Модели работы с памятью в учебном языке программирования СИНХРО

Л.В. Городняя[1, 2]

[1] *Институт систем информатики им. А.П. Ершова СО РАН*
[2] *Новосибирский государственный университет*

Аннотация. Статья посвящена ряду решений по организации работы с данными в учебном языке СИНХРО, нацеленном на ознакомление с феноменами параллелизма и обучение подготовке многопоточных программ над общей памятью. В центре внимания модели работы в условиях наследования идей функционального программирования при необходимости обеспечения производительности многопроцессорных программ.

Ключевые слова: обучение программированию, виртуальная машина, система команд, функциональное программирование, восстановление данных, освобождение памяти

Memory processing in the programming language for educational purposes SYNHRO

L.V.Gorodnyaya[1,1,]

[1] *A.P. Ershov Institute of Informatics Systems*
[2] *Novosibirsk State University*

Abstract. The article is devoted to a number of solutions for organizing work with data in the SYNHRO programming language, aimed at teaching the preparation of multi-threaded programs over shared memory. The focus of attention is on the model of work in terms of inheriting the ideas of functional programming when it is necessary to ensure the performance of multiprocessor programs.

Keywords: programming training, virtual machine, instruction set, functional programming, data recovery, memory freeing

В лаборатории информационных систем ИСИ СО РАН разрабатывается учебный язык программирования СИНХРО,

предназначенный для ознакомления с миром параллелизма и особенностями многопоточного программирования [1, 2]. В сравнении с привычными понятиями языков высокого уровня, параллельное программирование использует более широкий спектр понятий, включая понятия как более низкого, так и более высокого уровня. Поэтому учебный язык содержит такие части как низкоуровневое ядро, высокоуровневая оболочка и сборщик программ. Ядро языка — виртуальная машина для многопроцессорного комплекса, на котором будет выполняться результат компиляции многопоточной программы, представленной средствами оболочки. Оболочка нацелена на представление предельно распределённых многопоточных программ, примерно как на языках высокого уровня в рамках парадигмы функционального программирования. Сборщик поддерживает отладку, компиляцию и преобразования программ, что может служить полигоном для формирования навыков метапрограммирования. Сборщик программ устроен как макропроцессор, дополненный контролем синтаксической совместимости параметров макроопределений. В программах допускаются взаимосвязанные потоки с барьерами, что можно рассматривать как вариант ленивых вычислений. Прагматика языка обладает некоторыми отличиями от строгого функционального программирования, направленными на обучение решениям задач эффективности и производительности вычислений. В данной статье описаны такие отличия, предложенные на уровне ядра языка СИНХРО для экспериментов по взаимодействию процессов, полученных в результате компиляции взаимосвязанных потоков.

С середины 1990-ых годов в сфере параллельных вычислений обрели популярность системы функционального программирования, принципы которого оказались достаточно удобными для подготовки многопоточных программ, устроенных из независимых потоков [3]. В данной статье рассматривается возможность подготовки программ с взаимосвязанными потоками. В языке СИНХРО такие взаимосвязи выражаются в терминах, похожих на сети Петри.

Изложение начинается с нестрого описания класса рассматриваемых многопоточных программ и некоторых черт парадигмы функционального программирования, повлиявших на выбор команд виртуальной машины (раздел 1). Даны пояснения по расширению используемых понятий (разделы 2). Затем кратко описан переход от потоков к процессам (раздел 3) и вытекающие из этого решения по системе команд виртуальной машины (раздел 4), допускающей поддержку взаимодействия процессов (раздел 5) с описанием предложенных моделей работы с общей памятью (раздел 6).

1. Многопоточность

Программа на языке СИНХРО строится как комплект потоков, являющихся рядами действий – выражений или директив. Выражения могут использовать размещённые в общей памяти данные, но не изменяют их. Директивы могут изменять хранимые в общей памяти данные. Параллелизм допускает независимость порядка выполнения действий и получения их результатов от порядка записи действий в программе и размещения их результатов в памяти. Для развития навыков сознательного выбора решений, влияющих на изменение данных в общей памяти, предлагается в представлении структур данных учебного языка разнести смысл скобок и разделителей. Скобки означают порядок доступа к элементам структур данных в памяти, а разделители — порядок вычисления элементов при исполнении программы. В результате структуры данных наполняются элементами, порядок вычисления которых может отличаться от порядка вхождения в программу (Таблица 1).

Таблица 1
Структуры данных в языке СИНХРО

БНФ	Примечание
Структура::= '('Выр(';' Выр)…')'	Список последовательно вычисляемых элементов, заполняемый в порядке записи в программе.
|'('Выр(',' Выр)…')'	Список, заполняемый элементами в порядке записи в программе, вычисляемыми асинхронно, возможно в другой последовательности.
|'['Выр (',' Выр)…']'	Вектор, заполняемый элементами в порядке записи в программе, вычисляемыми асинхронно, возможно в другой последовательности.
|'['Выр(';′ Выр)…']'	Вектор, заполняемый последовательно вычисляемыми элементами в порядке записи в программе.

Умение учитывать такое различие полезно для понимания оптимизирующих преобразований программ, связанных с разнообразием многопроцессорных комплексов. Различие проявляется в числе допустимых процессов вычисления, что можно видеть на следующих примерах:

1. (a; b) => { a’ b’ ↓b ↓a } % *один допустимый процесс*

2. (a, b) => { b’ a’ ↓b ↓a | b’ ↓b a’ ↓a | a’ b’ ↓b ↓a }
% *три варианта допустимых процессов*
3. [a; b] => { a’ b’ 1:↓a 2:↓b | a’ 1:↓a b’ 2:↓b | a’ b’ 2:↓b 1:↓a }
% *три варианта допустимых процессов*
4. [a, b] => { a’ b’ 1:↓a 2:↓b | a’ 1:↓a b’ 2:↓b | a’ b’ 2:↓b 1:↓a
| b’ a’ 2:↓b 1:↓a | b’ 2:↓b a’ 1:↓a | b’ a’ 1:↓a 2:↓b }
% *шесть вариантов допустимых процессов*

где x’ – вычисление выражения x,

↓x – размещение полученного значения x в памяти,

n:↓x – размещение значения x как элемента вектора с номером n,

=> соответствие выражения и множества допустимых процессов его вычисления,

{ … }– множество допустимых процессов,

| – разделитель элементов множества процессов.

В первом примере согласно порядку записи в программе вычисляется a’, затем b’. После этого в памяти размещается ↓b, получается список (b), затем в этом списке размещается ↓a и получается список (a b). Во втором примере b’ может быть вычислено раньше, чем a’, и его результат может быть сразу размещён ↓b в списке (b) до вычисления a’ и размещения полученного результата ↓a. Это даст такой же список (a b). Выражения «(a, b)» и «(a; b)» функционально эквивалентны, а мощность пространств допустимых процессов вычислений для них отличается. В третьем примере размещение 1:↓a первым элементом вектора может произойти раньше вычисления b’, результат которого размещается вторым элементом вектора 2:↓b, и размещение 1:↓a может быть выполнено после размещения 2:↓b. В четвёртом примере допускается возможность вычисления b’ раньше вычисления a’. Результат совпадает с результатом третьего примера, различна лишь мощность пространств допустимых процессов вычисления. Выражения «[a; b]» и «[a, b]» функционально эквивалентны. При ясном понимании теоретической разницы, на лабораторной практике многие студенты учитывают допустимые процессы лишь первого и третьего примеров, глаза воспринимают текст как последовательность независимо от разделителя. В задачу разработки языка СИНХРО входит поддержка лабораторных работ для формирования навыков учёта полных пространств допустимых процессов.

Для параллельного программирования оказались удобными свойственные представлению и выполнению функций принципы функционального программирования, такие как универсальность, самоприменение, равноправие параметров функций, гибкость ограничений на блоки памяти, неизменяемость хранимых данных и строгость результата

вычислений. Эти принципы открывают полезные возможности, такие как метапрограммирование, верификация и автономность развиваемых модулей на фоне неявных интуитивных моделей непрерывности процессов, обратимости действий и сводимости определений к унарным функциям. Они дают основу для оперирования достаточно полным множеством допустимых процессов. Модели более подробно описаны в статье [4]. В данной статье приведены решения уровня ядра СИНХРО, связанные с выбором между обработкой локальной и общей памяти, в предположении, что такое ядро может быть использовано и как бек-энд при реализации других учебных языков программирования.

2. Параллельные вычисления и производительность

При ознакомлении с параллелизмом полезно несколько расширить свод базовых понятий программирования, чтобы привлекать разнообразие учебных задач, позволяющее сформировать интуитивную картину многопоточности. В однопроцессном программировании нет необходимости в представлении процессора — он существует по умолчанию всегда. При демонстрации явлений параллелизма в роли процессоров выступают разные роботы. учебно-игровые исполнители, программируемые устройства, многопроцессорные конфигурации, поэтому представление процессоров или выполняемых ими процессов становятся столь же необходимым данным как представления значений и функций. Можно вспомнить, что в UNIX унифицировано представление файлов, устройств, заданий и процессов. В языке РОБИК использовался образ фабрики роботов, которых можно было придумывать, тиражировать, именовать, выключать или включать в обстановку по мере развития сюжета учебной игры или оперирования программой. В языке mpC используются номера процессоров. Сети управления взаимосвязанными потоками — не более чем усложнённая структура данных, допускающая равноправное расположение данных, действий и процессоров в узлах сети, примерно как это проектировалось в языке БАРС. Система управления выполнением действий учитывает геометрию сети, допускающую синхронизацию и преобразования фрагментов программ при их сборке, отладке и оптимизации, что образует полигон для учебных задач по метапрограммированию. На так расширенном своде понятий можно предлагать целевые учебные задачи, включая оттеснение маловероятных вычислений, балансировку нагрузки процессоров, представление пространства итераций. Возможны эксперименты по автоматизированному и ручному распараллеливанию программ, моделированию необходимой при отладке идентичности повторных прогонов на суперкомпьютерах и

представлению многоконтактных узлов для эффективного использования нестандартных аппаратных решений.

Пришло время признать, что при переходе к производственным программам и параллельным вычислениям удобство приложения, работоспособность и производительность программ на практике оцениваются выше, чем полнота отладки и повышение эффективности, мало заметные пользователю. Корректность параллельных программ часто наследуется от ранее созданных последовательных программ решения тех же самых задач. Параллелизм привлекается лишь ради ускорения вычислений. Массово применяется изрядное число жизнеспособных или работоспособных программных приложений, корректность которых сомнительна, что нисколько не мешает их популярности. Известны и примеры резкого взлёта удобных языков программирования, существенно проигрывающих в эффективности привычному семейству C/C++/C#. На практике работает иное понимание правильности, обусловленное ожиданиями или привычками пользователя. Оно не формализовано и не вполне согласуется с общепринятыми и научно обоснованными рекомендациями, нацеливающими на эффективность, а также на формальную верификацию, хотя верификация заметно прогрессирует в рамках парадигмы функционального программирования и со временем может смягчить груз ответственности программиста за качество программных продуктов. Возникает специальная задача обеспечивать успех и производительность программ без потери достоинств формальной верификации и системных решений, нередко дающих улучшение, значительно превосходящее теоретические прогнозы.

Обычно в системы производственного функционального программирования включают механизмы практичного компромисса, внешне выглядящие как специальные функции, не нарушающие стиль записи программ. Например, Lisp 1.5 [5], Clisp, Cmucl, Clojure и другие члены семейства Lisp предоставляют варианты функций, обладающих разными свойствами при сохранении функциональной эквивалентности. В их числе контроль типов данных, схемы циклов, возможность восстановления данных, программируемые прогнозы времени счёта и объёма расходуемой памяти, мемоизация и псевдофункции, дающие доступ к разным устройствам, начиная со средств ввода-вывода и обработки файлов.

3. От потоков к процессам

По умолчанию виртуальная машина языка СИНХРО имеет хотя бы один процессор, на котором происходит процесс выполнения основной программы. Компилятор по многопоточной программе, представленной

средствами высокого уровня, строит функционально эквивалентную ей многопроцессорную программу низкого уровня. Компилятор преобразует комплект потоков в комплекс процессов, каждый из которых обычно выполняется одним процессором, если не возникает проблем неравномерной загрузки процессоров. Система команд ядра выбрана из соображений удобства конструирования шаблонов компиляции функций и операций языка СИНХРО. Предполагается, что ознакомление с параллелизмом включает упражнения на конструирование шаблонов, ориентированных на различные многопроцессорные конфигурации.

Контекст выполнения программы при переходе к процессам становится общей памятью, данные из неё доступны всем процессам многопроцессорной программы. Общая память содержит представления имён и значений глобальных переменных, возможно изменяемых разными процессами с помощью так называемых «деструктивных» функций, работающих подобно привычным присваиваниям. Разница в том, что каждая деструктивная функция обладает чистым функциональным эквивалентом, позволяющим отладку выполнять в два шага. Сначала безопасная отладка под защитой принципа неизменяемости данных, затем – повышение эффективности выборочной заменой чистых функций на их деструктивные аналоги, более экономно расходующие память. Память состоит из двух частей – блока регистров прямого доступа к переменным и кучи произвольных структур данных для хранения значений переменных с протоколами, содержащими старые значения на случай необходимости восстановления состояний памяти после присваивания.

Для простоты изложения здесь не рассматривается учет разнообразия категорий систем команд отдельных процессоров и видов используемой памяти с различной дисциплиной функционирования, хотя в учебных задачах всё это имеет место. Механизмы освобождения памяти обычно требуют приостановки вычислений, что снижает общую производительность программы. Это особенно заметно при необходимости освобождать общую память, приостанавливая все процессы комплекса. Возможные подходы к смягчению этой проблемы рассмотрены в разделе 6.

4. Виртуальная машина

При функционировании виртуальной машины, в отличие от реальной, число процессоров может изменяться, что влечёт появление динамических запросов к памяти, не всегда заметных при статическом анализе и компиляции программы. Кроме обычных процессоров в выполнении программы могут участвовать дополнительные устройства, которые можно рассматривать как специальные процессоры, способные выполнять разные действия по запросам обычного процессора. Каждый обычный процессор

выполняет один процесс, порождаемый одним рядом команд. Шаги разных процессов не синхронизованы, но могут происходить одновременно.

Учитывая особенности компиляции языков, поддерживающих парадигму функционального программирования, при описании системы команд ядра языка СИНХРО используется расширение машины SECD, предложенной П. Ландиным для спецификации аппаратной реализации языка Lisp. Машина работает над регистрами, приспособленными к хранению списков. При определении виртуальной машины языка СИНХРО к регистрам машины «SECD» добавлен регистр «m» (Memory) для описания команд над общей памятью. Состояние процесса полностью определяется содержимым этих регистров. Система команд виртуальной машины поддерживает выполнение результата компиляции функций и операций языка СИНХРО. Часть команд определены в книге П. Хендерсона [6], другие описаны в статье [7], здесь показаны лишь те, что нужны для иллюстрации взаимодействия потоков и работы с общей памятью.

M(SECD)+ – обозначение многопроцессорного комплекса над общей памятью, оно символизирует, что M – общая память для всех процессоров, а (SECD)+ – что хотя бы одни процессор обязателен, общее число процессоров произвольно и не исключено изменение их числа в динамике.

Для четкого отделения обрабатываемых данных от остальной части списка в описания команд используются обозначения, как в книге П. Хендерсона:

(x . L) – это список, в котором первый, головной элемент списка – x, а остальные, хвостовые находятся в списке L;

(x y . L) – в таком списке первый элемент – x, второй элемент – y, остальные находятся в списке L и т. д.

При описании шагов процесса используются следующие обозначения:

s – Stack – стек для окончательных и промежуточных результатов,

e – Environment – контекст для размещения локальных значений переменных,

c – Control_list – управляющая вычислениями программа,

d – Dump – резервная память для обеспечения надёжности вычислений.

m – Memory – общая память, используется как внешний регистр, доступный всем процессам.

m s e c d → m' s' e' c' d' – шаг процесса – переход от старого состояния к новому.

m s e c d → { m' s' e' c' d' | … }
– переход, меняющий состояние группы процессов.
{ m s e c d | …} → m' s' e' c' d' – переход, зависящий от группы процессов.
AM = m s e c d – именование текущего состояния процесса на одном процессоре.
AMk – имя отдельного процесса AM с номером k. Процесс знает свой номер «k», у основного процесса номер 0 – (AM0).

Шаги работы виртуальной машины происходят как изменение состояния процесса или изменение состояний группы процессов. Возможны переходы, требующие учёта состояний группы процессов.

Для поддержки параллельных вычислений требуется несколько команд по организации комплексов процессов (неупорядоченных наборов – KIT) и рядов действий (последовательностей – ROW) с передачей их результатов и явной обработкой ошибок. Процессы образуются следующими командами:

KIT – комплекс из процессов для выполнения рядов действий, построенных в результате компиляции комплекта потоков;
ROW – ряд действий, созданный при компиляции потока в один процесс;
REZ – результат процесса с обозначением его номера размещается в стек;
WAIT – ожидание приостановки процесса (завершения или сообщения);
SEND – сообщение и его тэг передаются указанному процессу;
NEXT – ожидание полного завершения предыдущего действия процесса.

AM0 = m (2 [1] . s) e (KIT c1 c2 . c) d → { AM0 = m s e c d
| AM1 = @m s e c1 d
| AM2 = @m s e c2 d
}

Порядок порождения процессов произвольный. Процесс AM0 работает как монитор на основном процессоре и существует по умолчанию. В данном примере в произвольном порядке создано два процесса AM1 и AM2, для каждого из которых порождён свой процессор. Общая память для порождённых процессоров доступна через ссылку.

[1] Для простоты иллюстраций рассматривается два процесса, а в процессе два действия.

Введено обозначение @x – адрес позиции «x» или доступ к общей памяти в целом из процессора.

Процесс выполняется как ряд действий. Виртуальная машина поочерёдно запускает действия ряда, выделяя каждому из них свой процессор. Запуски действий происходят в порядке записи в программе, а завершения действий могут происходить в произвольном порядке, что соответствует множеству допустимых процессов показанному на примерах раздела 1.

```
AMt = m (2 . s) e (ROW c1 c2 . c) d → { AMt = m (1 . s) e ( ROW c2 . c) d
                                      | AM1 = @m (1 . s) e ( c1 SEND 1 AMt) d
                                      }
```

Процесс AMt запускает из двух предстоящих действий первое – c1, создавая для него свой процесс AM1, и уменьшает счётчик предстоящих действий ряда на 1. В стеке процесса стоит его номер. Созданный процесс AM1 сообщит процессу AMt о завершении выполнения действия c1.

```
AMt = m (1 . s) e ( ROW c2 . c) d → { AMt = m s e c d
                                   | AM2 = @m (2 . s) e (c2 ( SEND 2 AMt)) d
                                   }
```

Потом, когда-нибудь, процесс AMt запустит второе действие – c2 и для него создаст другой процесс AM2. Уже оставался 1 элемент – выход из рекурсии порождения процессов[2] для выполнения ряда действий. Завершилось ли первое действие – пока не известно. Если нужна строгая очерёдность действий, её надо при компиляции обеспечивать сообщениями или командой NEXT.

По мере завершения ряда действий процесса выполняется сборка результата командой REZ, записывающей в стек число результатов комплекса процессов или результаты действий с номерами потоков.

```
{ AMt = m s e c d                    % c2 уже запущен,
| AM1 = @m (a 1 . s) e (REZ) d
      % получен результат c1 для передачи в AMt,
      % возможно требующий свёртки для получения строгого результата.
| AM2 = @m (2 . s) e (WAIT AM1 c2) d  % c2 дожидается завершения c1
}       % переход по готовности трёх процессов AMt, AM1, AM2
```

[2] Здесь в процессе два действия

```
                → { AMt = m (1 a . s) e (ass 2 . c) d
        % AMt забирает результат c1 и ждёт свёртки с результатом c2.
                    | AM2 = @m (2 . s) e c2 d
        % c2 становится исполнимым фрагментом
                        } % изменение состояния двух процессов AMt и AM2
```

ass — представление функции, выполняющей свёртку указанного числа элементов в стеке и выдающей одно данное, рассматриваемое как строгий результат. Это может быть функция, такая как список, структура, сумма, произведение, максимум, минимум, последний и т. п. Со временем и второй процесс будет завершён, что позволит основному процессу выполнить свёртку результатов этих двух процессов.

5. Неизменяемость и/или восстановление данных

Локальные переменные, хранимые в памяти обычных процессов, подчинены принципу неизменяемости, а глобальные переменные в общей памяти доступны через блок прямого доступа, для них вместо неизменяемости поддержан механизм восстановления оттеснённых присваиваниями значений. Наличие чисто функциональных эквивалентов для деструктивных функций гарантирует гладкий спуск от строгого функционального программирования в направлении дозированной императивности. Для возможности восстанавливать значения переменных они сопровождаются протоколом, в котором новые значения размещаются вслед за именем переменной, а к прежним данным можно вернуться при необходимости, например, при отладке посмотреть историю изменения значений. При выполнении обратимых действий допускается и обратимость воздействий на общую память, хранящую переменные с неизменяемыми указателями на изменяемое значение. Каждый элемент общей памяти в любой момент времени обрабатывается только в одном процессе программы, подобно захвату-освобождению файла или устройства. В протоколе изменений можно видеть какой процесс изменил значение данной переменной и, при необходимости, вернуть утраченное значение.

Для описания команд работы с общей памятью используются дополнительные обозначения:

[n] – содержимое n-го элемента общей памяти.

r=[…] – установка значения регистра виртуальной машины.

m=**[**. . . x … y . . .**]** – можно указывать отдельные элементы памяти.

[… Var**:** Val … **]** – размеченное множество, содержащее элемент Val , помеченный меткой Var.

Регистр «m» представляет собой размеченное множество, метки в котором являются именами переменных и допускают прямой доступ, а помеченные элементы расположены в куче, содержат значения переменных и протоколы их изменения, выглядящие как последовательность имён процессов с установленными этими процессами значениями.

Если память «m» содержит для переменной X хранимое в памяти значение xt, установленное процессом AMt, то хранящийся в памяти элемент имеет вид «X**:** (xt AMt xt AM1 x1 AM2 x2 . . .)», где помеченный именем переменной X список содержит её текущее значение xt, а далее размещены имена процессов и ранее установленные этими процессами значения переменной. Здесь показано, что переменная X ранее имела значения x1 и x2, установленные процессами AM1 и AM2 соответственно, а теперь процесс AMt установил ей значение xt. Если после этого процесс AMk присвоит переменной X значение xk, то после такого изменения элемент памяти приобретёт вид:

m'=[. . . X**:** (xk AMk xk AMt xt AM1 x1 AM2 x2 . . .)]

Произойдёт побочный эффект присваивания, допускающий при необходимости восстановление прежних значений.

Воздействия на общую память сводятся к пересылкам и обменам данными:

LD – размещение в стек «s» константы из регистра управления «c»;

MLL – пересылка головного элемента из одного списка в другой;

MLV– пересылка головного элемента списка в элемент вектора;

CHNG – обмен данными в общей памяти комплекса.

При пересылке элемент из списка исчезает, а появляется в другой структуре данных. Команды пересылок и обмена данными в общей памяти рассматриваются как средства низкого уровня для профилактики возникновения временных интервалов между парами взаимосвязанных директив, воздействующих на память на уровне аппаратуры.

Для примера рассмотрим подробнее команду CHNG — обмен данными в общей памяти. При компиляции выражения «X <=> Y», представляющего в потоке R обмен значениями переменных X и Y, где X=x и Y=y, в скомпилированном для него коде процесса AR будет выстроен список команд вида «(LD Y CHNG X . c)». Это значит, что сначала произойдёт загрузка имени переменной Y в стек текущего процесса, затем будет обмен значениями с переменной X из регистра управления. Обмен значениями происходит в общей памяти, если в ней уже расположены значения этих переменных. Получается, что при организации обмена

компилятор размещает в стеке имя одной переменной, а в регистре управления программой – имя другой переменной.

AR = m s e (LD Y CHNG X . c) d → m (Y . s) e (CHNG X . c) d

% *Сначала команда LD подготовит первого участника обмена*

Пусть выполнена команда LD и память имеет состояние, содержащее значения переменных X и Y, а Px и Py – протоколы с прежними значениями этих переменных соответственно:

m=[. . . X**:** (x . Px) . . . Y**:** (y . Py) . . .]

Таково условие возможности выполнять команду обмена:

AR = m=[. . . X**:** (x . Px) . . . Y**:** (y . Py) . . .] (Y . s) e (CHNG X . c) d
→ m'=[. . . X**:** (y AR y . Px) . . . Y**:** (x AR x . Py) . . .] ((y x) . s) e c d)

Прежние протоколы Px и Py для переменных X и Y остались без изменений, но в новых протоколах будет (X=y, а Y= x) с указанием, что это сделал процесс AR. Здесь процесс AR производил обмен значениями x и y между переменными X и Y, в результате обмена обе переменные получили новые значения, они в протоколах сопровождены именем процесса AR, выполнявшего обмен, между действиями обмена стороннее вмешательство невозможно. Сформирован строгий результат из новых значений переменных X и Y – (y x).

6. Модели общей памяти

Освобождение общей памяти может использовать решения, подобные опробованным в практике операционных систем при организации управления распределёнными системами с совмещением работы независимых программ. Реализация таких решений использует специальные структуры данных. Здесь рассмотрим варианты использования паспорта процесса и выделения пассивного процессора общей памяти.

Допустим, что каждый локальный процесс вместо доступа к общей памяти обладает вектором с именами или адресами используемых в нём глобальных переменных, что можно рассматривать как паспорт процесса, работающий как размазанная переменная, доступная в разных процессах. При освобождении локальной памяти процесса формируется новое, возможно уменьшенное, состояние этого вектора, доступное в любой момент и процессору общей памяти. Обычные процессы вместо полного доступа к общей памяти «m» в таком случае обладают лишь вектором «v» для доступа к определённым регистрам из блока прямого доступа. Вместо

формата «m s e c d» тогда используется формат «v(m) s e c d», где «v(m)» – регистр для паспорта процесса, использующего общую память «m».

v(m) s e c d → v'(m') s' e' c' d' – переход от старого состояния виртуальной машины к новому, где «v» – вектор доступа к глобальным переменным из общей памяти «m».

Обычные процессы несколько изменяют метод взаимодействия с общей памятью. Она уже не принадлежит им равноправно, а доступна через паспорт – вектор указателей на данные в общей памяти. Разницу можно показать на примере команды MLL – пересылка из головного элемента одного списка в другой список, где этот элемент становится головным. Списки доступны через указатели x и y для переменных X и Y соответственно.

Пусть паспорт процесса AM содержит указатели на значения двух переменных X и Y. Это значит, что в общей памяти «m» сохраняются соответствующие этим переменным списки со значениями x и y, ранее установленные возможно другими процессами.

v (m)=[X: @x Y: @y …]
m=[X: @x … Y: @y … X: (x AMi (xh . xt) . Px) … Y: (y AMj (yh . yt) . Py)
…]

По указателям на значения этих переменных x и y в памяти M доступны данные:

«x=(xh . xt)» и «y=(yh . yt)»
m=[x=(xh . xt) y=(yh . yt) % *значения переменных X и Y до обмена*
X: @x Y: @y % *паспорт процесса AM*
X: (x AMi (xh . xt) . Px) Y: (y AMj (yh . yt) . Py) % *данные переменных X и Y*
…]

Новое состояние памяти должно приобрести вид:

m'=[x=xt y=(xh yh . yt) % *значения после обмена*
X: @x Y: @y % *паспорт без изменений*
X: (*x* AM *x* AMi (xh . xt) . Px) Y: (*y* AM *y* AMj (yh . yt) . Py)
% *данные о переменных изменены при неизменном указателе*
% указан процесс AM*, изменивший значения по указателю*
...]

В таком случае можно выполниеть команду MLL для переменных X и Y.

v(m) (X . s) e (MLL Y . c) d → v(m') (X Y . s) e c d

Или более подробно:

v(m=[x=(xh . xt) y=(yh . yt) X: @x Y: @y
X: (x AMi (xh . xt) . Px) Y: (y AMj (yh . yt) . Py]
…) (X . s) e (MLL Y . c) d
→ v(m'=[x= xt y=(xh yh . yt) X: @x Y: @y

X: (*x* AM *x* AMi (xh . xt) . Px) Y: (*y* AM *y* AMj (yh . yt) . Py)
...]) (X Y . s) e c d

Изменилось состояние памяти «m», доступное по прежним адресам изменённых значений *x* и *y,* и пополнились протоколы оттеснёнными значениями переменных X и Y. Паспорт не изменился, но он теперь ссылается на изменённое состояние общей памяти. Такой подход при освобождении общей памяти даёт возможность частичного освобождения памяти, не задействованной в объединении паспортов локальных процессов, без приостановки всех процессоров.

Несколько проще устроена модель со специальным процессором общей памяти, поддерживающим пассивный процесс, память которого размечена на занятые и свободные регистры, а команды выполняются лишь по запросам из активных процессов. В таком случае многопроцессорный комплекс состоит из обычных процессоров и одного процессора общей памяти, с которым могут взаимодействовать обычные процессоры. Между собой процессоры взаимодействуют только через общую память. Каждый обычный процессор поддерживает активный процесс из ряда команд, среди которых встречаются команды запросов к пассивному процессу общей памяти. Это обычные команды, выполняемые по ходу активного процесса без особенностей, управляющие доступом к общей памяти, всеми обменами данных и пересылками значений переменных, упомянутыми в разделе 5.

Команды запроса активных процессов к общей памяти имеют двойников среди команд пассивного процесса общей памяти. Это команды типа ленивых вычислений, возбуждаемые при выполнении запросов из активных процессов. Команды пассивного процесса общей памяти структурированы в ряд очередей, каждая из которых соответствует активному процессу и содержит двойников запросов в том же порядке, что и в активном процессе.

Пара запрос из активного процесса и его двойник из пассивного процесса здесь названа «дублет». Он выполняются неразделимо, одновременно, в стиле рандеву языка Ada. Механизм рандеву обеспечивает исключение случайного вмешательства сторонних процессов в работу общей памяти (см. Таблица 2).

Таблица 2

Дублеты – парные команды – срабатывают только неразрывно вместе.

Запросы	*Двойники*	*Неразделимая пара*	*Примечание*
GIVE	TAKE	GIVE-TAKE	Запрос на переменную
SAFE	WRITE	SAFE-WRITE	Запись значения
READ	VAR	READ-VAR	Запрос значения
UNDO	UNDO	UNDO-UNDO	Восстановить прежнее
FREE	FREE	FREE-FREE	Освободить память

При описании дублетов используются дополнительные обозначения:

H – Имя переменной для данного в общей памяти. Все имена различны, они известны во всех процессах, определены при компиляции.

@H – адрес текущего значения переменной H в регистре, хранящем и протокол изменения значений – историю присваиваний.

< @H … > – шкала свободных регистров, содержащая адрес @H.

K – кратность доступа к переменной в текущий момент — число процессов, использующих переменную.

H*K – имя переменной с указанным числом использующих её процессов.

((v s e (C1 . c1) d | v s e C2 . c2) d)) – одновременное срабатывание команд C1 и C2, первых из регистров «c» двух процессов.

Для примера рассмотрим парную команду-дублет GIVE-TAKE. При её выполнении на стеке процесса «s» имя переменной H замещается на адрес её значения @H, а имя переменной заносится в паспорт процесса «v». Одновременно @H адрес в общей памяти исчезает из шкалы свободных регистров.

```
(( v (H . s) e (GIVE . c) d → (H . v) (@H . s) e c d      % активный процесс
                                  % адрес для значения в стеке
 | m=[<@H … > … ] s e (TAKE . c) d → m'=[… H*1 ...] s e c d % пассивный
        % новая переменная, счётчик был установлен в 0, теперь он = 1.
 )) % команды GIVE и TAKE срабатывают только одновременно

% переменная H уже была, ей уже выделена память другим процессом.
 (( v (H . s) e (GIVE . c) d → (H . v) (@H . s) e c d   % активный процесс
 | m=[… (H*K): Ph …] s e (TAKE . c) d → m'=[… (H*(K+1)): Ph …] s e c d
 )) % команды GIVE и TAKE срабатывают только одновременно
```

Описанная схема работы с памятью позволяет объединять методы освобождения памяти типа «сборки мусора» с методами счётчиков доступа при условии исключения слияния ссылок.

Заключение.

В данной статье описаны решения, принятые в учебном языке СИНХРО, предназначенном для ознакомления с параллелизмом и особенностями многопоточного программирования, а также для приобретения навыков организации взаимодействия процессов над общей памятью и профилактики страха перед побочными эффектами при определении сценариев отладки учебных программ. Главные из них – поддержка независимости порядка вычислений от порядка записи выражений в программе в соответствии с принципом независимости параметров, расширение понятия «данное» на представление процессоров и процессов, а понятия «структуры данных» – на сети взаимодействующих потоков в соответствии с принципом универсальности, учёт динамики критериев при переходе от одной парадигмы программирования к другой в соответствии с принципом гибкости ограничений, описание системы команд виртуальной машины для определения учебного языка параллельного программирования в соответствии с принципом самоприменимости. Кроме того, предложены две модели работы с общей памятью, позволяющие от неизменяемости данных перейти к методам восстановления данных. Такие решения смягчают противопоставление методов строгого функционального программирования, нацеленного на правильность программ, и императивно-процедурного программирования, позволяющего привычными способами повышать эффективность и производительность программ.

Благодарность. Благодарю Дмитрия Владимировича Мажугу за экспериментальную реализацию ядра языка СИНХРО на языке функционального программирования Clojure

Литература

1. Городняя Л.В. Язык параллельного программирования СИНХРО, предназначенный для обучения. – Новосибирск : ИСИ им. А.П.Ершова СО РАН, 2016. – 30 с. (Препринт/ИСИ СО РАН; N 180).
2. Городняя Л.В. О курсе «Начала параллелизма» // Ершовская конференция по информатике. Секция «Информатика образования». 27 июня – июля 2011 года. Новосибирск : ИСИ им. А.П.Ершова СО РАН. – с. 51-54.

3. Пеппер П., Экснер Ю., Зюдхольд М. Функциональный поход к разработке программ с развитым параллелизмом //Системная информатика. Вып 4. Методы теоретического и системного программирования. − Новосибирск: Наука. Сиб. изд. фирма, 1995. − с. 334-360.
4. Городняя Л.В. О функциональном программировании // Компьютерные инструменты в образовании (в печати).
5. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106 p.
6. Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.
7. Городняя Л.В. Абстрактная машина языка программирования учебного назначения СИНХРО // Вестник Новосибирского государственного университета. Серия: Информационные технологии 2021. № 4. – С. 16–35.

References

1. Gorodnyaya L.V. Language of Parallel Programming SYNHRO, Intended for Training // Novosibirsk, Preprint A.P. Ershov IIS SB RAS № 180, 2016, – 30 p.
2. Gorodnyaya L.V. About the course "The Beginnings of Parallelism" // Ershov Conference on Informatics. Section "Computer science of education". June 27 – July 2011. Novosibirsk : A.P. Ershov IIS SB RAS. p. 51-54.
3. Pepper P., Exner J., Südhold M. A functional approach to the development of programs with advanced parallelism // System Informatics. Issue 4. Methods of theoretical and system programming. – Novosibirsk: Science. Sib. ed. firm, 1995 – p. 334-360.
4. Gorodnyaya L.V. On Functional Programming // Computer Tools in Education (in press)
5. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106 p.
6. Henderson P. Functional programming. – Moscow: Mir, 1983 .– 349 p.
7. Gorodnyaya L.V. Abstract machine of the programming language for educational purposes SYNHRO // Bulletin of the Novosibirsk State University. Series: Information technologies 2021. № 4, p. 16-35.