Научная статья

УДК 004.434
DOI 10.25205/1818-7900-2023-21-2-5-17

К вопросу о разрешении проблемы конкурирующих процессов в процесс-ориентированном программировании

Дмитрий Андреевич Пермяшкин

Институт автоматики и электрометрии СО РАН
Новосибирск, Россия

d.permiashkin@g.nsu.ru, https://orcid.org/0009-0004-6045-3228

Аннотация

В современном мире значительная часть фабрик и промышленных производств уже управляются программируемыми микроконтроллерами и число такого рода производств непрерывно растет. Данный процесс тесно взаимосвязан с идеями Индустрии 4.0, а если быть точнее – с идеей полной автоматизации производственных процессов для облегчения принятия решений человеку. И одновременно уменьшения числа принимаемых человеком решений вплоть до полного отсутствия человека в роли принимающего решения. Для этого требуются управляющие алгоритмы, которые должны работать по событиям, учитывать наличие и тесный контакт с внешней средой и иметь повышенные требования к устойчивости к внутренним отказам и отказам оборудования. Разработанная в Институте автоматики и электрометрии СО РАН процесс-ориентированная парадигма программирования, рассматриваемая в данной статье, прекрасно подходит для разработки управляющих алгоритмов такого рода. Данная парадигма учитывает тот факт, что производственный процесс представляет собой очень большое число одновременно работающих процессов, тесно связанных с элементами реального мира. Отсюда вытекает необходимость решения проблемы конкурирующих процессов для обеспечения требуемой степени устойчивости к отказам. В данной статье будет поставлена и проанализирована проблема конкурирующих процессов. Для этого в работе был произведен анализ существующих решений проблемы конфликтов при параллельной работе произвольного числа процессов с целью анализа применимости данных методов для процесс-ориентированных программ или же возможности адаптации некоторых методов. В результате будет получен ответ на вопрос, насколько эффективно процесс-ориентированная парадигма позволяет решать конфликты при параллельном исполнении программы и насколько сильно удовлетворяет требованию отказоустойчивости.

Ключевые слова

индустриальное программирование, параллельные программы, процесс-ориентированное программирование, процесс-ориентированные языки

Для цитирования

Пермяшкин Д. А. К вопросу о разрешении проблемы конкурирующих процессов в процесс-ориентированном программировании // Вестник НГУ. Серия: Информационные технологии. 2023. Т. 21, № 2. С. 5–17. DOI 10.25205/1818-7900-2023-21-2-5-17

On Solving Concurrent Process Problem in Process-Oriented Programs

Dmitry A. Permiashkin

Institute of Automation and Electrometry SB RAS
Novosibirsk, Russian Federation

d.permiashkin@g.nsu.ru, https://orcid.org/0009-0004-6045-3228

Abstract

Prevailing portion of the factories and manufacturers are controlled by programming microcontrollers in the modern world. And the portion keeps growing which is closely tied to processes of the Fourth Industrial Revolution. Precisely, with an idea of fully automated manufactures to help humans to make less decisions and make them faster. Or to exclude humans from the decision making process at all. Due to that, there is a need for the controlling algorithms which should react to the different events, be aware of the external world and be tolerant to both internal and hardware failures. There is a process-oriented paradigm which was developed in Institute of Automation and Electrometry SB RAS and suits perfectly for automatization of such algorithms. This is achieved by splitting the algorithm into huge amounts of the small parallel processes highly tied to the elements of the real world. Which is how real processes on real manufactures work. This is where the conflicts during concurrent programming appear. And because there is a fault tolerance requirement there is a need to solve those conflicts. This work presents the analysis of already existing solutions to the conflicts during concurrent programming with the goal of either reusing those solutions in process-oriented programming or adapting them to it. As a result, there is an answer on how effective the process-oriented paradigm is in solving those kinds of conflicts and how fault tolerant those programs are.

Keywords

industrial programming, concurrent programming, process-oriented programming, process-oriented language

For citation

Permiashkin D. A. On Solving Concurrent Process Problem in Process-Oriented Programs. *Vestnik NSU. Series: Information Technologies*, 2023, vol. 21, no. 2, pp. 5–17. DOI 10.25205/1818-7900-2023-21-2-5-17

Введение

В современном мире активно идет процесс автоматизации. Автоматизируется складской учет, производства и даже уже сам процесс автоматизации. Для автоматизации таких процессов очень важна возможность параллельного исполнения. В связи с этим в данной области популярны системы на основе программируемых логических контроллеров (ПЛК), которые не только позволяют поддерживать параллелизм на очень высоком уровне, но также требуют меньше электричества, имеют меньшее тепловыделение и более устойчивы к воздействию внешней среды (тепловому и электрическому). Для облегчения разработки самих контроллеров и программного обеспечения был разработан стандарт IEC 61131 в 10 частях, из которых самой важной частью является третья, описывающий стандарты языков для программирования ПЛК.

Третья и последняя версия стандарта IEC 61131-3 вышла в 2013 году [1], что не кажется недостатком на первый взгляд. Но тут встает вопрос, насколько успели повлиять на стандарт идеи четвертой индустриальной революции, или же Индустрии 4.0, учитывая вынужденную медлительность разработки любого стандарта. Данный вопрос очень важен, поскольку в ключевых темах находится сильная взаимосвязь всех элементов и активная помощь в принятии решений человеку с одновременным исключением необходимости принятия решения человеком в как можно большей части задач. В связи с этим уже ведется разработка новых парадигм программирования и языков на них с учетом Индустрии 4.0. Одним из основных языков, реализующих идеи из Индустрии 4.0, является рассматриваемая в данной статье процесс-ориентированная парадигма программирования, разработанная в Институте автоматики и электрометрии СО РАН и уже проверенная на практике при автоматизации производств [2, 3, 4].

Этот подход обеспечивает комфортное создание и сопровождение управляющих алгоритмов, отличие которых от вычислительных алгоритмов заключается в наличии внешней среды, необходимости работать по событиям, в частности, временным, параллелизму, и повышенным требованиям по надежности, в частности, требованиям устойчивости к внутренним отказам и отказам оборудования.

Для такого рода алгоритмов очень важна возможность параллельного исполнения. И не просто возможность выполнения нескольких никак не связанных процессов, но возможность выполнения большого числа процессов, связанных между собой разделяемыми ресурсами – элементами реального мира. Поэтому при автоматизации такого рода всегда встает вопрос о разрешении конфликтов, возникающих при параллельных вычислениях.

Данная сфера уже достаточно активно изучается, и первый алгоритм, решающий задачу о конфликтах при параллельных вычислениях при произвольном числе параллельных программ, был опубликован в 1965 году. Однако в силу специфики управляющих алгоритмов методы, разработанные в рамках теории параллельных вычислений, часто не могут быть использованы напрямую и нуждаются в адаптации. Основные причины – относительно низкая вычислительная мощность узлов и необходимость обеспечения устойчивости в случае отказов в системе. Таким образом, алгоритмы разрешения конфликтов должны иметь низкие требования к ресурсам и быть способны в случае отказа приводить оборудование в безопасное состояние. Например, при автоматизации процесса выращивания монокристаллического кремния разрушение тигля с расплавленным кремнием должно вызывать отключение нагревателя и перевод тигля в крайнее нижнее положение [5].

Таким образом, эффективная и надежная реализация межпроцессного взаимодействия в рамках процесс-ориентированного программирования является актуальной задачей.

В статье анализируются существующие подходы к разрешению конфликтов в параллельных программах и распределенных системах управления, рассматривается специфика межпроцессного взаимодействия в процесс-ориентированной модели программы, обсуждаются применимость существующих подходов к организации межпроцессного взаимодействия в рамках процесс-ориентированной парадигмы.

Обзор существующих подходов к разрешению конфликтов в параллельных программах

Для целей решения конфликтов параллельная программа состоит из двух частей – процессы и переменные. Процесс – это последовательный набор подписанных инструкций (или же операций). Переменные содержат какие-либо значения и могут быть доступны либо всем процессам (публичная), либо только внутри одного конкретного процесса (частная или же локальная). При этом у любого процесса есть счетчик команд – особая переменная, которая указывает на текущую выполняемую операцию в процессе. Процесс выполняет некоторый бесконечный цикл из двух секций:

- некритическая секция, где любая операция изменяет только частные переменные;
- критическая секция, где операции могут изменять публичные и частные переменные.

Введем понятие примитива – последовательного набора операций. Примитив будет атомарным, если гарантируется, что в момент исполнения примитива любая другая операция не будет разрешенной. Поэтому далее атомарный примитив будет рассматриваться как одна операция.

Состоянием назовем отображение некоторых значений во все переменные программы. Состояние может быть начальным – возможным при начале исполнения программы. Назовем операцию s разрешенной, если можно перейти из состояния t в состояние u после выполнения операции s. История – это последовательность из состояний и операций, где любая операция

является разрешенной. А конфликт – момент в истории, где изменение очередности выполнения разрешенных операций меняет итоговое состояние.

В основном стандартные подходы к решению можно разделить на два класса:

1. Использующие взаимное исключение:
– на основе общей памяти,
– при помощи сообщений;
2. Не использующие взаимное исключение.

Первый алгоритм для разрешения конфликтов на основе взаимного исключения был опубликован Дийкстрой в 1965 году в статье [6] и расширяет критическую секцию, разделяя ее на секцию входа, саму критическую секцию и секцию выхода. Далее к секциям предъявляются дополнительные требования:

- Процесс не может остановиться в критической секции.
- Любая инструкция либо в любом состоянии не является разрешенной, либо она исполнилась (требование честности для истории).
- Секции входа и выхода должны гарантировать, что критическая секция исполняется максимум одним процессом в любой момент времени (требование исключения).
- Секции входа и выхода должны гарантировать, что если какой-то процесс находится в критической секции, то какой-то другой процесс рано или поздно будет в критической секции (требование свободы от блокирования).

Статья [6] породила много обсуждений, из которых стоит упомянуть статью Кнута [7], где было введено более строгая версия требования свободы от блокирования:

- Секции входа и выхода должны гарантировать, что если какой-то процесс находится в секции входа в критическую секцию, то этот процесс рано или поздно будет в критической секции.

Дальнейшие работы в основном не вводили каких-то новых требований, а сфокусировались на решении самой проблемы в рамках этих требований.

Алгоритмы взаимного исключения на основе общей памяти. Задача исключения в данных алгоритмах решается хранением в общей памяти переменной или переменных, описывающих состояние критической секции. При этом любая переменная, кроме счетчика команд, может одновременно использоваться либо в секциях входа и выхода, либо в критических и некритических секциях. При этом хоть чтение данной переменной в цикле и является рабочим алгоритмом, но для практических задач он не подходит из-за забивания канала память–процессор операциями чтения для проверок. Одним из подходов для решения проблемы забивания канала являются алгоритмы локальной прокрутки (local-spin), где процесс при входе в критическую секцию проверяет локальные копии некоторой переменной (или переменных) вместо проверки публичной переменной.

Первыми алгоритмами локальной прокрутки были алгоритмы прокрутки очереди на основе примитива “чтение–модификация–запись” [8, 9]. В них каждый процесс при входе в критическую секцию записывает свою “позицию” в очереди в виде указателя и ждет, пока ее предшественник не выйдет из секции выхода. В случае системы с общим кэшем они требуют максимум O(1) операций над памятью, но в случае когда у процессов нет общего кэша, то невозможно установить верхнюю границу числа операций над памятью.

Позднее Янг и Андерсон [10] предложили алгоритм на основе примитива “чтение–запись”, где из процессов строится двоичное дерево. В качестве листов дерева лежат сами процессы, а в узлах хранится информация, кто должен исполнять критическую секцию из пары потомков. И когда процесс входит в критическую секцию, он пытается захватить контроль над родительским узлом, а когда выходит из секции, то отпускает его. При этом данный алгоритм на любой системе требует всего лишь O(log N) процессов.

Следующим подклассом решений являются “быстрые” алгоритмы исключения. В них существует путь исполнения за константное время для случая, когда всего один процесс хочет

исполнять критическую секцию. Например, алгоритм за авторством Лампорта [11], который тоже строит дерево, но при помощи “разделителя”, который гарантирует, что не более одного процесса может остановиться в конкретном узле, а другие будут в потомках узла. Сам концепт разделителя в отрыве от самого алгоритма оказался настолько популярным, что его используют для уменьшения пространства имен другие алгоритмы [12, 13, 14]. Без соревнования данный алгоритм требует всего семь операций, но при соревновании данный алгоритм не имеет верхней границы на число операций.

Дальнейшие подклассы алгоритмов с исключением в основном сфокусированы на том, чтобы как можно быстрее решить, какие процессы исключают друг друга, но при этом оставить как можно быстрее путь в случае отсутствия конфликта. Для этого используется как статический анализ – прямая модификация алгоритма Лампорта с локальной прокруткой [15], так и динамический – алгоритм Стиера [16] (напрямую использующий Лампорта), Чоу и Сингха [17] (строящий при помощи “заполнителя” несбалансированное дерево со сбалансированными потомками в левом потомке), анализирующие соревновательность в точке истории и на некотором интервале истории [12].

До этого считалось, что один процессор исполняет один процесс одновременно. В реальности процессор может переключаться между различными процессами по истечению отведенных квантов времени. Алгоритм Фишера [18] требует четкого знания о длительности кванта времени для процесса. Идея состоит в том, что в переменной для исключения хранится номер процесса в критической секции. Когда процесс пытается зайти в секцию (в переменной 0), то при захвате секции он пишет туда свой номер и ждет квант. И если в переменной остался его номер, то можно заходить. Алгоритм Алюра, Аттия и Таунберфелда [19] ослабляет требование точного знания кванта до знания верхней границы оценки кванта. При этом априорное знание данной границы не требуется для алгоритма, для уточнения текущей оценки границы используется алгоритм Лампорта.

Теперь перейдем к подклассу алгоритмов, где не требуется наличия атомарных операций. К сожалению, число таких алгоритмов достаточно мало. Первая причина – нет нужды разрабатывать данные алгоритмы. Поскольку во всех существующих популярных процессорах есть поддержка на аппаратном уровне атомарности примитива “чтение–запись”, то алгоритмы без него интересны только с научной точки зрения. Второй причиной является сложность данных подходов. Отсутствие атомарных операций часто требует очень аккуратной работы с переменными, отведенными под общение между процессами, вдобавок к необходимости правильной разметки критических секций.

Из значимых работ можно отметить серию из двух работ за авторством Лампорта [20, 21], где при попытке входа в критическую секцию процесс опрашивает другие процессы об их состоянии. Сложность процесса опроса зависит от того, насколько строгое требование по честности захода в критическую секцию.

Из главных особенностей подхода Лампорта можно отметить поднятие вопроса об устойчивости к ошибкам и постановке четырех критериев устойчивости:

- Безопасность при выключении – схема не ломается, если процесс завершил свою работу, потому что процесс выставил все переменные для общения в базовые значения и тогда только завершился.
- Безопасность при прерывании – схема не ломается, если процесс прервали в критической секции, потому что процесс выставил все переменные для общения в базовые значения и перешел в некритическую секцию (и далее он может продолжать свою работу).
- Безопасность при ошибках – схема не ломается, если процесс встретил ошибку при своей работе, потому что процесс просто постепенно мягко выключится.
- Самостабилизация – если процесс вошел в ошибочное состояние и далее ошибок не будет, то он вернется в нормальное состояние со временем.

Алгоритмы разрешения конфликтов при помощи сообщений. Технически схема Лампорта из [20, 21] может быть и отнесена сюда, поскольку он уже использовал сообщения для обеспечения общей памяти. И в основном большая часть схем взаимного исключения использует сообщения именно с такой целью – превратить частные переменные в подобие публичных переменных для хранения состояния о секции. Именно поэтому они редко когда показывают сильно отличающиеся результаты [22, 23]. С одной стороны, схемы с общей памятью обычно четко фиксируют число процессов, когда исключение при помощи сообщений может быть организовано среди переменного числа процессов [22]. Но с другой стороны, при автоматизации промышленных производств такие ситуации редки, поэтому дальше взаимное исключение при помощи сообщений рассматриваться отдельно от исключения при помощи разделяемой памяти не будет.

Алгоритмы без взаимного исключения. Основная проблема, с которой сталкиваются разработчики схем без взаимного исключения процессов, – отсутствие поддержки необходимых атомарных примитивов в системах команд современных процессоров. Поэтому данные алгоритмы включают реализацию требуемых атомарных примитивов на основе уже существующих, которая проигрывает реализациям алгоритмов со взаимным исключением по скорости.

Например, в работе [24] авторы смогли подобрать набор инструкций для эффективной реализации требуемых атомарных примитивов. Однако авторы рассматривают ситуации, в которых работа алгоритма приводит к рассинхронизации между различными процессами. Авторы предлагают решать эту проблему через проверку корректности чтения, что занимает дополнительное время.

Критический анализ существующих подходов

Среди всех алгоритмов разрешения конфликтов при параллельных вычислениях в контексте автоматизации производств можно отметить несколько общих проблем. Одна из основных проблем состоит в том, что в основном схемы имеют сложность O(log N) по числу операций. При этом стоит отметить, что производственные процессы состоят из множества мелких операций и, следовательно, программа для автоматизации данных производств будет склонна к большому числу процессов внутри себя. Следовательно, стоимость решения возможных конфликтов может быстро расти, даже если не каждый процесс будет конфликтовать за конкретный ресурс.

Дополнительно стоит отметить, что для разрешения конфликтов требуется организовывать межпроцессную коммуникацию, что требует либо памяти, либо канала между процессами. Другим следствием сильной ограниченности ресурсов является организация многопоточности на малом числе физических потоков. И в случае использования квантов времени для организации такого рода многопоточности может возникнуть проблема, когда много процессов просто ожидают один, хотя при другом распределении квантов или при другом размере процессы могли бы просто работать друг за другом. Если бы планировщик не использовал кванты времени, то данной проблемы можно было и избежать, но тогда возникает требование знания верхней границы времени работы каждого процесса (очень сильное требование).

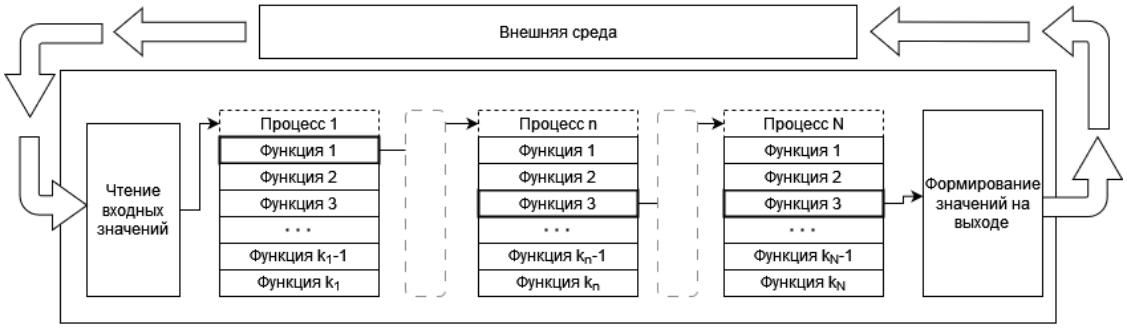
При этом вопрос об отказоустойчивости внутри критической секции перекладывается на разработчика системы полностью. Программист сам должен заботиться о том, чтобы процессы выходили из критической секции нормально в любой ситуации.

Ситуация усложняется в случае мультипроцессорных систем. Даже в варианте системы на одном чипе (SoC) организация межпроцессного взаимодействия усложняется на порядок из-за необходимости организации межпроцессорного взаимодействия и почти гарантированного отсутствия общего кэша. Хотя идея использования мультипроцессорных систем со специализированными под некоторый набор подзадачами является привлекательной для целей автоматизации производства.

Процесс-ориентированная парадигма. Определения и модель

Перед обсуждением, как проблема конкурирующих процессов выглядит в процесс-ориентированном программировании, требуется определить несколько ключевых понятий и описать модель программы.

Процесс-ориентированная парадигма была разработана в ИАиЭ СО РАН [2, 3, 4]. Данная парадигма и реализующие их языки направлены на автоматизацию предприятий и фабрик. Программа в ней представляет собой один гиперпроцесс – упорядоченный набор процессов. Сам же процесс – модифицированный конечный автомат, где у каждого состояния есть некоторая связанная с ней функция. У любого процесса существуют особые состояния: покоя и ошибки. В состоянии покоя процесс может перейти из любого состояния и в нем процесс ничего не делает. И только из состояния покоя процесс может перейти в начальное состояние по внешней причине. Состояние ошибки тоже существует у любого процесса, и оно сигнализирует о произошедшей ошибке в процессе исполнения функции, и из него процесс самостоятельно выйти не может. Функция, связанная с текущим состоянием, исполняется при передаче контроля процессу, и она же отвечает за выбор следующего состояния процесса. Сами функции представляют последовательный набор операций, аналогично процессу из обычных программ. Переменные в процесс-ориентированных программах полностью аналогичны обычным программам. Но у процесса есть важная публичная переменная, в которой он хранит свое текущее состояние.



Цикл процесс-ориентированной программы
Process-oriented program cycle

Исполнение процесс-ориентированной программы происходит в цикле, состоящим из трех частей:

1. Чтение данных из внешней среды.
2. Передача контроля процессам в строго заданном порядке.
3. Запись реакции программы во внешнюю среду.

Таким образом, процесс-ориентированная программа реализует корпоративный параллелизм на одном потоке вычисления. Стоит отметить, что порядок передачи контроля процессам связан с порядковым номером процесса и известен до начала исполнения программы. А сам цикл активируется с некоторым фиксированным заранее известным периодом, и гарантируется, что цикл завершится до начала следующего цикла.

В теории можно ввести состояние функции по аналогии с состоянием процесса, но смысла в этом нет в рамках данной статьи, поскольку далее будет рассматриваться только случай, когда процесс возвращает контроль сам для передачи следующему процессу. Тогда любая функция в процесс-ориентированной программе будет являться атомарным примитивом. Назовем конфликтующими процессами такое подмножество процессов, которые имеют некоторое общее

подмножество публичных переменных, с которыми они работают в рамках одного цикла программы. Конфликт – ситуация, когда от порядка работы конфликтующих процессов зависит состояние программы в конце цикла.

Проблема конфликтов в процесс-ориентированной парадигме

Для удобства анализа рассмотрим некоторую систему, состоящую только из одного клапана. У нее будет всего лишь две операции – открыть и закрыть клапан. Поскольку речь идет об автоматизации производств, то наше вычислительное устройство напрямую работает с содержимым клапана, поэтому любая операция состоит из подачи питания на требуемые элементы в установленном порядке, контроля за состоянием клапана и отключением питания с данных элементов.

Попробуем представить программу в процесс-ориентированной парадигме. Стоит сразу отметить, что единого процесса, скорее всего, не будет. На это есть несколько причин:

- Сама парадигма побуждает дробить алгоритмы на как можно более мелкие элементы. При выполнении данного условия из-за корпоративного параллелизма можно приблизиться к истинному параллелизму (как на ПЛИС). Дополнительно при дроблении на как можно более мелкие элементы отпадает строгая нужда в планировщике и квантовании времени, поскольку квант времени будет порядка времени выполнения одной функции в среднем.
- Процесс является расширением конечного автомата, поэтому он имеет всего одно начальное состояние, что в случае клапана с двумя различными желаемыми результатами будет требовать дополнительное время на обработку “что же хочет пользователь”, что приведет к задержке между сигналом и реакцией системы…

Следовательно, будет два процесса, каждый из которых будет ответствен за свою функцию у клапана. Каждый процесс будет последовательно выполнять функции выставления нужных значений в переменных клапана, анализа состояния клапана и выставления некоторых нейтральных значений в переменные клапана.

Сразу виден сценарий, когда может произойти конфликт – а что если при закрытии клапана мы попробуем открыть клапан? А что делать, если есть ситуации, когда надо закрыть открывающийся клапан? А что если система теперь состоит из нескольких клапанов и есть некоторый порядок, в котором можно открывать и закрывать?

Рассмотрим ситуации по очереди. Первый случай решается относительно просто – надо ввести некоторую публичную переменную, где бы отображалось текущее состояние клапана… И неожиданно такая переменная уже есть – состояние процесса. Поэтому процесс закрытия клапана может проверить состояние открытия клапана и поменять свое поведение. Аналогично решается и вторая проблема – поскольку каждый процесс имеет доступное для всех состояние, то процесс открытия может в начале функции контроля состояния клапана просто проверять состояние процесса закрытия клапана и узнать, что совершается аварийное закрытие клапана. Поскольку набор процессов известен заранее, то данная проверка будет являться проверкой переменной из кэша, а не некоторой процедурой опроса другого процесса.

Перед рассмотрением последнего вопроса рассмотрим несколько иной. Допустим, теперь в системе есть монитор, где отображается текущее состояние клапана. Естественно, его заполняет некоторый отдельный процесс и возникает вопрос, конфликтует ли данный процесс с другими двумя. Формально да, поскольку результат на мониторе будет зависеть от момента, когда процесс отображения будет выполняться в цикле относительно двух других процессов. Но при этом задержка состояния на мониторе и реального состояния системы будет все время стабильной и предсказуемой. Тем более если программист желает разрешить данный конфликт, то из-за гарантии строго порядка исполнения процессов ему достаточно просто раз-

местить процесс отображения после процессов управления путем выставления большего порядкового номера процесса.

Вернемся к последовательной активации нескольких клапанов. Учитывая предыдущий конфликт и его решение, имеется два варианта разрешения конфликта:

1. Всего один процесс на открытие всей системы и один на закрытие.
2. Следить за состоянием некоторых клапанов и не позволять открываться до открытия предыдущих по цепочке. Данный вариант позволяет легко реализовать сложные и разветвленные схемы зависимостей и заодно гарантировать начало открытия клапанов в рамках одного цикла.

Как видно, в большинстве случаев конфликты в рамках процесс-ориентированной программы решаются либо правильной расстановкой порядковых номеров процессов, либо использованием состояний процессов как примитивов синхронизации.

Преимущества и недостатки использования известных методов решения конфликтов в процесс-ориентированных программах

Несмотря на то что методы решения конфликтов в процесс-ориентированных и обычных программах схожи между собой из-за использования идеи критической секции, все же есть некоторые различия между ними и получаемым результатом при разрешении конфликтов. Поэтому можно говорить о преимуществах и недостатках процесс-ориентированных программ в данной сфере по сравнению с обычными программами.

Сразу стоит отметить отсутствие операций с неопределенным временем исполнения (ввод-вывод как пример). Данный факт позволяет обойтись без прерывания хода исполнения процессов из-за истечения некоторого выделенного кванта времени. Поэтому программист будет четко знать места, где программа может потерять управление, и разрабатывать процесс как череду четко определенных критических секций. По итогу каждый отдельный процесс будет захватывать контроль над секцией каждый раз, что может позволить избежать проблемы истощения и взаимной блокировки. Стоит отметить, что в таком случае только на процессе лежит ответственность за реализацию честных вычислений, и если некоторая функция будет иметь неоправданно долгое время исполнения (или же бесконечное), то вся программа может остановиться.

Вторым важным элементом процесс-ориентированной программы является существование явного приоритета всех процессов относительно друг друга и гарантия соблюдения данного приоритета в процессе исполнения. В предыдущем разделе было показано, как данный приоритет позволяет избежать необходимости реализации любого метода решения конфликтов путем правильной расстановки порядкового номера. Таким образом, несмотря на то что как бы конфликт и есть формально, но дополнительных вычислительных усилий он не требует.

Существование состояний у процессов является как положительной стороной процесс-ориентированной программы, так и отрицательной. С одной стороны, их можно использовать в других процессах для разрешения конфликтов и реализации взаимного исключения через состояния процессов. С другой стороны, состояния всех процессов придется все равно синхронизировать между всеми процессам, и в случаях систем с раздельным кэшем это может повлечь большие проблемы, особенно учитывая, что в одной процесс-ориентированной программе число процессов можно оценить цифрой 1000.

Но тут есть пара интересных моментов. Хоть явно и существует требование о том, что любой процесс знает про состояния всех других процессов, но на практике произвольному процессу не интересна большая часть процессов. Если построить граф, где вершинами будут являться процессы, а ребрами будут показаны зависимости процессов от состояний других процессов, то в большинстве случаев такой граф можно поделить на подграфы, которые будут либо являться компонентами связности в исходном графе, либо будут иметь очень малое число

ребер с другими подграфами (относительно ребер внутри). Тогда каждый подграф можно будет исполнять на своем вычислительном устройстве. Данное разбиение уменьшит сложность разрешения конфликтов не только из-за минимизации общения между вычислительными устройствами, а еще потому, что можно будет решать данную задачу в рамках одного подграфа, который будет иметь порядок, сравнимый с числом процессов в обычной программе.

Дополнительно стоит отметить наличие состояния ошибки, в которое достаточно легко перейти и явно сигнализирующее об ошибке в процессе исполнения. При этом поскольку переход в данное состояние аналогичен переходу в любое другое состояние, и если процесс реализует безопасность при выключении, то безопасность при прерывании будет тоже обеспечена. Поскольку вся функция и является критической секцией, то безопасность при ошибке аналогична безопасности при прерывании. Следовательно, для реализации безопасных параллельных вычислений достаточно иметь безопасность при завершении работы функции и самовосстановление. Стоит отметить, что из состояний покоя и ошибки можно перейти только в начальное состояние, где и будут содержаться инициализаторы для локальных переменных. Следовательно, безопасность при завершении работы уже реализована в процесс-ориентированной парадигме и от разработчика требуется только реализовать некоторые процедуры по восстановлению из ошибок, запускающиеся по переходу процесса или процессов в состояние ошибки.

Заключение

В статье была рассмотрена проблема конфликтов при параллельном исполнении различных процессов в программе в контексте промышленной автоматизации. Данный контекст отличается не только требованием реализации данного исполнения на маломощных вычислительных устройствах, но также требованием отказоустойчивости. К сожалению, уже существующие подходы хоть и могут удовлетворять первому требованию, но вопрос отказоустойчивости остается слабо изученным. Поэтому в статье была рассмотрена процесс-ориентированная парадигма как способ написания программ, удовлетворяющим требованиям.

В результате анализа было показано, что процесс-ориентированная программа позволяет решать проблему конфликтов при параллельном исполнении различных процессов путем решения проблемы конкурирующих процессов в данной программе. При этом использование процесс-ориентированной парадигмы позволяет избежать излишних затрат на критические секции и проблем с неправильным определением данных секций разработчиком программы. Дополнительно сама парадигма из-за отсутствия необходимости в дополнительных примитивах синхронизации не требует ввода специальных атомарных операций со стороны производителей вычислительных устройств. Для разработчиков при использовании парадигмы основное преимущество заключается в автоматическом выполнении трех из четырех требований для организации отказоустойчивой параллельной программы.

В качестве дальнейшей работы в этом направлении предлагается рассмотреть вопрос о необходимости требования разделения программы на как можно мелкие процессы. При рассмотрении схемы из нескольких клапанов было сделано предположение, что каждым клапаном управляет свой процесс. Но что, если это не так? Что, если у процесса могло бы быть несколько начальных состояний? Ну и, естественно, остается открытым вопрос о возможности создания средств обнаружения конфликтов в процесс-ориентированной программе при статическом анализе кода.

Список литературы

1. **John K. H., Tiegelkamp M.** Programming Industrial Automation Systems // IEC 61131-3. 2010. DOI: 10.1007/978-3-642-12015-2

2. **Зюбин В. Е.** Язык Рефлекс. Математическая модель алгоритмов управления // Датчики и системы. 2006. № 5. С. 24–30.
3. **Зюбин В. Е.** Статическая балансировка вычислительных ресурсов в процесс-ориентированном программировании // Вестник НГУ. Серия: Информационные технологии. 2012. Т. 10. Вып. 2. С. 44–54.
4. **Зюбин В. Е. и др.** Базовый модуль, управляющий установкой для выращивания монокристаллов кремния //Датчики и системы. 2004. №. 12. С. 17–22.
5. **Булавский Д. В. и др.** Автоматизированная система управления установкой для выращивания монокристаллов кремния //Автометрия. 1996. №. 2. С. 26.
6. **Dijkstra E. W.** Solution of a Problem in Concurrent Programming Control, Pioneers and Their Contributions to Software Engineering, p. 289–294, 2001, DOI: 10.1007/978-3-642-48354-7_10
7. **Knuth D. E.** Additional comments on a problem in concurrent programming control, Communications of the ACM, vol. 9, no. 5, p. 321–322, May 1966, DOI: 10.1145/355592.365595
8. **Anderson T. E.** The performance of spin lock alternatives for shared-money multiprocessors, IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, p. 6–16, 1990, DOI: 10.1109/71.80120
9. **Mellor-Crummey J. M., Scott M. L.** Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems, vol. 9, no. 1, p. 21–65, Feb. 1991, DOI: 10.1145/103727.103729
10. **Kim Y.-J., Anderson J. H.** A space- and time-efficient local-spin spin lock, Information Processing Letters, vol. 84, no. 1, p. 47–55, Oct. 2002, DOI: 10.1016/s0020-0190(02)00224-7
11. **Lamport L.** A fast mutual exclusion algorithm, ACM Transactions on Computer Systems, vol. 5, no. 1, p. 1–11, Jan. 1987, DOI: 10.1145/7351.7352
12. **Afek Y., Attiya H., Fouren A., Stupp G., Touitou D.** Long-lived renaming made adaptive, Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, May 1999, DOI: 10.1145/301308.301335
13. **Afek Y., Merritt M.** Fast, wait-free (2k-1)-renaming, Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, May 1999, DOI: 10.1145/301308.301338
14. **Attiya H., Fouren, A.** Adaptive long-lived renaming with read and write operations (No. CS Technion report CS0956), Computer Science Department, Technion, 1999.
15. **Michael M. M., Scott M. L.** Fast Mutual Exclusion, Even with Contention, Jun. 1993, DOI: 10.21236/ada272947
16. **Styer E.** Improving fast mutual exclusion, Proceedings of the eleventh annual ACM symposium on Principles of distributed computing - PODC ’92, 1992, DOI: 10.1145/135419.135453
17. **Choy M., Singh A. K.** Adaptive solutions to the mutual exclusion problem, Distributed Computing, vol. 8, no. 1, p. 1–17, Aug. 1994, DOI: 10.1007/bf02283567.
18. **Alur R., Attiya H., Taubenfeld G.** Time-adaptive algorithms for synchronization, Proceedings of the twenty-sixth annual ACM symposium on Theory of computing - STOC ’94, 1994, DOI: 10.1145/195058.195464
19. **Kuhn F., Maus Y., Weidner S.** Deterministic Distributed Ruling Sets of Line Graphs, Lecture Notes in Computer Science, p. 193–208, 2018, DOI: 10.1007/978-3-030-01325-7_19
20. **Lamport L.** The mutual exclusion problem: part I - A theory of interprocess communication, Journal of the ACM, vol. 33, no. 2, p. 313–326, Apr. 1986, DOI: 10.1145/5383.5384
21. **Lamport L.** The mutual exclusion problem: Part II - Statement and solutions, Journal of the ACM, vol. 33, no. 2, p. 327–348, Apr. 1986, DOI: 10.1145/5383.5385
22. **Schneider J., Elkin M., Wattenhofer R.** Symmetry breaking depending on the chromatic number or the neighborhood growth, Theoretical Computer Science, vol. 509, p. 40–50, Oct. 2013, DOI: 10.1016/j.tcs.2012.09.004

23. **Daymude J. J., Richa A. W., Scheideler C.** Local mutual exclusion for dynamic, anonymous, bounded memory message passing systems, arXiv.org, 24-Feb-2022. [Online]. URL: https://arxiv.org/abs/2111.09449 (accessed on: 19.02.2023)
24. **Fraser K., Harris T.** Concurrent programming without locks, ACM Transactions on Computer Systems, vol. 25, no. 2, p. 5, 2007

References

1. **John K. H., Tiegelkamp M.** Programming Industrial Automation Systems // IEC 61131-3. 2010. DOI: 10.1007/978-3-642-12015-2
2. **Zyubin V. E.** Reflex Language. Mathematical model of control algorithms // Sensors and Systems. 2006. Vol. 5. Pp. 24–30. (in Russ.)
3. **Zyubin V. E.** Static balancing of computing resources in process-oriented programming // Vestnik NSU. Series: Information Technologies. 2012. Vol. 10, no. 2. Pp. 44–54. (in Russ.)
4. **Zyubin V. E. et al.** Basic module controlling the installation for growing single crystals of silicon // Sensors and Systems. 2004. Vol. 12. Pp. 17–22. (in Russ.)
5. **Bulavskiy D. V. et al.** Automated control system for a facility for growing single crystals of silicon // Autometric. 1996. Vol. 2. P. 26. (in Russ.)
6. **Dijkstra E. W.** Solution of a Problem in Concurrent Programming Control // Pioneers and Their Contributions to Software Engineering. 2001. Pp. 289–294. DOI 10.1007/978-3-642-48354-7_10
7. **Knuth D. E.** Additional comments on a problem in concurrent programming control // Communications of the ACM. 1966. Vol. 9, no. 5. Pp. 321–322. DOI 10.1145/355592.365595
8. **Anderson T. E.** The performance of spin lock alternatives for shared-money multiprocessors // IEEE Transactions on Parallel and Distributed Systems. 1990. Vol. 1, no. 1. Pp. 6–16. DOI 10.1109/71.80120
9. **Mellor-Crummey J. M., Scott M. L.** Algorithms for scalable synchronization on shared-memory multiprocessors // ACM Transactions on Computer Systems. 1991. Vol. 9, no. 1. Pp. 21–65. DOI 10.1145/103727.103729
10. **Kim Y.-J., Anderson J. H.** A space- and time-efficient local-spin spin lock // Information Processing Letters. 2002. Vol. 84, no. 1. Pp. 47–55. DOI 10.1016/s0020-0190(02)00224-7
11. **Lamport L.** A fast mutual exclusion algorithm // ACM Transactions on Computer Systems. 1987. Vol. 5, no. 1. Pp. 1–11. DOI 10.1145/7351.7352
12. **Afek Y., Attiya H., Fouren A., Stupp G., Touitou D.** Long-lived renaming made adaptive // Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing. 1999. DOI 10.1145/301308.301335
13. **Afek Y., Merritt M.** Fast, wait-free (2k-1)-renaming // Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing. 1999. DOI 10.1145/301308.301338
14. **Attiya H., Fouren A.** Adaptive long-lived renaming with read and write operations (No. CS Technion report CS0956). Computer Science Department, Technion, 1999.
15. **Michael M. M., Scott M. L.** Fast Mutual Exclusion // Even with Contention. 1993. DOI 10.21236/ada272947
16. **Styer E.** Improving fast mutual exclusion, Proceedings of the eleventh annual ACM symposium on Principles of distributed computing. PODC'92, 1992. DOI 10.1145/135419.135453
17. **Choy M., Singh A. K.** Adaptive solutions to the mutual exclusion problem // Distributed Computing. 1994. Vol. 8, no. 1. Pp. 1–17. DOI 10.1007/bf02283567
18. **Alur R., Attiya H., Taubenfeld G.** Time-adaptive algorithms for synchronization // Proceedings of the twenty-sixth annual ACM symposium on Theory of computing. STOC'94, 1994. DOI 10.1145/195058.195464

19. **Kuhn F., Maus Y., Weidner S.** Deterministic Distributed Ruling Sets of Line Graphs // Lecture Notes in Computer Science. 2018. Pp. 193–208. DOI 10.1007/978-3-030-01325-7_19
20. **Lamport L.** The mutual exclusion problem: part I – A theory of interprocess communication // Journal of the ACM. 1986. Vol. 33, no. 2. Pp. 313–326. DOI 10.1145/5383.5384
21. **Lamport L.** The mutual exclusion problem: Part II – Statement and solutions // Journal of the ACM. 1986. Vol. 33, no. 2. Pp. 327–348. DOI 10.1145/5383.5385
22. **Schneider J., Elkin M., Wattenhofer R.** Symmetry breaking depending on the chromatic number or the neighborhood growth // Theoretical Computer Science. 2013. Vol. 509. P. 40–50. DOI 10.1016/j.tcs.2012.09.004
23. **Daymude J. J., Richa A. W., Scheideler C.** Local mutual exclusion for dynamic, anonymous, bounded memory message passing systems [Online]. URL: https://arxiv.org/abs/2111.09449 (accessed on: 19.02.2023).
24. **Fraser K., Harris T.** Concurrent programming without locks // ACM Transactions on Computer Systems. 2007. Vol. 25, no. 2. P. 5.

Информация об авторе

Пермяшкин Дмитрий Андреевич, старший инженер ООО «Техкомпания Хуавэй»

Information about the Author

Dmitry A. Permyashkin, Senior Engineer of Huawei Tech Company LLC, Moscow, Russian Federation

Статья поступила в редакцию 20.03.2023;
одобрена после рецензирования 01.06.2023; принята к публикации 01.06.2023

The article was submitted 20.03.2023;
approved after reviewing 01.06.2023; accepted for publication 01.06.2023