Министерство образования и науки Украины
Севастопольский государственный технический университет

Методические указания

к лабораторной работе №2 по курсу
“Параллельные и распределенные вычисления” на тему
«Решение задачи «производитель-потребитель»
для студентов направления
“Компьютерная инженерия ”

Севастополь
2009

Цель работы:

Исследовать возможности, предоставляемые системой программирования JAVA для синхронизации взаимодействующих параллельных процессов.

Задание на лабораторную работу

1. Разработать в соответствии с вариантом задания программу, создающую два подпроцесса, осуществляющих обмен данными
2. Исследовать поведение программы при использовании различных методов синхронизации
3. Проанализировать полученные результаты.

Методика выполнения работы

Задача “производитель-потребитель” относится к базовым задачам синхронизации. В общем (простейшем) случае взаимодействуют два процесса, один из которых (производитель) вырабатывает некоторые сообщения, а другой (потребитель) эти сообщения принимает. Обмен сообщениями осуществляется через общую область данных, являющуюся критическим ресурсом. Следовательно, для корректного решения задачи “производитель-потребитель” необходимо, во-первых, обеспечить режим взаимного исключения по отношению к общей области; во-вторых, обеспечить строгую очередность работы процессов. Потребитель имеет право считывать сообщение только при условии, что производитель уже поместил это сообщение в общую область. Далее, производитель имеет право записывать очередное сообщение только в том случае, если предыдущее сообщение уже считано потребителем.

Рассмотрим пример программы, реализующий простейший вариант задачи “производитель-потребитель” (прототип взят из книги А.В.Картузова “Программирование на языке JAVA”).

Основной поток создает два подпроцесса, которые будут выполняться в течение 5-ти секунд. Первый подпроцесс (Producer) в своем методе run записывает новое значение в общую переменную n; второй подпроцесс (Consumer) – выбирает очередное значение n.

Результаты выполнения программы очевидно неправильны. Поскольку диспетчеризация подпроцессов выполняется операционной системой, логично ожидать, что последовательность обмена сообщениями непредсказуема. Практическая ценность приведенного примера – значение количества итераций, выполненных подпроцессами.

```
class PC
{
  public static void main(String args[])
  {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    try { Thread.sleep(5000); } catch(InterruptedException e) {} ;
    q.work = false;
  }
}
class Q
{
  int n;
  boolean work = true;
  int get()
  {
    System.out.println("Got: " + n);  return n;
  }
  void put(int n)
  {
    this.n = n; System.out.println("Put: " + n);
  }
}
class Producer implements Runnable
{
  Q q;
  Producer(Q q)
  {
    this.q = q; new Thread(this, "Producer").start();
  }
  public void run()
  {
    int i = 0;  while (q.work) { q.put(i++); }
  }
}
class Consumer implements Runnable
{
  Q q;
  Consumer(Q q)
  {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run()
  {
    while (q.work) { q.get(); }
  }
}
```

Введем в программу управляющую переменную, характеризирующую состояние общего ресурса (есть очередное сообщение).

```
class Q
{
  int n;
  boolean valueSet = false;
  boolean work = true;
  int get()
  {
    while (!valueSet) {};
    System.out.println("Got: " + n);
    valueSet = false;
    return n;
  }
  void put(int n)
  {
    while (valueSet) {};
    this.n = n;
    System.out.println("Put: " + n);
    valueSet = true;
  }
}
```

Проанализируем полученные результаты. Очевидно, применение пустого цикла (активное ожидание) приводит к непроизводительным затратам.
Добавим явное освобождение процессора в метод run.

```
class Q
{
  int n;
  boolean valueSet = false;
  boolean work = true;
  int get()
  {
    while (!valueSet) Thread.yield();
    System.out.println("Got: " + n);
    valueSet = false;
    return n;
  }
  void put(int n)
  {
    while (valueSet) Thread.yield();
    this.n = n;
    System.out.println("Put: " + n);
    valueSet = true;
  }
}
```

Проанализируем результаты.

Теперь перейдем к использованию средств синхронизации, предоставляемых Java. Каждый объект в Java можно использовать в режиме взаимного исключения (заблокировать). Следует отметить, что блокировка не означает запрет обращения к объекту, система не позволяет другому подпроцессу повторно заблокировать уже заблокированный объект.
Для того, чтобы воспользоваться механизмом блокировок, используется ключевое слово synchronized. Для блокировки метода это ключевое слово указывается как модификатор в заголовке. Для блокировки полей и методов конкретного объекта используется следующая конструкция:

```
synchronized (object) {
  ...
}
```

Кроме этого, каждому объекту можно сопоставить свою очередь ожидания. Подпроцесс, вызвавший метод wait() любого объекта попадает в эту очередь и будет активизирован, если другой подпроцесс вызовет метод этого же объекта notifyAll() (освободить все подпроцессы) или, если повезет, Notify() (освободить один подпроцесс).

Применение указанных выше методов требует установки блокировки.

```
class Q
{
 int n;
 boolean valueSet = false;
 boolean work = true;
 synchronized int get()
 {
  if (!valueSet) try { wait(); }
  catch(InterruptedException e) {} ;
  System.out.println("Got: " + n);
  valueSet = false;
  notify();
  return n;
 }
 synchronized void put(int n)
 {
  if (valueSet) try { wait(); }
  catch(InterruptedException e) {} ;
  this.n = n;
  System.out.println("Put: " + n);
  valueSet = true;
  notify();
 }
}
```

Проанализируем результаты.

И, в заключение, воспользуемся классическим механизмом синхронизации – семафорами. Соответствующий класс в Java описан в пакете java.util.concurrent. Задействуем два семафора с именами emp и ful. Первый, если он открыт, сигнализирует, что общая область пуста. Второй, соответственно, означает, что в общей области есть данные. Таким образом, первый семафор регулирует подпроцесс-производитель, второй – подпроцесс-потребитель. Изначально семафор emp закрыт, а ful – открыт.
Классическим примитивам P(S) и V(S) соответствуют методы acquire() и release(). Начальное значение семафора задается параметром конструктора.

```
class Q
{
  int n;
  boolean work = true;
  Semaphore emp = new Semaphore(1,true);
  Semaphore ful = new Semaphore(0,true);
  int get()
  {
    try { ful.acquire(); }
    catch(InterruptedException e) {} ;
    System.out.println("Got: " + n);
    emp.release();
    return n;
  }
   void put(int n)
  {
    try { emp.acquire(); }
    catch(InterruptedException e) {} ;
    this.n = n;
    System.out.println("Put: " + n);
    ful.release();
  }
}
```

Варианты заданий

№	Тип первого процесса	Тип второго процесса
1	1	5
2	1	6
3	1	7
4	1	8
5	2	5
6	2	6
7	2	7
8	2	8
9	3	5
10	3	6
11	3	7
12	3	8
13	4	5
14	4	6
15	4	7
16	4	8

Типы процессов:

1. Генерация ряда натуральных чисел
2. Генерация последовательности случайных чисел
3. Генерация ряда квадратов натуральных чисел
4. Генерация чисел Фибоначчи
5. Вычисление среднего арифметического значения элементов последовательности
6. Вычисление суммы элементов последовательности
7. Нормализация (округление до заданного порога) элементов последовательности
8. Выбор максимального значения из пар соседних элементов последовательности

Содержание отчета

1. Постановка задачи
2. Текст программы с комментариями
3. Протоколы испытаний
4. Выводы

Контрольные вопросы

1. Средства решения задачи взаимного исключения
2. Средства решения задачи «производитель-потребитель»