчики прерывания нити будет написано позже.)

Неявный вызов *pthread_exit()* происходит, когда в поток, отличный от *main*, возвращается из ф-ции, с которой он создавался, посредством оператора *return*. В данном случае, возвращаемое значение ф-ции можно считать статусом выхода в *pthread_exit()*.

атрибуты нити: При создании нити можно указать ей блок атрибутов при помощи параметра *attr*. Данный блок представляет из себя структуру *pthread_attr_t*. Ниже приведён список с описанием основных атрибутов нити. Кроме этих атрибутов структура *pthread_attr_t* содержит некоторые другие (например *cancellation state* и *cancelation type*, которые будут рассмотрены позже).

Для каждого атрибута из блока *pthread_attr_t* определены функции установки и взятия значения вида (get/set) - *pthread_attr_set$AttributeName$* и *pthread_attr_get$AttributeName$*. Изменение атрибутов в структуре *pthread_attr_t* не повлияет на атрибуты нитей, которые были уже созданы с ней. Таким образом, одну и ту же структуру *pthread_attr_t* можно использовать для инициализации нескольких нитей.

Некоторые атрибуты самой нити могут быть модифицированы после ее создания, например detachstate и priority.

- scope (область действия) - допустимые значения: *PTHREAD_SCOPE_SYSTEM* и *PTHREAD_SCOPE_PROCESS*.
    - *PTHREAD_SCOPE_SYSTEM* - нить планируется системным планировщиком и соревнуется за системные ресурсы с другими процессами.
    - *PTHREAD_SCOPE_PROCESS* - нить планируется пользовательским планировщиком и считается частью своего процесса (по умолчанию для Solaris).
- Solaris и Linux являются ОС, в которых многопоточность имеет реализацию через *системный планировщик* и каждый поток соответствует собственному *LWP*, поэтому данный атрибут не имеет практического значения. В Linux данный атрибут может быть равен только *PTHREAD_SCOPE_SYSTEM*, в Solaris 10 таких ограничений нет, но атрибут всё-равно бесполезен. В таком случае, правильнее было бы указать *PTHREAD_SCOPE_SYSTEM* как значение по умолчанию для Solaris, но имеем, что имеем.

- detachstate (присоединен/отсоединен) - Допустимые значения: *PTHREAD_CREATE_JOINABLE* и *PTHREAD_CREATE_DETACHED*.
    - *PTHREAD_CREATE_DETACHED* - ресурсы нити будут освобождены сразу после её завершения.
    - *PTHREAD_CREATE_JOINABLE* - ресурсы нити будут освобождены после того, как другая нить вызовет *pthread_join()* к ней.

- stacksize (размер стека) - допустимые значения: 0 или размер стека в байтах. 0 означает, что размер стека определяется операционной системой. В Solaris 10 на 32-битной архитектуре стек выделяется размером 1mb, на 64-битной - 2mb. Минимальный размер стека, который можно выделить нити, определён константой *PTHREAD_STACK_MIN* в <pthread.h>.

- stack_addr (адрес начала стека) - Допустимые значения: NULL(адрес определяется ОС) или указатель на область памяти для размещения стека нити. Не рекомендуется выделять память под стек самостоятельно,

поскольку, в таком случае, система не будет освобождать эту память автоматически, это должен реализовать программист ?.
- Concurrency (степень параллелизма) - допустимые значения: положительные целые числа. В системах с гибридной реализацией планирования нитей, данной значение с определенным приближением соответствует кол-ву системных нитей, создаваемых системой для текущего процесса.

В Solaris 10 данный атрибут игнорируется и нужен только для совместимости со стандартом POSIX.

- Schedpolicy (политика планирования) - допустимые значения: Допустимые значения – SCHED_FIFO, SCHED_RR и SCHED_OTHER.
    - SCHED_FIFO - планирование нитей осуществляется в порядке линейной очереди. Также, при данной политике планирования, после получения управления нить отдаст его только, когда заблокируется на примитиве синхронизации.
    - RR - планирование нитей осуществляется в порядке кольцевой очереди. Нить, получившая управление отдаёт его по истечении кванта времени, либо до момента явной отдачи управления. При отдаче управления нить помещается в конец очереди.
    - SCHED_OTHER - данное значение соответствует системной политики планирования.

атрибут	значение по умолчанию	описание
scope	PTHREAD_SCOPE_PROCESS	Нить планируется пользовательским планировщиком. В Solaris 9 и последующих версиях Solaris этот параметр не имеет практического значения
detachstate	PTHREAD_CREATE_JOINABLE	нить создаётся присоединённой
stackaddr	NULL	адрес начала стека нити определяется системой
stacksize	0	размер стека нити определяется системой
priority	0	нить имеет приоритет 0
inheritsched	PTHREAD_EXPLICIT_SCHED	Нить не наследует приоритет у родительской нити
schedpolicy	SCHED_OTHER	Нить использует фиксированные приоритеты, задаваемые ОС. Используется вытесняющая многозадачность (нить исполняется, пока не будет вытеснена другой нитью или не заблокируется на примитиве синхронизации)