8. Программирование для систем с общей памятью. Основные понятия: процесс, поток, ресурс. Взаимоисключение потоков, критическая секция. Программная реализация взаимоисключения. Проблемы, связанные с взаимоисключением.

**Программирование для систем с общей памятью**

1. Ветвление процессов

Современные операционные системы базируются на понятии процесс. **Процесс** - это некоторая последовательность команд, имеющая собственный контекст и уникальный идентификатор. Под контекстом понимается набор атрибутов процесса - собственный стек исполнения; набор страниц памяти, составляющих адресное пространство процесса; таблицу дескрипторов файлов и т.д. [9].

**Понятие ресурса** обычно используется для обозначения любых объектов вычислительной системы, которые могут быть использованы процессом для своего выполнения. В качестве ресурса может рассматриваться процесс, память, программы, данные и т. п.

Именно, то обстоятельство, что система функционирует как набор процессов, позволяет организовать многопользовательский, многозадачный режим работы даже на обычных однопроцессорных системах. В этом случае, ресурсы единственного процессора поочередно предоставляются каждому процессу. Поскольку квант времени, предоставляемый процессу значительно меньше, чем время реакции человека, то создается иллюзия одновременного выполнения множества процессов (задач). На многопроцессорных системах с общей памятью выполнение разных процессов может происходить одновременно на разных процессорах, и, таким образом, осуществляется реальное параллельное выполнение процессов. Этот механизм может быть использован для организации параллельного выполнения вычислений.

Поскольку основная часть вычислительной работы в программах сосредоточена внутри циклов (тело цикла выполняется много раз и поэтому суммарное время его выполнения составляет основную часть времени выполнения программы), то распараллеливание чаще всего сводится к распараллеливанию выполнения циклов. В нашей задаче мы можем запустить два процесса для выполнения вычислительного цикла, используя для этого системные функции fork(), wait() and exit(). Функция fork() создает новую копию выполняющегося процесса. Новый процесс называется дочерним, а первичный процесс родительским. Значение, возвращаемое функцией fork, позволяет различить родительский и дочерний процессы: родительский процесс получает идентификатор дочернего процесса, а дочерний получает нуль. Используя этот механизм можно поручить различную работу родительскому и дочернему процессам. Дочерний процесс завершается вызовом функции exit(), а родительский ожидает завершения дочернего процесса функцией wait(). Этот механизм ветвления процессов является базовым механизмом в unix-подобных операционных системах. Применительно к параллельным вычисления эта процедура может быть повторена n-1 раз, а планировщик операционной системы автоматически распределит процессы по n процессорам.

Поскольку все процессы независимы и имеют ограниченный набор общих ресурсов, то в программе необходимо предусмотреть операции обмена информацией между процессами. Для этой цели можно использовать функции для работы с общей памятью: shmget (), shmat () и shmctl (). Функция shmget () резервирует сегмент общей памяти, функция shmat() связывает этот сегмент с процессом, а функция shmctl () устанавливает атрибуты для этого сегмента. Когда порождается дочерний процесс, он наследует этот сегмент общей памяти, и таким образом оба процесса получают к нему доступ.

В программе pi\_fork.с представлен вариант программы с двумя процессами: родительский процесс выполняет четные шаги цикла i = 2, 4, 6, ..., в то время как дочерний процесс выполняет нечетные шаги i = 1, 3, 5, .... Дочерний процесс хранит свой результат по адресу \*shared, а родительский процесс получает значение этой переменной по окончании работы дочернего процесса и прибавляет его к своему значению

Программа pi\_fork.с

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

main(int argc, char \*\*argv)

{

int n, i;

double d, s, x, pi;

int shmid, iproc;

pid\_t pid;

double \*shared;

n = atoi(argv[1]);

d = 1.0/n;

shmid = shmget(IPC\_PRIVATE,

sizeof(double), (IPC\_CREAT | 0600));

shared = shmat(shmid, 0, 0);

shmctl(shmid, IPC\_RMID, 0);

iproc = 0;

if ((pid = fork()) == -1) {

fprintf(stderr, "The fork failed!\n");

exit(0);

} else {

if (pid != 0) iproc = 1 ;

}

s = 0.0;

for (i=iproc+1; i<=n; i+=2) {

x = (i-0.5)\*d;

s += 4.0/(1.0+x\*x);

}

pi = d\*s;

if (pid == 0) {

\*shared = pi;

exit(0);

} else {

wait(0);

pi = pi + \*shared;

printf("pi=%.15f\n", pi);

}

}

Механизм ветвления процессов не является наилучшим механизмом для параллельных вычислений, поскольку процедура дублирования процесса является затратной как с точки зрения расходования ресурсов процессора, так и с точки зрения расходования оперативной памяти, поскольку происходит полное дублирование копии процесса. В параллельном программировании более оправданным является использование потоков или нитей (threads), называемых также легковесными процессами.

Поток - это последовательность выполняемых команд процессора.

2. Многопоточность. Библиотека Pthread

Системная функция fork() порождает полную копию родительского процесса, и эта процедура является затратной как с точки зрения расходования процессорного времени, так и с точки зрения расходования оперативной памяти. В параллельном программировании зачастую достаточно использования частичной копии процесса. Такая частичная копия называется **потоком**, <легковесным процессом> или нитью (thread). Нить - это поток команд, который может самостоятельно управляться планировщиком. В отличие от дочернего процесса нить создается в контексте головного процесса, в том же самом адресном пространстве. На однопроцессорных системах планировщик чередует выполнение нитей посредством квантования процессорного времени, а на многопроцессорных системах нити могут выполняться параллельно на различных процессорах. Потоки не требуют специального механизма взаимодействия между подзадачами, поскольку они имеют равноправный доступ к общей памяти родительского процесса.

Для работы с нитями в разных версиях ОС UNIX существовали свои библиотеки. В настоящее время в качестве стандарта для работы с нитями принята библиотека Pthread [10]. Пользовательский интерфейс библиотеки определен в стандарте POSIX 1003.1 1995, разработанном комитетами по выпуску стандартов ANSI/IEEE. Библиотека Pthread содержит более 60 функций, которые можно разделить на три категории: функции управления потоками, функции управления мьютексами (mutex - взаимное исключение) и функции управления переменными окружения.

В программе pi\_thread.с представлена версия вычисления числа π с использованием библиотеки Pthread. Программа создает нить, используя функцию pthread\_create (), затем назначает уникальный идентификатор переменной типа pthread\_t. Головная программа вызывает вычислительную функцию, которая будет выполняться как нить. Функция pthread\_exit () используется для завершения <легковесного> процесса. Функция pthread\_join () является аналогом функции wait(), но в этом случае любая нить может связываться с любой другой нитью в процессе, так как для нитей нет жесткой связи родитель-потомок.

Поскольку многопоточное приложение выполняет команды одновременно на нескольких процессорах, то доступ к совместно используемой памяти требует синхронизации. Это реализуется с помощью механизма взаимной блокировки (mutex). Этот механизм защищает структуры данных от параллельной модификации. Функции pthread\_mutex\_init (), pthread\_mutex\_lock () и pthread\_mutex\_unlock () используются для этой цели.

Головная программа запускается из командной строки с двумя аргументами: n - числом точек разбиения интервала интегрирования и num\_threads - числом порождаемых нитей. Значение этой глобальной переменной может быть присвоено и за пределами программы в виде переменной окружения оболочки Shell. Первая нить вычислительной функции PIworker, получает свой идентификатор i от головного процесса и выполняет n/num\_threads часть вычислительного цикла. Поскольку доступ к глобальной переменной pi не должны получить больше чем одна нить одновременно, то операция ее модификации запирается механизмом взаимной блокировки (mutex).Рассмотренный алгоритм реализует следующая программа.

Программа pi\_thread.с

#include <stdio.h>

#include <pthread.h>

int n, num\_threads;

double d, pi;

pthread\_mutex\_t reduction\_mutex;

pthread\_t \*tid;

void \*PIworker(void \*arg)

{

int i, myid;

double s, x, mypi;

myid = \*(int \*)arg;

s = 0.0;

for (i=myid+1; i<=n; i+=num\_threads) {

x = (i-0.5)\*d;

s += 4.0/(1.0+x\*x);

}

mypi = d\*s;

pthread\_mutex\_lock(&reduction\_mutex);

pi += mypi;

pthread\_mutex\_unlock(&reduction\_mutex);

pthread\_exit(0);

}

main(int argc, char \*\*argv)

{

int i;

int \*id;

n = atoi(argv[1]);

num\_threads = atoi(argv[2]);

d = 1.0/n;

pi = 0.0;

id = (int \*) calloc(n,sizeof(int));

tid = (pthread\_t \*) calloc(num\_threads,

sizeof(pthread\_t));

if(pthread\_mutex\_init(&reduction\_mutex,NULL)) {

fprintf(stderr, "Cannot init lock\n");

exit(0);

};

for (i=0; i<num\_threads; i++) {

id[i] = i;

if(pthread\_create(&tid[i],NULL,

PIworker,(void \*)&id[i])) {

exit(1);

};

};

for (i=0; i<num\_threads; i++)

pthread\_join(tid[i],NULL);

printf("pi=%.15f\n", pi);

}

Следует отметить, что написание многопоточных программ на языке Си довольно трудоемко и требует чрезвычайной аккуратности. При разработке таких приложений программист обязан обеспечить безопасность параллельно выполняемых функций. На сегодняшний день нет полной гарантии, что все функции системных библиотек безопасны для нитей и обеспечивают полную защиту от несанкционированного доступа в системные области памяти. Поэтому рекомендуется для многопоточного программирования использовать средства более высокого уровня, такие как Java или OpenMP.

3. Многопоточное программирование на языке Java

Язык Java изначально разрабатывался как язык, специально ориентированный на поддержку многопоточности. Java библиотека предоставляет класс Thread, который поддерживает широкий набор методов: например, метод start () вызывает нить для выполнения, метод join () включает ожидание завершения нити. Механизм отпирания-запирания легко реализуется с помощью деклараций синхронизации. Базовые библиотеки системы хорошо защищены от некорректного использования функций в нитях. Эти свойства языка Java делают его более подходящим для многопоточного программирования. Приведем текст нашей программы, написанной на языке Java.

Программа pi.java

public class PiJavaThread {

int n, numThreads;

double pi = 0.0;

synchronized void addPi(double p) {

pi += p;

}

public PiJavaThread(int nd, int nt) {

n = nd;

numThreads = nt;

Thread threads[] = new Thread[numThreads];

for (int i=0; i<numThreads; i++) {

threads[i] = new Thread(new PIworker(i));

threads[i].start();

}

for (int i=0; i<numThreads; i++) {

try {

threads[i].join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

class PIworker implements Runnable {

int myid;

public PIworker(int id) {

myid = id;

}

public void run() {

double d, s, x;

d = 1.0/n;

s = 0.0;

for (int i=myid+1; i<=n; i+=numThreads) {

x = (i-0.5)\*d;

s += 4.0/(1.0+x\*x);

}

addPi(d\*s);

}

}

public static void main(String[] args) {

PiJavaThread piJavaThread

= new PiJavaThread(Integer.parseInt(args[0]),

Integer.parseInt(args[1]));

System.out.println(" pi = " + piJavaThread.pi);

}

}

Из листинга программы видно, что программирование на Java не намного проще, чем на языке С с использованием библиотеки Pthread. Хотя защита данных упрощается использованием специальной функции синхронизации, тем не менее, программирование на Java не менее трудоемко и вряд ли более удобно для написания больших вычислительных программ.

4. Программирование средствами OpenMP

Прикладной программный интерфейс OpenMP (API OpenMP) предоставляет средства для разработки переносимых масштабируемых параллельных программ для систем с общей памятью. По своей сути OpenMP является высокоуровневой надстройкой над Pthread, призванной облегчить разработку многопоточных приложений. OpenMP реализует идею "инкрементального распараллеливания", когда для многопроцессорной системы не пишется специальная параллельная программа, а в обычную последовательную программу добавляются распараллеливающие OpenMP директивы, которые игнорируются обычным компилятором.

**Взаимоисключением потоков** (mutual exclusion). Главное требование к механизмам разделения ресурсов является гарантированное обеспечение *использования каждого разделяемого ресурса только одним потоком* от момента выделения ресурса этому потоку до момента освобождения ресурса; Командные последовательности потоков, в ходе которых поток использует ресурс на условиях взаимоисключения, называется **критической секцией потока**. С использованием последнего понятия условие взаимоисключения потоков может быть сформулировано как требование нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного потока.

Требование взаимоисключения не является единственным к способам организации критических секций; дополнительный перечень необходимых свойств состоит в следующем:

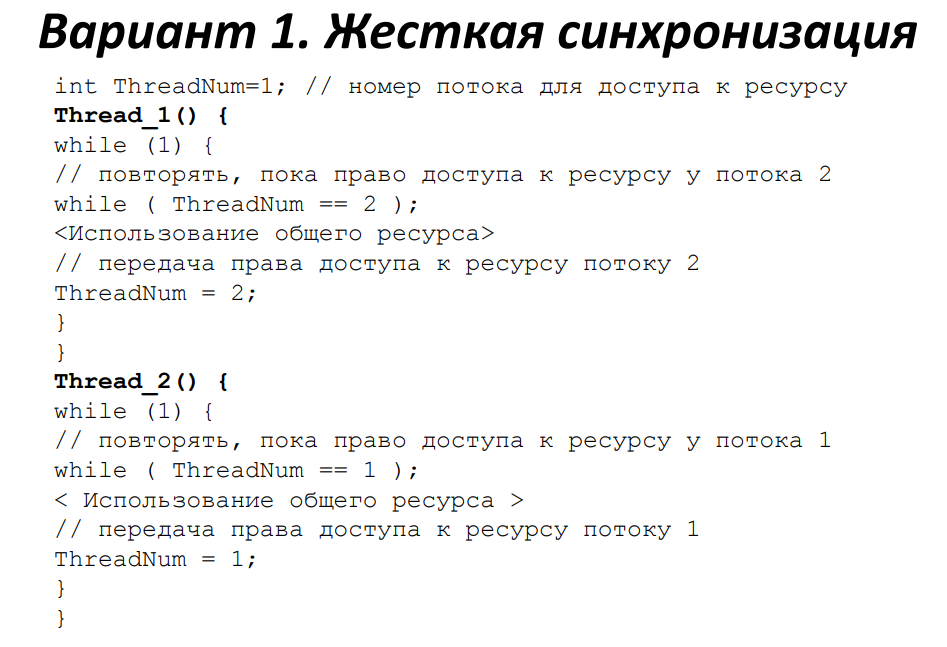
-*Отсутствие взаимной блокировки*. Потоки (по отдельности или совместно) не могут мешать каким-либо потокам обращаться к выполнению своих критических секций.

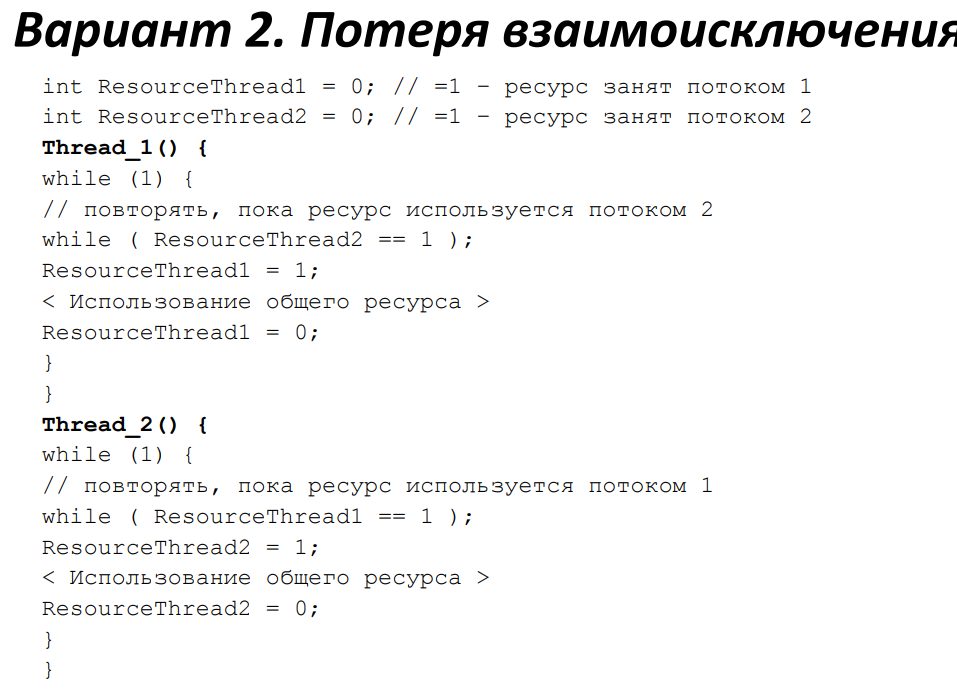
-*Эффективность.* При наличии нескольких потоков, пытающихся начать выполнение своих критических секций, выбор единственного потока для продолжения осуществляется за конечное (малое) время.

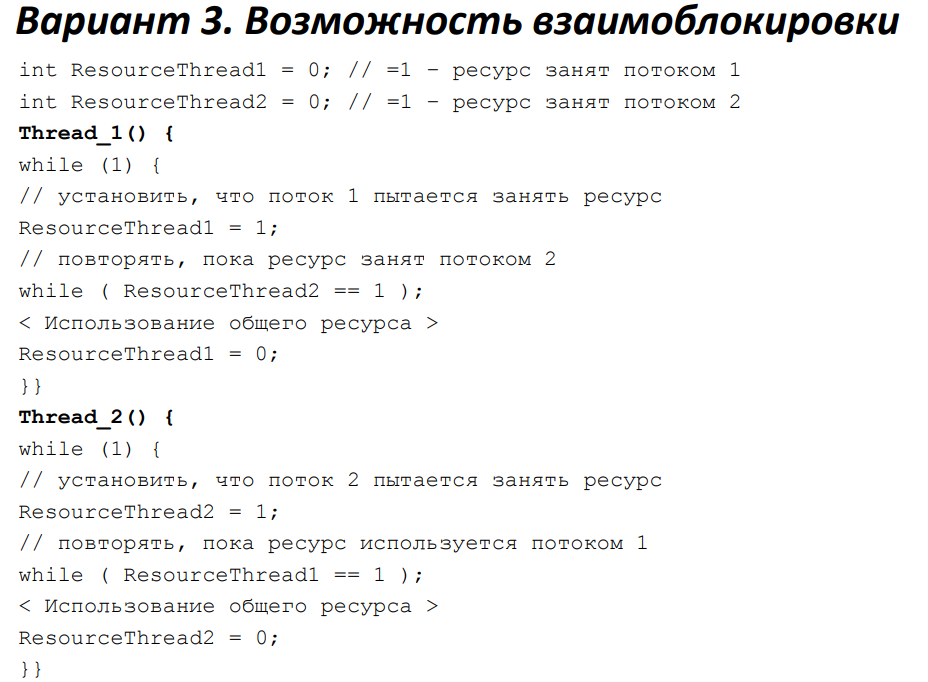
-*Отсутствие бесконечного ожидания.* Поток, пытающийся начать выполнение своей критической секции, гарантированно должен когда-либо получить такую возможность.

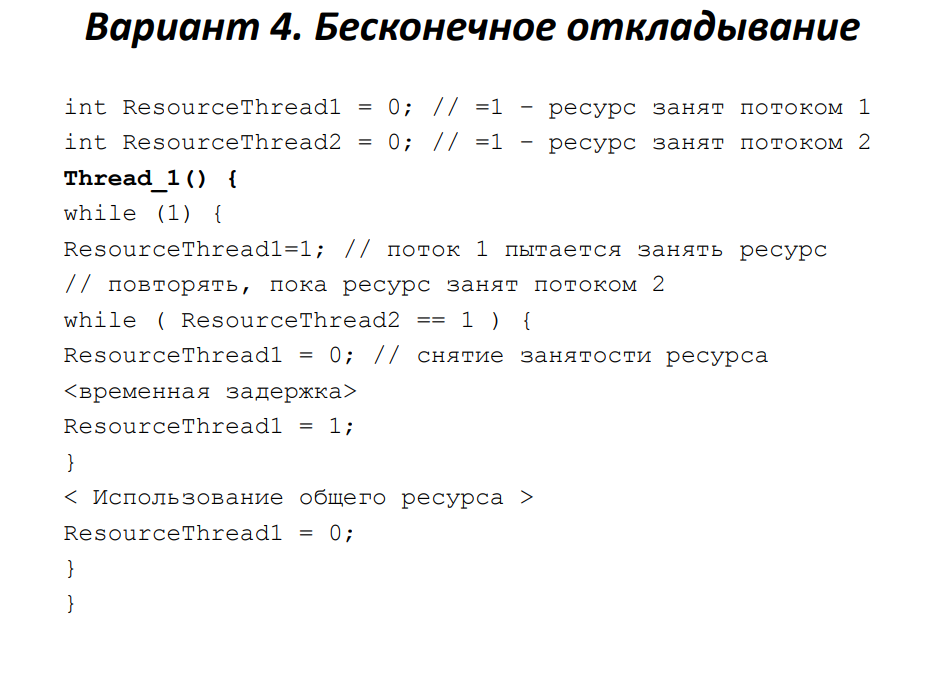
При разработке способов обеспечения критических секций обычно предполагается также, что относительные скорости выполнения потоков неизвестны и произвольны, а длительность нахождения потоков в своих критических секциях является конечной.

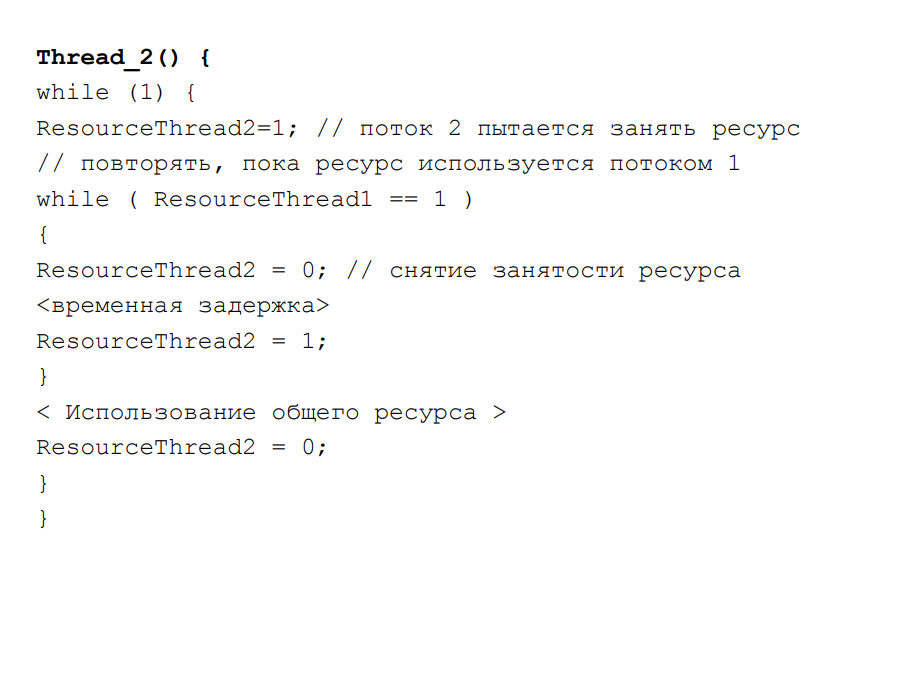
**Программная реализация взаимоисключения.**

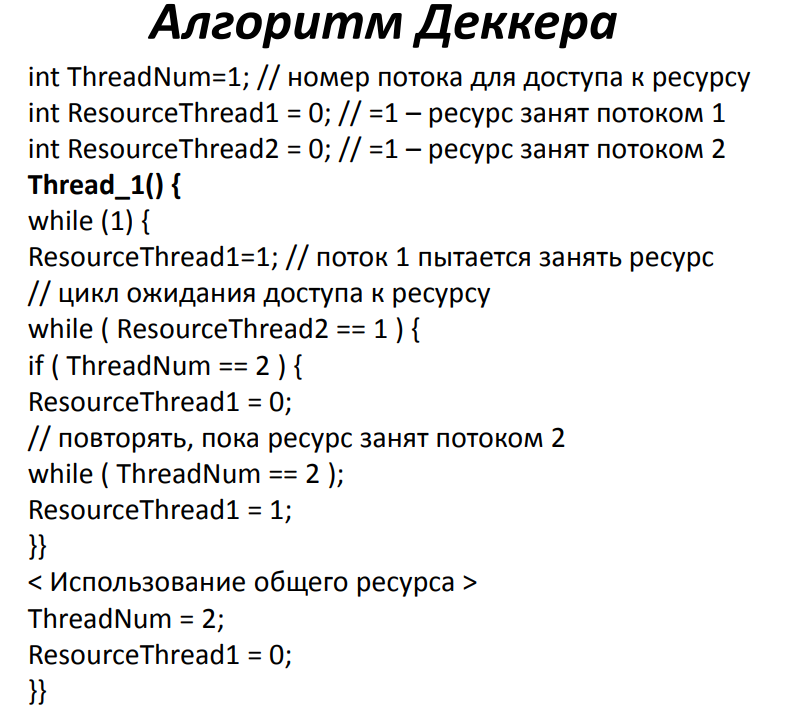


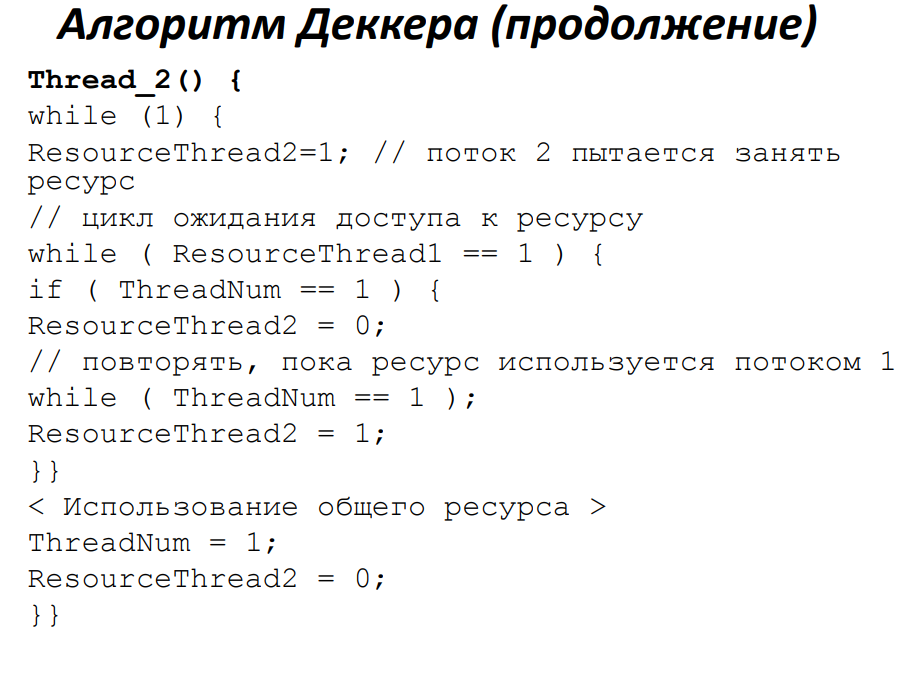












**Проблемы, связанные с взаимоисключением.**

Одной из причин зависимости результатов выполнения программ от порядка чередования команд, может быть, разделение одних и тех же данных между одновременно исполняемыми потоками. Данная ситуация может рассматриваться как проявление общей *проблемы использования разделяемых ресурсов* (общих данных, файлов, устройств и т. п.). Для организации разделения ресурсов между несколькими потоками необходимо иметь возможность:

определения доступности запрашиваемых ресурсов (ресурс свободен и может быть выделен для использования, ресурс уже занят одним из потоков программы и не может использоваться дополнительно каким-либо другим потоком);

выделения свободного ресурса одному из потоков, запросивших ресурс для использования;

приостановки (блокировки) потоков, выдавших запросы на ресурсы, занятые другими потоками.