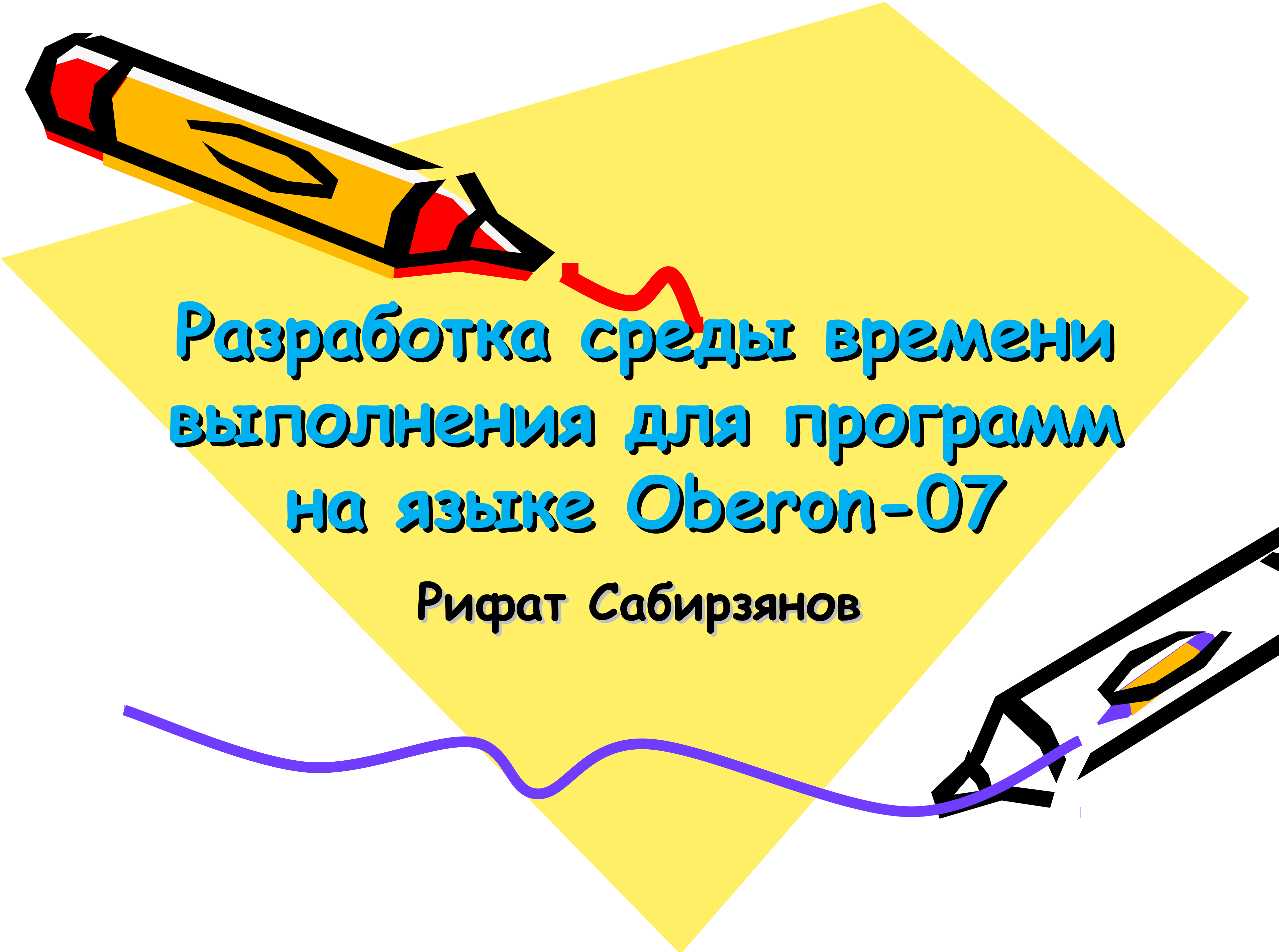
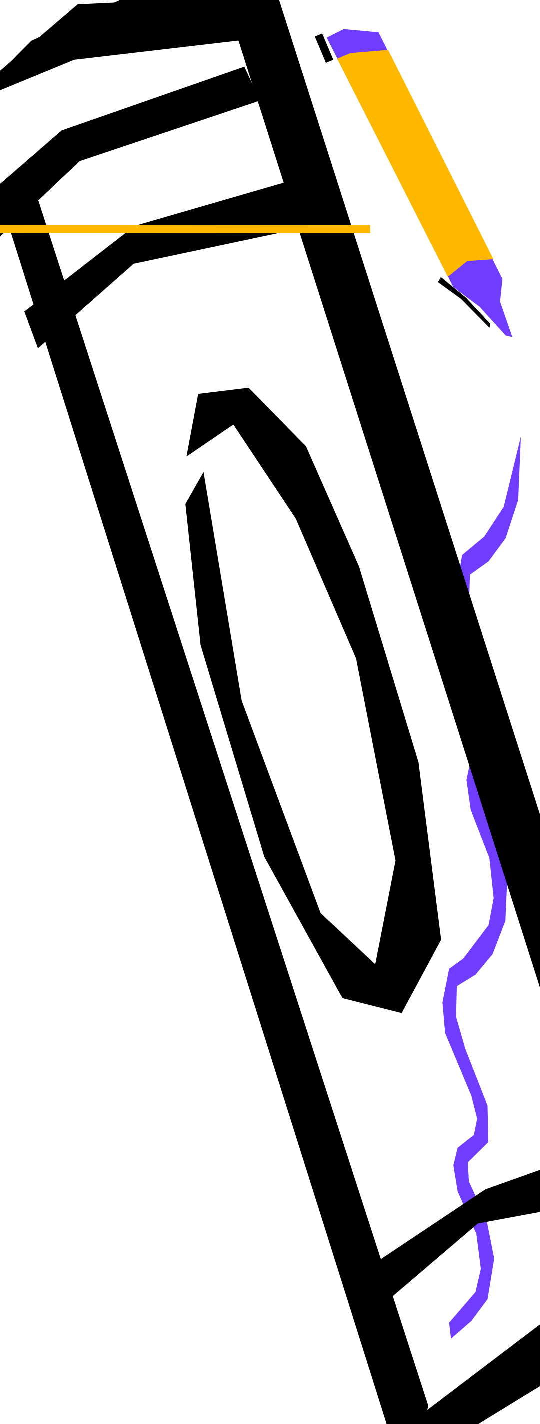
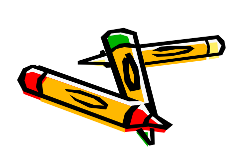

Разработка среды времени выполнения для программ на языке Oberon-07
Рифат Сабирзянов

План

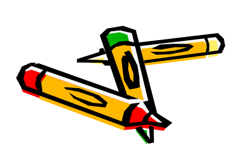
- Модель параллельности
- Детали реализации
- Про другое





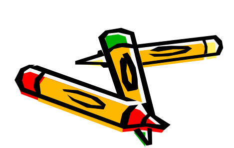
История

- Начал изучать формальную верификацию программ и задумался над тем, а как же верифицировать параллельные программы.
- Оказалось, что нет каких-то простых методов верификации для моделей параллельности с общей памятью.



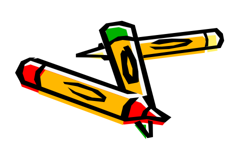
История

- Для одного подкласса моделей с посылкой сообщений есть довольно простой метод формальной верификации.



Модель параллельности

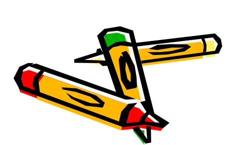
- В отличие от CSP где сообщения могут посылаться или приниматься в любом месте процесса, этот класс параллельных программ представляет собой цикл с инициализацией и посылка или приём сообщений могут осуществляться только в начале цикла.



Модель параллельности

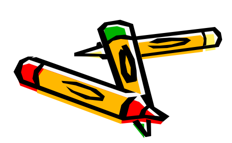
```
SENDER = i := 0;
do i # M;link!a[i] -> i := i + 1 od

RECEIVER = j := 0;
do j # M;link?b[j] -> j := j + 1 od
```



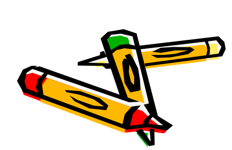
Модель параллельности

```
i := 0; j := 0;
WHILE (i # M) & (j # M) DO
  link := a[i];
  b[j] := link;
  i := i + 1;
  j := j + 1;
END;
```



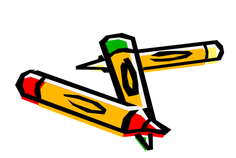
Как описывать программу?

```
SENDER = i := 0;
do i # M;link!a[i] -> i := i + 1 od
```



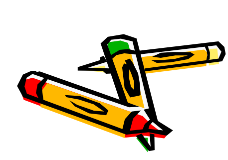
Как описывать программу?

```
TYPE
  SENDER = RECORD
    guardedCommands: List of guarded commands;
    i: INTEGER;
    a: ARRAY Max OF CHAR
  END;
```



Как описывать программу?

```
PROCEDURE Init(sender: SENDER)
BEGIN
  sender.i := 0;
END Init;
```

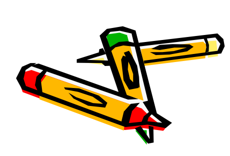


Как описывать программу?

```
PROCEDURE Guard(sender: SENDER): BOOLEAN;
BEGIN
  RETURN sender.i # sender.M
END Guard;
PROCEDURE SendMessage(sender: SENDER; link: MESSAGE);
BEGIN
  link := sender.a[sender.i];
END SendMessage;
```

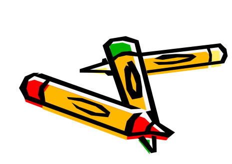
Как описывать программу?

```
PROCEDURE Command(sender: SENDER);
BEGIN
  sender.i := sender.i + 1;
END Command;
```



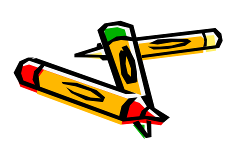
Как описывать программу?

```
PROGRAM = RECORD
  components: List of components
END
```



Алгоритм

- Выполнить Init во всех компонентах и вычислить Guard всех охраняемых команд.
- Пока есть сообщения для пересылки.
- Сбросить Guard всех условий у компонентов, которым принадлежат сообщения.
- Переслать сообщение.
- Выполнять соответствующие команды в параллельном режиме.
- После выполнения команды пересчитать Guard условий в компоненте.

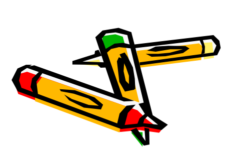


Как определять какие посылаемые и принимаемые сообщения соответствуют друг другу?

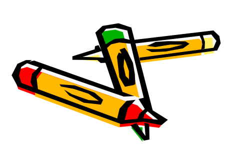
```
TYPE
  MyMessage =
POINTER TO   RECORD
    a: CHAR
  END;
  OtherMessage =
POINTER TO   RECORD
    a: CHAR
  END;
```

Как принимать сообщение от определенного компонента?

```
PROCEDURE
Pair(sender: SENDER): INTEGER;
BEGIN
  RETURN -1
END Pair;
```

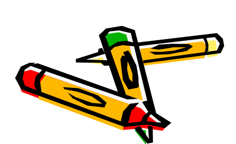


Могут ли быть вложенные параллельные программы?



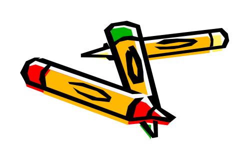
Могут ли быть вложенные параллельные программы?

- По сути наша параллельная программа представляет собой просто цикл.
- А если брать аналогию с обычными циклами, где бывают вложенные циклы, то имеет смысл разрешить вложенные параллельные программы.



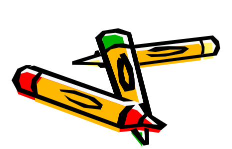
Потоки и операционная система

- В Windows некоторые WinApi функции требуют, чтобы они были выполнены в определенном потоке.
- Поэтому необходимо еще немного расширить нашу модель и для каждого компонента указывать должен ли он выполняться в определенном потоке.



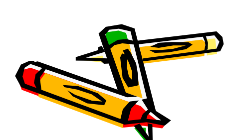
Потоки и операционная система

- Возможно, это является одной из причин почему распространено распараллеливание с помощью потоков и общей памяти.



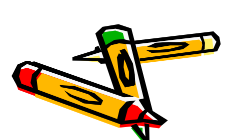
Как быть с выделением памяти

- Один из вариантов — это выделять память в отдельном компоненте.
- Другой вариант — останавливать все потоки на время сборки мусора.



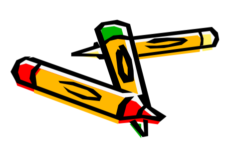
Какие бывают сбрщики мусора?

- Сборка мусора с подсчетом ссылок.
- Параллельный сборщик мусора.
- Старый добрый Mark and Sweep.



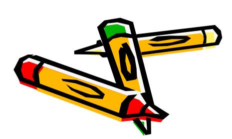
Решение

- У каждого компонента должна быть своя память и свой сборщик мусора.
- Данное решение также позволит уменьшить задержки, связанные со сборкой мусора, при большом количестве памяти, которое становится доступным в 64 битном режиме.



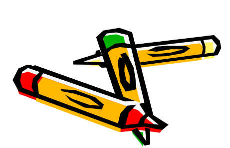
Решение

- Что должно произойти с памятью компонента при завершении параллельной программы?
- Выделенная память должна остаться или же она должна быть уничтожена?



Решение

- Память компонентов сливается в общую память после завершения параллельной части.

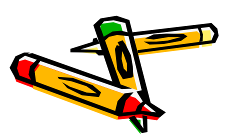


Детали работы планировщика

- Возможна ситуация, что у компонента несколько условий одновременно истинно. И какое же сообщение должно быть послано в этом случае?
- Возможна ситуация, что какое-то сообщение будет постоянно пропускаться.
- Это называется проблемой честности (fairness) планировщика.

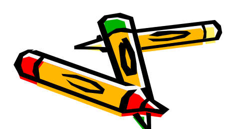
Детали работы планировщика

- Теория говорит, что любые подходы будут иметь недостатки, кроме очереди из всех возможных пар сообщений.
- То есть, если есть 1000 компонентов, которые посылают 1 сообщение, и 1000 компонентов, которые принимают 1 сообщение, то всего будет 1000000 комбинаций.



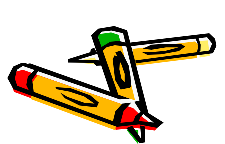
Детали работы планировщика

- Проход по списку будет работать за O(N)
- Можно реализовать «быструю очередь» на основе взвешенного сбалансированного дерева, тогда поиск будет работать за O(log(N)), правда для обновления дерева при пересчете Guard нужно будет сделать M*O(log(N)), где M — количество связей у сообщения.



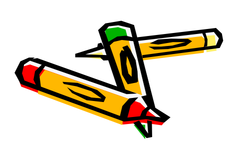
Детали работы планировщика

- При наличии вложенных параллельных программ возникает вопрос: как распределять вычислительные ресурсы между ними.



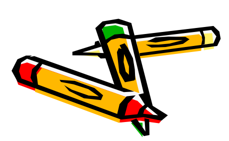
Детали работы планировщика

- Решение состоит в том, чтобы также создать очередь для параллельных программ.
- В свою очередь лучше использовать «быструю очередь», если возможно большое количество параллельных программ внутри программы.



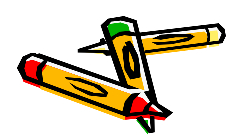
Детали работы планировщика

- Внутри планировщика необходимо создать критическую секцию, чтобы только один поток в каждый момент времени мог обновлять очереди.



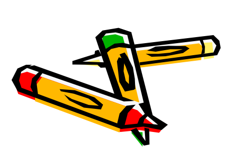
Детали работы планировщика

- Есть алгоритм Деккера для взаимоисключения двух процессов.
- Обобщение алгоритма Деккера для случая N процессов из книги «Структурное программирование» не обладает свойством честности. Быстрый поток может постоянно захватывать критическую секцию.



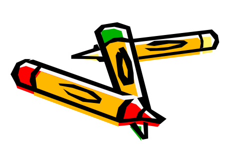
Детали работы планировщика

- Критические секции из операционной системы также могут не обладать свойством честности.



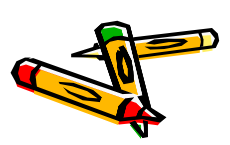
Детали работы планировщика

- Решение: реализовать взаимоисключение для N потоков на основе алгоритма Деккера для двух процессов путем создания дерева (типа пирамиды или турнирной таблицы).



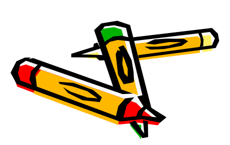
Возможные применения параллельных программ на основе посылки сообщений

- Любые программы, где может потребоваться добавить буферизацию. Например, если есть компилятор, который состоит из лексера и синтаксического анализатора, то можно вставить буферизацию для лексем.



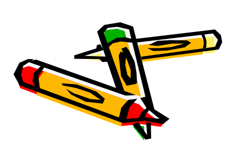
Возможные применения параллельных программ на основе посылки сообщений

- Любые программы, где может потребоваться добавить буферизацию. Например, если есть компилятор, который состоит из лексера и синтаксического анализатора, то можно вставить буферизацию для лексем.



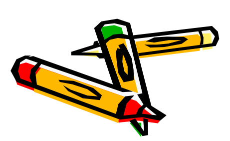
Возможные применения параллельных программ на основе посылки сообщений

- Если требуется транзакционная обработка данных, то можно обойтись без базы данных, а данные будет обрабатывать один компонент. Соответственно, другие компоненты не смогут одновременно изменять или считывать данные и транзакционность будет обеспечиваться.



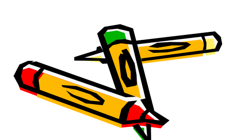
Выводы

- При использовании параллельности на основе посылки сообщений программа остаётся интеллектуально управляемой и можно разобраться с любой проблемой на основе исходного текста программы и логических рассуждений.



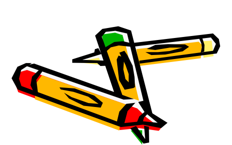
Про другое

- Думал над тем как записывать предусловия, постусловия и инварианты в программе, с целью дальней обработки их утилитой, которая будет их автоматически доказывать.



Про другое

- А также современные и IDE показывают описание функции или параметров. Но для этого надо как-то сообщить IDE эти данные.



Про другое

- Думал сначала использовать XML для описания параметров и предусловий:

```
<parameter name=''field''>
Some text
</parameter>
<precondition>
a < b
</precondition>
```

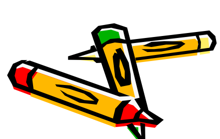
Про другое

- Думал сначала использовать XML для описания параметров и предусловий:

```
<parameter name=''field''>
Some text
</parameter>
<precondition>
a < b
</precondition>
```

Про другое

```
(*****************************************************************************************************
[[SUMMARY]]Преобразовывает дерево в соответствие с рисунком:                                  [[END]]
[[SUMMARY]]      B           A                                                                [[END]]
[[SUMMARY]]  A     -       -     B                                                            [[END]]
[[SUMMARY]]-   -   -  ->   -   -   -                                                          [[END]]
[[SUMMARY]]-   -           -   -   -                                                          [[END]]
[[SUMMARY]]-                                                                                  [[END]]
[[PARAMETER "f"]]Запись с функциями и процедурами для работы с вершинами.                     [[END]]
[[PARAMETER "node"]]Корневая вершина поддерева, которая преобразовывается.                    [[END]]
[[PARAMETER "tempA"]]Временная вершина.                                                       [[END]]
[[PARAMETER "tempB"]]Временная вершина.                                                       [[END]]
[[PARAMETER "tempC"]]Временная вершина.                                                       [[END]]
[[PRECONDITION]]Дерево имеет вид, как на рисунке до преобразования.                           [[END]]
{{PRECONDITION}}~f.IsAddrNull(node)                                                           {{END}}
*****************************************************************************************************)
PROCEDURE RotateRight(VAR f: Functions; VAR node, tempA, tempB, tempC: Object.Address);
```



Вопросы