УДК 004.04

НЕ ТОЛЬКО ПОТОКИ

Дагаев Дмитрий Викторович, Директор по АСУТП, ОАО «ДЖЭТ», Россия, Москва, dvdagaev@mail.ru

Задачи и сопрограммы

При разработке серверных приложений часто встает вопрос необходимости одновременной работы и взаимодействия нескольких подзадач. Это сродни многозадачности, в [1] говорится о "разновидности процесса внутри самого процесса". Как решения для серверной задачи предлагаются 3 варианта:

- многопоточный процесс;
- однопоточный процесс;
- параллельная работа с неблокирующими системными вызовами.

Там, где однопоточный процесс с блокирующими системными вызовами невозможен, обычно используется многопоточное программирование. Мы же будем рассматривать последний случай, который предполагает несколько нитей вычислений с сохраняемым состоянием у каждого, и события, по которым осуществляются переходы состояний. В качестве реализации предлагаются:

- задачи (Tasks), в которых нужно делать функцию перехода состояний в стиле конечного автомата, и возвращающие управление;
- сопрограммы (Coroutines), позволяющие выходить из произвольной точки кода процедуры и возвращаться на следующий за ней оператор.

Для каждого объекта-задачи вводится состояние и функция перехода, которая для простого примера печати натуральных чисел выглядит так:

```
PROCEDURE Do (t: Task);
BEGIN IF t.count < t.max THEN INC(t.count); Log.Int(t.count); Log.Ln; END
END Do;
```

Функция должна быть написана таким образом, чтобы при периодическом вызове изменять состояние `t.count` и возвращать управление.

Вариант с объектом-сопрограммой более универсален:

```
PROCEDURE Do (t: Cotoutine); VAR j: INTEGER;
BEGIN j := 0; WHILE TRUE DO INC(j); Log.Int(j); Log.Ln; Co.Yield END
END Do;
```

Функция содержит только точку выхода `Co.Yield` и возвращается при следующем вызове на следующий за ней оператор.

Использование сопрограмм имеет долгую историю. По утверждению Кнута [2], "подпрограмма является частным случаем сопрограммы". Сопрограммы широко использовались в ассемблерных проектах, Вирт ввел сопрограммы в язык Модула-2. Работа подпрограммы определяется более строгой дисциплиной LIFO, при которой выделяется стековая область для локальных переменных каждого экземпляра подпрограммы (детально: записывает в стек адрес следующей команды, переходит по адресу процедуры, выделяет в стеке место для локальных переменных). При выходе из подпрограммы эта область освобождается (помещая в стек возвращаемое значение и переходя к следующей команде).

В сопрограммах такой дисциплины нет, для их использования нужны 3 свойства [3]:

- локальные данные сохраняются между успешными вызовами;
- выполнение приостанавливается в определенных точках разрыва, при дальнейшей передаче управление переходит на следующий оператор;
- управление может передаваться от одной сопрограммы к другой, при этом вызывающая сопрограмма приостанавливается. Последнее свойство запрещено использовать для решения задач разделения времени, чтобы обеспечить герметичность взаимодействия.

Оператор `Yield` осуществляет как раз этот выход из процедуры с последующим возвратом на следующий за ним оператор. Это достигается путем отдельного сохранения всего стека выполнения каждой сопрограммы в предварительно отведенной памяти, а также состояния регистров (указателя стека, счетчика команд, и др.). При вызове `Yield` происходит

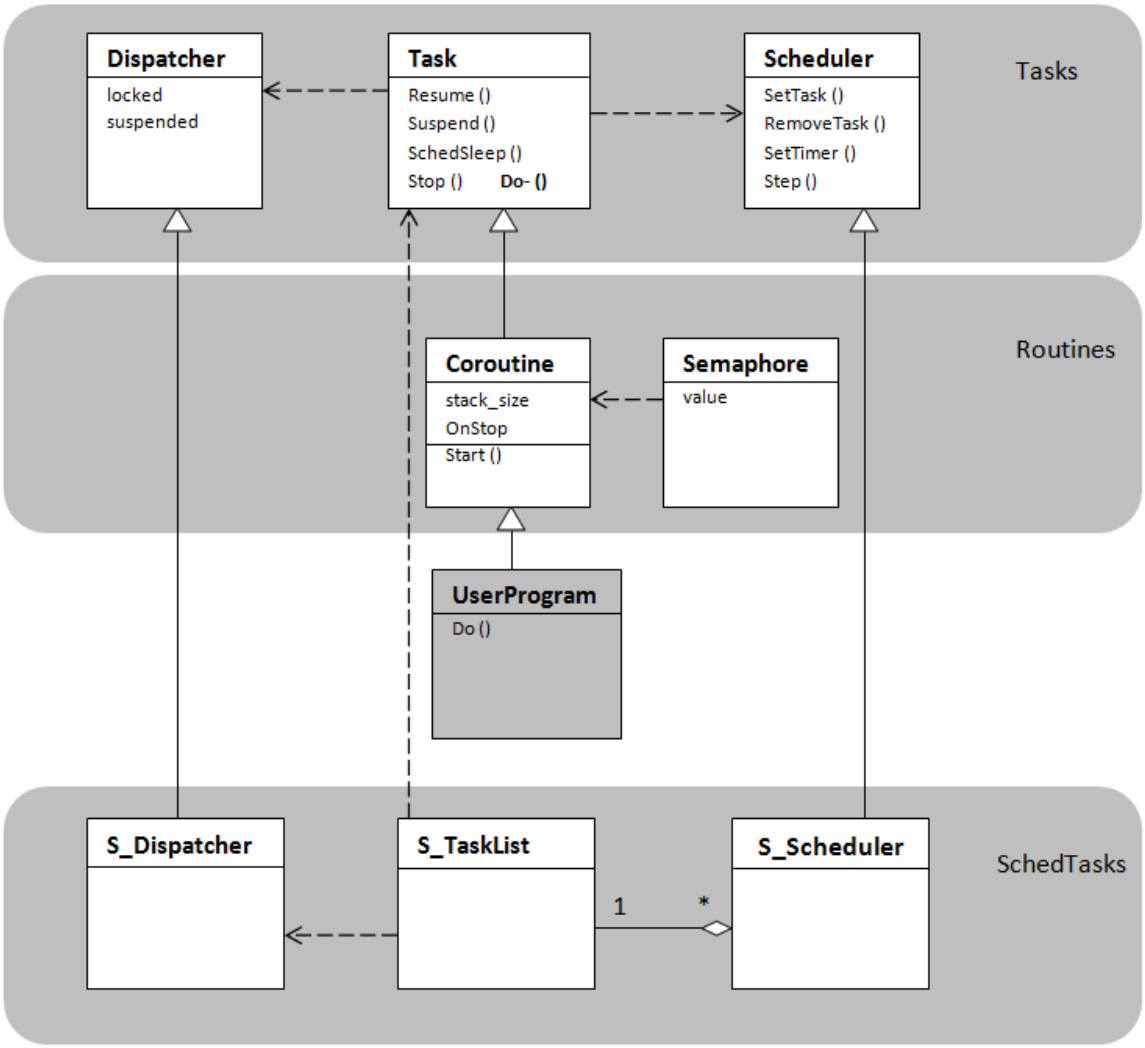
такой переход с переключением контекстов, но сопрограммы считаются легковеснее потоков, т.к. последние, как минимум, имеют дополнительные издержки на штатный планировщик.

Для управления задачами и сопрограммами потребовалось реализовать планировщик на уровне пользовательского процесса, который и осуществляет механизмы переключения.

Объектная модель планировщика

Разработанная модель берет свое начало в [4], при этом пользовательская задача наследуется от Task и реализует состояния и функцию перехода `Do`, либо пользовательская

сопрограмма наследуется от Coroutine и реализует вызываемую планировщиком функцию `Do` с точками разрыва. В соответствие с заложенными принципами, пользовательский тип наследуется от типа, определенного в Tasks, и реализуется виртуальная функция `Do`. Т.е. есть наследование типов и нет наследования реализации.



Построение основано на тройке классов Scheduler/Task/Dispatcher, предложенной в [4] как вариация обобщенной тройки Carrier/Rider/Link (другой вариацией является известная Модель/Вид/Контроллер). При этом иерархии классов Task-Coroutine-UserProgram (возможно, и Task-UserProgram) и Dispatcher-S_Dispatcher+Scheduler-S_Scheduler развиваются независимо.

Рис. 1 – Объектная модель планировщика

Первая группа отвечает за пользовательскую реализацию. В соответствии с принципом отделения абстракции от реализации пользовательская программа наследуется от типа Coroutine и остается одинаковой, несмотря на разные реализации сопрограммы на 2 компиляторах для 2 ОС каждый. Помимо реализации метода `Do` нужно также иметь в виду, что переходы на другие сопрограммы в режиме планировщика запрещены на уровне рантайма, а управление поведением сопрограмм (скажем, для защиты общих ресурсов) осуществляется с помощью семафоров и методов Resume, Suspend, SchedSleep.

Вторая группа S_Scheduler+S_Dispatcher представляет собой реализацию абстрактных типов Scheduler и Dispatcher, которая работает со списком задач S_TaskList, выбирая адрес, куда передается управление. Реализуется алгоритм циклического планирования с двумя

уровнями приоритета. В зависимости от параметров планировщика, определяющих квоту времени, переход в точках разрыва может не осуществляться, если квота не выбрана.

17

Следует сравнить механизмы семафоров сопрограмм, с одной стороны, и считающиеся более структурированными мониторы Хоара-Хансена [1] и активные объекты, с другой [5]. Отсутствие вытесняющей многозадачности меняет поведение семафоров, т.к. вся сопрограмма превращается в «критическую область» в смысле Хансена или в эксклюзивную секцию активного объекта. Семафор определяет точку разрыва защищенной области, в которой сопрограмма будет находиться в состоянии активного ожидания:

```
BEGIN Do1; Co.SemDown(sem); Do2 END
```

Аналогичная секция активного объекта на языке Активный Оберон [5] будет выглядеть:

```
BEGIN {EXCLUSIVE} Do1; AWAIT(sem>0); DEC(sem); Do2 END
```

Следует заметить, что активные объекты и мониторы разрабатывались на замену менее структурированным средствам синхронизации, в т.ч. семафорам. Но для сопрограмм эта неструктурированность уходит, причем это решается без дополнительных языковых средств.

Сравнение реализаций

Для языков Оберон-семейства BlackBox и XDS и операционных систем Linux и Windows сравнивались варианты реализации.

XDS подключает одинаковый для Linux и Windows библиотечный модуль COROUTINES, основные функции которого – старт и переключение:

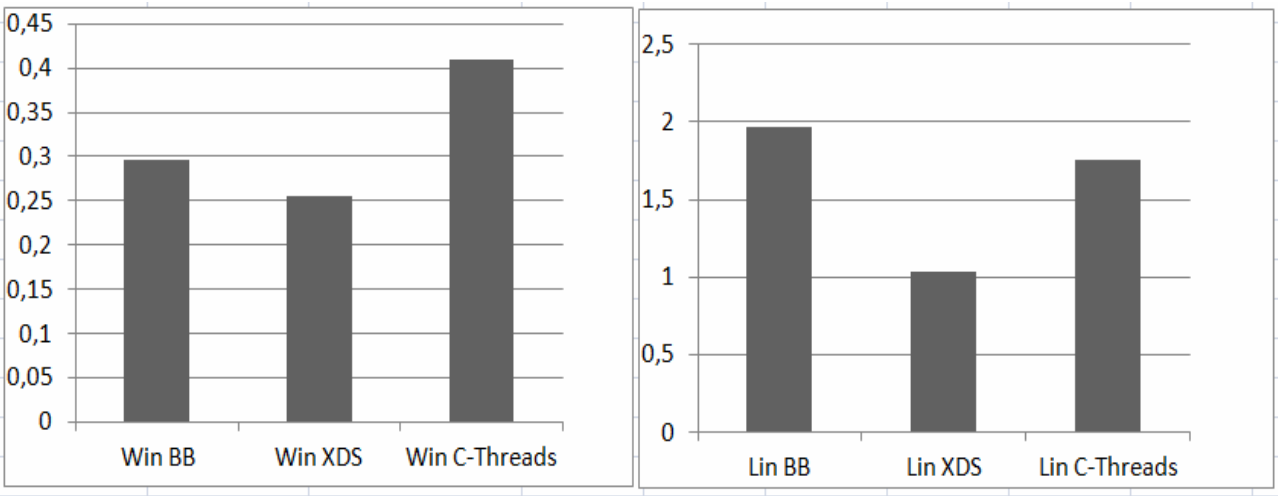
```
COROUTINES.NEWCOROUTINE(RunProc, stack, ssize, coroutine); (* старт *)
COROUTINES.TRANSFER(from, coroutine); (* переключение *)
```

BlackBox использует функции работы с Fibers библиотеки win32:

```
primary := WinApi.ConvertThreadToFiber(0);
fiber := WinApi.CreateFiber(ssize, RunProc, coroutine); (* старт *)
WinApi.SwitchToFiber(fiber); (* переключение *)
```

BlackBox для Linux использует для аналогичных целей контексты, подключая модуль CA как библиотеку libc CA[“libc.so.6”]:

```
context.uc_stack.ss_sp := stack;
context.uc_stack.ss_size := ssize;
rc := CA.makecontext(context, RunProc, 3, context
      , tmp_context, coroutine); (* старт *)
rc := CA.swapcontext(tmp_context, context) (* переключение *)
```



Были оценены времена переключений между сопрограммами по сравнению с временами переключения между потоками в реализации на C в тестовом примере с числом задач = 100.

Рис. 2 – Среднее время переключения между задачами, мкс

18

Приведенные диаграммы носят в большей степени качественный характер, т.к. реальные характеристики сильно зависят от оборудования, на котором производится

тестирование. В нормальном случае переключение между заведомо более «тяжеловесными» потоками должно занимать большее время, поэтому дополнительные 40-70% времени для потоков представляются разумной цифрой. Только времена по переключению контекстов BlackBox для Linux отличаются в худшую сторону, в связи с чем можно посетовать на реализацию библиотеки контекстов в Linux или перейти на более низкий, ассемблерный уровень.

В качестве рекомендаций по применению следует иметь в виду несколько моментов. Эти данные соответствуют однопроцессорному аппаратному обеспечению, многопоточные приложения, в отличие от сопрограмм, распределяются по аппаратуре. Также, несмотря на полную защищенность Оберонов, их сборщики мусора не отслеживают локальные стеки сопрограмм, поэтому указатели как переменные в локальных стеках не должны оставаться последними адресатами памяти, что-то должно быть в «куче» или основной программе. Зато такой подход дает более гибкие способы управления планированием.

Литература

1. Э.Таненбаум. Современные Операционные Системы. 3-е изд. 2012.
2. Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — M., 2006
3. George Frasier “Cross-Platform Coroutines in C++”, Dr.Jobbs, March 01, 2001
4. C.Szyperski "Insight-EthOS: An Object-Orientation in Operating Systems", 1992.
5. P.J. Muller. The Active Object System – Design and Multiprocessor Implementation. PhD thesis, ETH Zurich, 200