ЛАБОРАТОРНАЯ РАБОТА №2 Тупики. Предотвращение тупиковых ситуаций. Синхронизация процессов.

1. Теоретические сведения

1.1 Синхронизация процессов

Синхронизация процессов имеет важное значение, при использовании различными процессами одних и тех же ресурсов. Введем некоторые понятия. Под **активностями** мы будем понимать последовательное выполнение ряда операций, направленных на достижение определенной цели. Неделимые операции могут иметь внутренние невидимые действия. Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

```
P:   a b c
Q:   d e f
```

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

```
PQ: a b c d e f
```

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

```
a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
......
d e f a b c
```

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

```
P:  x=2          Q:  x=3
    y=x-1            y=x+1
```

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) **детерминирован**, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает **одинаковые** выходные данные. В противном случае он **недетерминирован**. Выше приведен пример недетерминированного набора программ.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P)

(R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

```
P:   x=u+v
     y=x*w
```

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

1. пересечение W(P) и W(Q) пусто;
2. пересечение W(P) с R(Q) пусто;
3. пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

1.2 Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
   entry section
      critical section
   exit section
      remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Требования, предъявляемые к алгоритмам:

1. задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

1.3 Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции.

Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

```
int Test_and_Set (int *target){
   int tmp = *target;
```

```
   *target = 1;
   return tmp;
}
```

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения

```
shared int lock = 0;

while (some condition) {
   while(Test_and_Set(&lock));
      critical section
   lock = 0;
      remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

Команда Swap (обменять значения)

Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

```
shared int lock = 0;
int key;

while (some condition) {
   key = 1;
   do Swap(&lock,&key);
   while (key);
      critical section
   lock = 0;
      remainder section
}
```

1.4 Механизмы синхронизации потоков

Существуют серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего CPU burst (если не появится процесс с еще большим

приоритетом). Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные механизмы синхронизации более высокого уровня.

Семафоры

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.
Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

```
P(S):   пока S == 0 процесс блокируется;
        S = S - 1;
V(S):   S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.
Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

2. Программные алгоритмы организации взаимодействия процессов.

2.1 Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе "Критическая секция").

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */

while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
```

```
      remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

2.2 Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
   while(turn != i);
      critical section
   turn = 1-i;
      remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

2.3 Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
   ready[i] = 1;
   while(ready[1-i]);
      critical section
   ready[i] = 0;
      remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания

ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

2.4 Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
   ready[i] = 1;
   turn =1-i;
   while(ready[1-i] && turn == 1-i);
      critical section
   ready[i] = 0;
      remainder section
}
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

2.5 Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

```
(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ...., an) – это число k такое, что
k >= ai для всех i = 0, ...,n
```

Структура процесса Pi для алгоритма булочной приведена ниже

```
while (some condition) {
```

```
   choosing[i] = true;
   number[i] = max(number[0], ...,
                   number[n-1]) + 1;
   choosing[i] = false;
   for(j = 0; j < n; j++){
      while(choosing[j]);
      while(number[j] != 0 && (number[j],j) <
         (number[i],i));
   }
      critical section
   number[i] = 0;
      remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

2.6 Реализация многопоточности средствами Win API

Создание процессов

Для создания процессов используются функции CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW и CreateProcessWithTokenW. Функция CreateProcess создает новый процесс, который будет исполняться от имени текущего пользователя потока, вызвавшего эту функцию. Функция CreateProcessAsUser позволяет запустить процесс от имени другого пользователя, который идентифицируется его маркером безопасности (security token); однако вызвавший эту функцию поток должен принять меры к правильному использованию реестра, так как профиль нового пользователя не будет загружен. Функции CreateProcessWithTokenW и CreateProcessWithLogonW позволяют при необходимости загрузить профиль пользователя и, кроме того, функция CreateProcessWithLogonW сама получает маркер пользователя по известному учетному имени, домену и паролю.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,     // имя исполняемого файла
  LPTSTR lpCommandLine,          // командная строка
  LPSECURITY_ATTRIBUTES  lpProcessAttributes,  // атрибуты доступа к процессу
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты доступа к потоку
  BOOL bInheritHandles,   // флаг наследования дескрипторов
  DWORD dwCreationFlags, // флаги создания и флаги класса приоритета
  LPVOID lpEnvironment,   // указатель на параметры настройки окружения
  LPCTSTR lpCurrentDirectory, // путь к текущему каталогу
  LPSTARTUPINFO lpStartupInfo,  // указатель на стуктуру отображения нового процесса
  LPPROCESS_INFORMATION lpProcessInformation // указатель на структуру с информацией о созданном процессе
);

typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;    // дескриптор нового процесса
HANDLE hThread;     // дескриптор первичного потока нового процесса
DWORD dwProcessId; // идентификатор нового процесса
DWORD dwThreadId;   // идентификатор первичного потока нового процесса
```

```
} PROCESS_INFORMATION;

#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

int main( void )
{
  STARTUPINFO         si;
  PROCESS_INFORMATION  pi;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  CreateProcess(
    NULL, "cmd.exe", DEFAULT_SECURITY, DEFAULT_SECURITY,
    FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi
  );
  CloseHandle( pi.hThread );
  WaitForSingleObject( pi.hProcess, INFINITE );
  CloseHandle( pi.hProcess );
  return 0;
}
```

При создании процесса ему можно передать описатели каналов (CreatePipe), предназначенные для перенаправления stdin, stdout и stderr. Описатели каналов должны быть наследуемыми.

Для завершения процесса рекомендуется применять функцию ExitProcess, которая завершит процесс, сделавший этот вызов. В крайних случаях можно использовать функцию TerminateProcess, которая может завершить процесс, заданный его описателем. Этой функцией пользоваться не рекомендуется, так как при таком завершении разделяемые библиотеки будут удалены из адресного пространства уничтожаемого процесса без предварительных уведомлений - это может привести в некоторых случаях к утечке ресурсов.

Завершение процессов

```
VOID ExitProcess( UINT uExitCode);
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Создание потоков

Каждый поток начинает выполнение с некоторой входной функции. В первичном потоке используется одна из функций: WinMain() или main(). Для создания вторичного потока необходимо определить его входную функцию:

```
DWORD WINAPI ThreadFunc ( PVoid pvParam)
{
   DWORD dwResult = 0;
   …
   Return dwResult
}
```

Для создания потока используется функция:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты доступа
  SIZE_T dwStackSize,                       // размер стека
  LPTHREAD_START_ROUTINE lpStartAddress,    // адрес функции потока
  LPVOID lpParameter,                       // параметр функции потока
  DWORD dwCreationFlags,                    // флаги потока
  LPDWORD lpThreadId                        // идентификатор потока
);
```

При необходимости можно создать поток в виртуальном адресном пространстве другого процесса. Для этого используется следующая функция:

```
HANDLE CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Пример:

```
#include <windows.h>
#include <strsafe.h>

#define MAX_THREADS 3
#define BUF_SIZE 255

typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pData;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.

    pData = (PMYDATA)lpParam;
```

```
    // Print the parameter values using thread-safe functions.

    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d,
%d\n"),
        pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);

    // Free the memory allocated by the caller for the thread
    // data structure.

    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}

void main()
{
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;

    // Create MAX_THREADS worker threads.

    for( i=0; i<MAX_THREADS; i++ )
    {
        // Allocate memory for thread data.

        pData = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        if( pData == NULL )
            ExitProcess(2);

        // Generate unique data for each thread.

        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
            NULL,              // default security attributes
            0,                 // use default stack size
            ThreadProc,        // thread function
            pData,             // argument to thread function
            0,                 // use default creation flags
            &dwThreadId[i]);   // returns the thread identifier

        // Check the return value for success.

        if (hThread[i] == NULL)
        {
            ExitProcess(i);
```

```
        }
    }

    // Wait until all threads have terminated.

    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);

    // Close all thread handles upon completion.

    for(i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThread[i]);
    }
}
```

Завершение потока

Завершение потока можно организовать четырьмя способами:

- Функция потока возвращает управление (предпочтительный способ)
- Поток самоуничтожается вызовом функции ExitThread
- Один из потоков данного или стороннего процесса вызывает функцию TerminateThread (нежелательный способ)
- Завершается процесс, содержащий данный поток

3. Индивидуальные задания

Необходимо разработать однопоточную и многопоточную программы, осуществляющие решение поставленной задачи (таблица 2.1). В многопоточном приложении синхронизацию потоков выполнить указанным в таблице 2.1 способом. Приложение должно позволять вводить количество используемых потоков для вычислений (1-4). Необходимо замерять время решения задачи при использовании различного числа потоков, сравнить время вычислений и результаты решения, сделать выводы

Исходные данные для решения задачи разместить в текстовом файле.

Таблица 2.1 Варианты заданий

№	Условие задачи	Действия, выполняемые в потоках	Метод синхронизации
1	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных тупоугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного треугольника 2. Вычисление площади и углов треугольника	Переменная замок
2	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных остроугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного треугольника 2. Вычисление площади и углов треугольника	Строгое чередование
3	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных трапеций, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одной трапеции 2. Вычисление площади и углов трапеции	Флаги готовности
4	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных параллелограммов, образованных данными	1.Формирование массива точек, которые могут быть вершинами одного параллелограмма	Алгоритм Булочной

	точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	2. Вычисление площади и углов параллелограмма	
5	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить коэффициенты в уравнениях параллельных прямых, которые можно провести через заданные точки. Каждая прямая должна проходить не менее, чем рез 2 точки из списка. Результаты сохранить в текстовый файл в виде координат точек, значения коэффициентов в уравнении прямой.	1.Формирование массива коэффициентов A, B, C, D прямых в пространстве 2. Проверка, являются ли прямые параллельными	Test-and-Set
6	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных тупоугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного тупоугольного треугольника 2. Вычисление площади и углов треугольника	Swap
7	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных остроугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного остроугольного треугольника 2. Вычисление площади и углов треугольника	Флаги готовности
8	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных трапеций, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива, в строке которого находятся точки попарно лежащие на параллельных прямых (в одой строке размещаются координаты 4 точек: первые 2 лежат на одной прямой, 2 другие – на параллельной ей) 2. Вычисление площади и углов трапеции	Строгое чередование
9	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных параллелограммов, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива, в строке которого находятся точки попарно лежащие на параллельных прямых (в одой строке размещаются координаты 4 точек: первые 2 лежат на одной прямой, 2 другие – на параллельной ей) 2. Вычисление площади и углов параллелограмма	Переменная замок
10	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить коэффициенты в уравнениях параллельных прямых, которые можно провести через заданные точки. Каждая прямая должна проходить не менее, чем рез 2 точки из списка. Результаты сохранить в текстовый файл в виде координат точек, значения коэффициентов в уравнении прямой.	1.Формирование массива коэффициентов A, B, C, D прямых в пространстве 2. Проверка, являются ли прямые параллельными	Алгоритм Булочной
11	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных тупоугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1. Формирование массива точек, которые могут быть вершинами одного треугольника 2. Вычисление площади и углов треугольника	Test-and-Set
12	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных остроугольных треугольников, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного треугольника 2. Вычисление площади и углов треугольника	Swap
13	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных трапеций, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одной трапеции 2. Вычисление площади и углов трапеции	Флаги готовности
14	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить площади всех возможных параллелограммов, образованных данными точками. Результаты сохранить в текстовый файл в виде координаты вершин, значения углов, площадь.	1.Формирование массива точек, которые могут быть вершинами одного параллелограмма 2. Вычисление площади и углов параллелограмма	Test-and-Set
15	Текстовый файл содержит координаты точек в трёхмерном пространстве (N=1000). Необходимо определить коэффициенты в уравнениях параллельных прямых, которые можно провести через заданные точки. Каждая прямая должна проходить не менее, чем рез 2 точки из списка. Результаты сохранить в текстовый файл в виде координат точек, значения коэффициентов в уравнении прямой.	1.Формирование массива коэффициентов A, B, C, D прямых в пространстве 2. Поиск прямой, параллельной выбранной	Строгое чередование

Вопросы к защите

1. Дайте определение понятия тупика. Приведите пример.
2. Сформулируйте условия возникновения тупика.
3. Перечислите подходы к разрешению проблемы тупиков.
4. Алгоритм банкира.
5. Понятие надежных и не надежных состояний.
6. Алгоритм банкира для одного и нескольких видов ресурсов.
7. Недостатки алгоритма банкира
8. Охарактеризовать проблему синхронизации потоков.
9. Понятия активностей, детерминированных и недетерминирован активностей.
10. Условия детерминированности активностей Бернстайна.
11. Понятие критической секции.
12. Требования, предъявляемые к алгоритмам взаимодействия процессов.
13. Аппаратная поддержка взаимоисключений
14. Алгоритм взаимодействия процессов – «Запрет прерываний».
15. Алгоритм взаимодействия процессов – «Пременная замок».
16. Алгоритм взаимодействия процессов – «Строгое чередование».
17. Алгоритм взаимодействия процессов – «Флаги готовности».
18. Алгоритм взаимодействия процессов – Петерсона.
19. Алгоритм «Булочной» взаимодействия процессов.
20. Уметь внести исправления в программу для реализации любого алгоритма из вопросов 6-12