Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self()

На лекциях упоминалось, что во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счётчик, своё содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их как элементы разделяемой памяти, не прибегая к механизму, описанному выше.

В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Вы кратко ознакомитесь с некоторыми функциями, позволяющими разделить процесс на thread'ы и управлять их поведением, в соответствии со стандартом POSIX. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор thread'а. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция `pthread_self()`. Нить исполнения, создаваемую при рождении нового процесса, принято называть *начальной* или *главной нитью исполнения* этого процесса.

Создание и завершение thread'а. Функции pthread_create(), pthread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом

```
void *thread(void *arg);
```

Параметр `arg` передается этой функции при создании thread'а и может, до некоторой степени, рассматриваться как аналог параметров функции `main()`, о которых мы говорили на семинарах 3–4. Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребёнка. Для создания новой нити исполнения применяется функция `pthread_create()`.

Мы не будете рассматривать её в полном объёме, так как детальное изучение программирования с использованием thread'ов не является целью данного курса.

Важным отличием этой функции от большинства других системных вызовов и функций является то, что в случае неудачного завершения она **возвращает не отрицательное, а положительное значение**, которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной `errno` при этом не устанавливается. Результатом выполнения этой функции является появление в системе новой нити исполнения, которая будет выполнять функцию, ассоциированную со thread'ом, передав ей специфицированный параметр, параллельно с уже существовавшими нитями исполнения процесса.

Созданный thread может завершить свою деятельность тремя способами:

- с помощью выполнения функции `pthread_exit()`. Функция никогда не возвращается в вызвавшую её нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'a, например, на статическую переменную;
- с помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'a;
- если в процессе выполняется возврат из функции `main()` или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции `exit()`, это приводит к завершению всех thread'ов процесса.

Стоит заметить, что третий вариант не является безусловно верным, так как существует возможность отсоединить нить исполнения, которая не рассматривается в рамках данного курса.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции `pthread_join()`. Нить исполнения, вызвавшая эту функцию, переходит в состояние **ожидание** до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в операционную систему.

Использование pthread_join() позволяет гарантировать, что главная нить не закончит свою работу раньше всех остальных и они успеют отработать до конца.

Для иллюстрации вышесказанного рассмотрим программу, в которой работают две нити исполнения - файл **07-2.c**. Каждая нить исполнения просто увеличивает на 1 разделяемую переменную a.

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается

автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра **–lpthread** – подключить библиотеку pthread. Откомпилируйте эту программу и запустите на исполнение.

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрирующей создание нового процесса (***Пример 1****: 03-1.c*), которую мы рассматривали на семинарах 3 – 4. Программа, создавшая новый процесс, печатала дважды одинаковые значения для переменной a, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной a. Рассматриваемая программа печатает два разных значения, так как переменная a является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.

Необходимость синхронизации процессов и нитей исполнения, использующих общую память

Все рассмотренные на этом семинаре примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения.

Вернёмся к рассмотрению программ **07-1a.c** и **07-1b.c**. При одновременном существовании двух процессов в операционной системе может возникнуть следующая последовательность выполнения операций во времени:

```
...
Процесс 1: array[0] += 1;
Процесс 2: array[1] += 1;
Процесс 1: array[2] += 1;
Процесс 1: printf("Program 1 was spawn %d times, program 2 - %d times, total
- %d times\n", array[0], array[1], array[2]);
...
```

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Мы не сможем подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но мы можем смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором array[2] + = 1; Это проделано в следующих программах: файлы **07-3a.c** и **07-3b.c**.

Откомпилируйте их и запустите любую из них один раз для создания и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите снова первую программу. Вы получите неожиданный результат:

количество запусков по отдельности не будет соответствовать количеству запусков вместе.

Как вы видите, для написания корректно работающих программ необходимо обеспечивать взаимоисключение при работе с разделяемой памятью и, может быть, взаимную очередность доступа к ней. Это можно сделать с помощью алгоритмов синхронизации, например, алгоритма Петерсона или алгоритма булочной.

Использование неизмененных алгоритмов Петерсона и булочной, вообще говоря, даст полностью корректно работающие программы только на одноядерных машинах, где используется модель строгой непротиворечивости памяти. В нашем случае из-за наличия большого пустого цикла кэши и оперативная память успевают синхронизироваться, и для программ 07-3a.c и 07-3b.c всё будет хорошо с алгоритмом Петерсона и на многоядерных системах. Наличие race condition наблюдается и в программе 07-2.c, где при определённых условиях обе нити исполнения могут выдать значение a равное 1 из-за неатомарности выполнения оператора a = a + 1.

Задачи на семинар

***Задача 1 (5 баллов)**:*

*Модифицируйте программу **07-2.c**, добавив к ней третью нить исполнения.*

***Задача 2 (10 баллов)**:*

*Исправьте программы **07-3a.c** и **07-3b.c**, используя алгоритм Петерсона так, чтобы они работали корректно.*