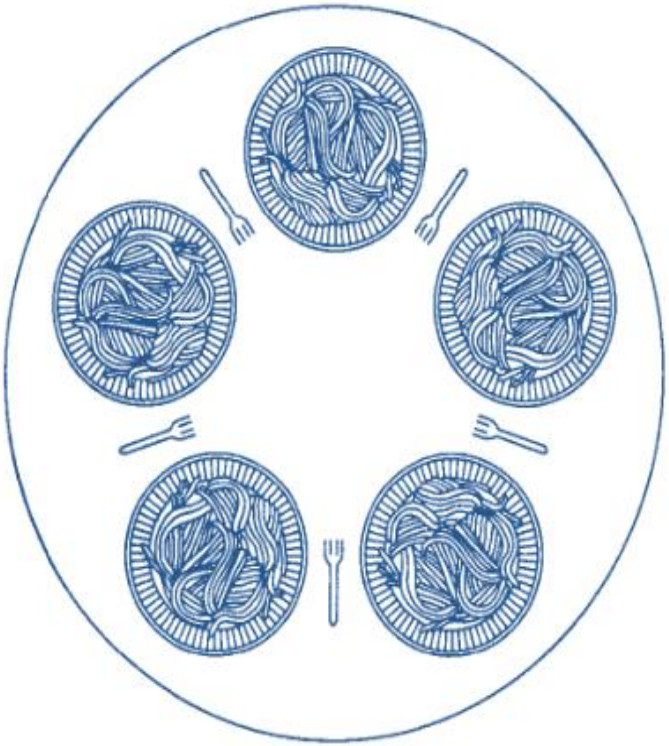
Модель обедающих философов



Сформулирована в 1965 году Дейкстрой.

Формулировка проблемы:

Пять философов сидят за круглым столом, и у каждого есть тарелка со спагетти. Спагетти настолько скользкие, что каждому философу нужно две вилки, чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка.

Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. Когда философ голоден, он пытается получить две вилки, левую и правую, в любом порядке. Если ему удалось получить две вилки, он ест некоторое время, затем кладёт вилки обратно и продолжает размышления.

Вопрос:

Можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает?

Решение задачи обедающих философов

```
# define N 5

# define LEFT (i) ((i – 1) % N)   /* номер левого философа */
# define RIGHT (i) ((i + 1) % N)  /* номер правого философа */

# define THINKING 0               /* состояние размышлений */
# define HUNGRY 1                 /* состояние голода */
# define EATING 2                 /* состояние поглощения пищи */

typedef int semaphore;            /* новый тип – семафор */
int state[N];                     /* состояния философов */
semaphore mutex = 1;              /* блокирующая переменная */
semaphore s[N];                   /* семафоры для философов */

void philosopher (int i)     /* i – номер философа от 0 до N-1 */
{
   while(TRUE)
   {
      think();               /* Философ размышляет */
      take_forks(i);         /* Берёт обе вилки или блокируется*/
      eat();                 /* Философ ест */
      put_forks(i);          /* Кладёт на стол обе вилки */
   }
}
void take_forks (int i)
{
  down(&mutex);              /* Вход в критическую область */
  state[i] = HUNGRY;         /* Состояние = голоден */
  test(i);                   /* Попытка получить две вилки */
  up(&mutex);                /* Выход из критической области */
  down(&s[i]);       /* Блокировка, если вилок не досталось */
}
```

Решение задачи обедающих философов (продолжение)

```
void put_forks(int i)
{
  down(&mutex);          /* Вход в критическую область */
  state[i] = THINKING; /* Состояние = размышляет */
  test(LEFT(i));       /* Проверка, может ли есть сосед слева */
  test(RIGHT(i));     /* Проверка, может ли есть сосед справа */
  up(&mutex);            /* Выход из критической области */
}

void test(int i)
{
  if (state[i] == HUNGRY && state[LEFT(i)] != EATING &&
      state[RIGHT(i)] != EATING)
  {
     state[i] = EATING;
     up(&s[i]);
  }
}
```

Модель читателей и писателей

Формулировка проблемы:

База данных, например бронирования билетов, к которой пытается получить доступ множество процессов. Можно разрешить одновременное считывание данных из базы, но если процесс записывает информацию в базу, доступ остальных процессов должен быть прекращён, даже доступ на чтение.

Вопрос:

Как запрограммировать читателей и писателей?

Решение проблемы читателей и писателей

```
typedef int semafore;
semaphore mutex = 1; /* Контроль доступа к переменной rc */
semaphore db = 1;    /* Контроль доступа к БД */
int rc = 0;          /* Количество читающих процессов*/
void reader()
{
   while(TRUE)
   {
      down(&mutex);  /* Получение монопольного доступа к rc */
      rc = rc + 1;
      if(rc == 1)
         down(&db);  /* Получение монопольного доступа к БД */
      up(&mutex);    /* Отказ от монопольного доступа к rc */
      read_database();
      down(&mutex);  /* Получение монопольного доступа к rc */
      rc = rc - 1;
      if(rc == 0)
         up(&db);    /* Отказ от монопольного доступа к БД */
      up(&mutex);    /* Отказ от монопольного доступа к rc */
      use_data_read();
   }
}
void writer()
{
   while(TRUE)
   {
      prepare_data();
      down(&db);       /* Получение монопольного доступа к БД */
      write_database();
      up(&db);         /* Отказ от монопольного доступа к БД */
   }
}
```

Модель «спящий брадобрей»

Формулировка проблемы:

В парикмахерской есть один брадобрей, его кресло и *n* стульев для посетителей.

Если желающих воспользоваться его услугами нет, брадобрей сидит в своём кресле и спит.

Если в парикмахерскую приходит клиент, он должен разбудить брадобрея.

Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет).

Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания.

Решение проблемы «спящего брадобрея»

```
#define CHAIRS 5

typedef int semaphore;
semaphore customers = 0; /* Количество ожидающих посетителей */
semaphore barbers = 0;   /*Количество брадобреев, ждущих клиентов*/
semaphore mutex = 1;     /* Для взаимного исключения */

int waiting = 0;         /* Ожидающие посетители */

void barber()
{
   while(TRUE)
   {
      down(&customers);  /* Если посетителей нет, засыпает */
      down(&mutex);      /* Запрос доступа к переменной waiting */
      waiting = waiting - 1;
      up(&barbers);      /* Брадобрей готов к работе */
      up(&mutex);        /* Отказ от доступа к waiting */
      cut_hair();
   }
}
void customer()
{
   down(&mutex);          /* Вход в критическую область */
   if (waiting < CHAIRS)
   {
      waiting = waiting + 1;
      up(&customers);     /* Разбудить брадобрея */
      up(&mutex);         /* Отказ от доступа к waiting */
      down(&barbers);     /* Если брадобрей занят, ожидать */
      get_haircut();
   }
   else
   {
      up(&mutex);         /* Нет свободных мест, ухожу */
   }
}
```

Создание вектора семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

`int semget (key_t key, int nsems, int semflag)`

- **`key`** – ключ
- **`sems`** – количество семафоров
- **`semflag`** – флаги, определяющие права доступа и те операции, которые должны выполняться (открытие семафора, проверка, и т.д.)

Возвращает целочисленный идентификатор созданного разделяемого ресурса, либо -1, если ресурс не удалось создать.

Создание ключа доступа

```
#include <sys/types.h>
#include <sys/ipc.h>
```

`key_t ftok (const char *pathname, char proj_id)`

- **`pathname`** – имя файла
- **`proj_id`** – символ, идентифицирующий проект

Возвращает целочисленный идентификатор созданного разделяемого ресурса, либо -1, если ресурс не удалось создать.

Операции над векторами семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semid, struct sembuf *semop,
                                    size_t nops)
```

- **semid** – идентификатор ресурса
- **semop** – указатель на структуру, определяющую операции, которые нужно произвести над семафором
- **nops** – количество указателей на эту структуру, которые передаются функцией semop() (операций может быть несколько и операционная система гарантирует их атомарное выполнение).

Операции над семафором

Операции определяются структурой:

```
struct sembuf
{
    short sem_num; /*номер семафора в векторе*/
    short sem_op;  /*производимая операция*/
    short sem_flg;  /*флаги операции*/
}
```

Значение семафора с номером **sem_num** равно **sem_val**

1) **sem_op = n → sem_val = sem_val + n**
2) **sem_op = -n → пока sem_val < |n|, процесс блокируется, затем sem_val = sem_val - |n|**
3) **sem_op = 0 → блокировка, пока sem_val ≠ 0**

Управление массивом семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

`int semctl (int semid, int num, int cmd, union semun arg)`

- **`semid`** – дескриптор массива семафоров
- **`num`** – индекс семафора в массиве
- **`cmd`** – операция
  - IPC_SET заменить управляющие наборы семафоров на те, которые указаны в **`arg.buf`**
  - IPC_RMID удалить массив семафоров и др.
- **`arg`** – управляющие параметры

Возвращает значение, соответствующее выполнявшейся операции (по умолчанию 0), в случае неудачи – -1

Управление массивом семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semid, int num, int cmd, union semun arg)

union semun {
   int val;  /* значение одного семафора */
   struct semid_ds *buf; /* параметры массива
                    семафоров в целом  (количество,
                   права доступа, статистика доступа)*/
   ushort *array;  /* массив значений семафоров */
}
```

Примеры

```
static struct sembuf sop_lock[2] = {
   0, 0, 0,/*ожидать обнуления семафора*/
   0, 1, 0 /*затем увеличить значение семафора на
             1*/
};

static struct sembuf sop_unlock [1] = {
   0, -1, 0     /*обнулить значение семафора*/
};


semop(semid, &sop_lock[0], 2);
semop(semid, &sop_unlock[0], 1);
```

Примеры

```
static struct sembuf sop_lock[1] = {
   0, -1, 0   /*ожидать разрешающего сигнала (1),
                затем обнулить семафор*/
};

static struct sembuf sop_unlock [1] = {
   0, 1, 0   /*увеличить значение семафора на 1*/
};


semop(semid, &sop_lock[0], 1);
semop(semid, &sop_unlock[0], 1);
```

Мониторы

Для упрощения написания программ в 1974 году Хоар (Hoare) и Бринч Хансен (Brinch Hansen) предложили механизм более высокого уровня чем семафоры – ***мониторы***.

Монитор – набор процедур, переменных и других структур данных, объединённых в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора.

При обращении к монитору в любой момент времени активным может быть только один процесс.

Для блокировки процессов используются ***переменные состояния*** и две операции: ***wait*** и ***signal***.

Когда процедура монитора обнаруживает, что она не может продолжать работу, она выполняет операцию ***wait*** на какой-либо переменной состояния. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор.

Другой процесс может активизировать ожидающего напарника, выполнив операцию ***signal*** на той переменной состояния, на которой он был заблокирован.

Литература

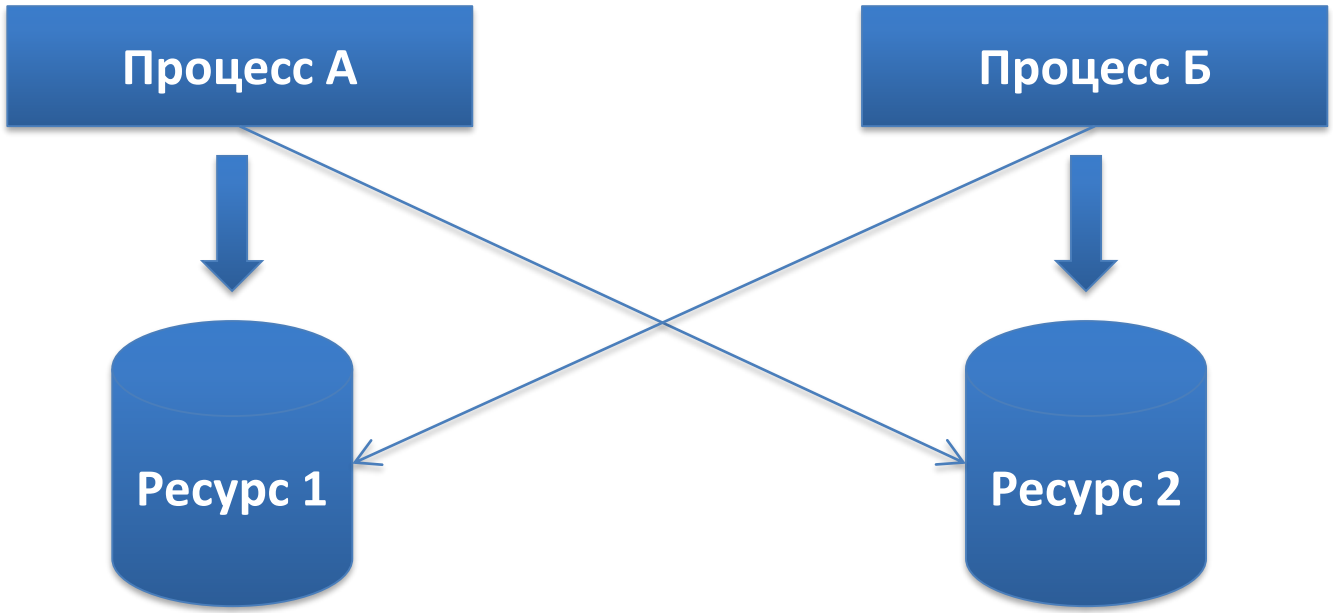
1. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста по разработке ПО: пер. с англ. – М.: ДМК Пресс, 2000. – 368 с. (раздел 8.3 Дополнительные средства межпроцессного взаимодействия)
2. Стивенс У. UNIX: взаимодействие процессов. – СПб.: Питер, 2003. – 576 с.
3. Иванов Н.Н. Программирование в Linux. Самоучитель. – СПб.: БХВ-Петербург, 2007. – 416 с. (часть 5. Межпроцессное взаимсодействие)

Взаимоблокировка процессов

Взаимоблокировка

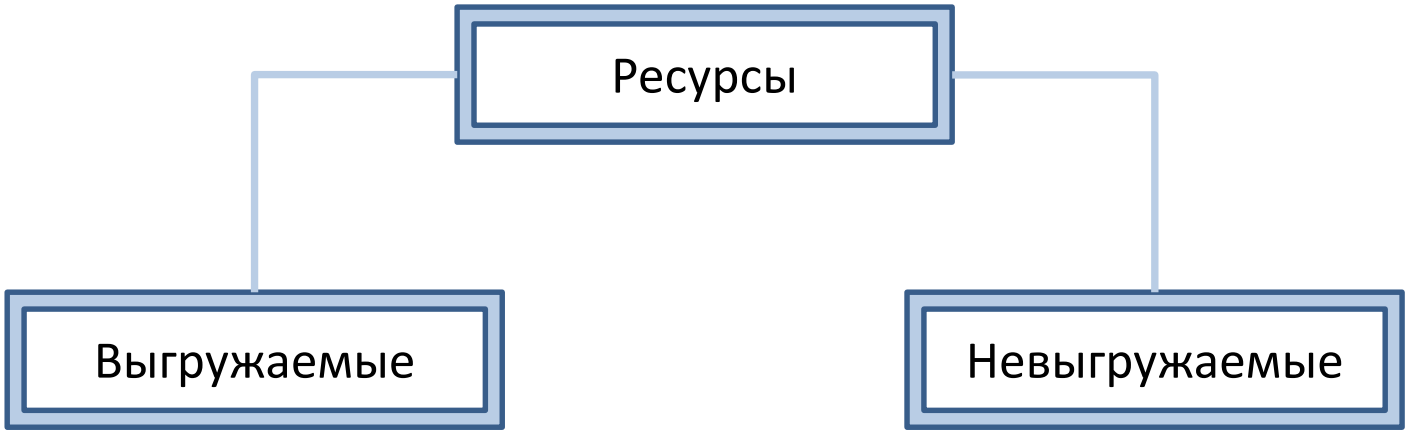
При одновременном использовании ресурсов несколькими процессами может возникнуть ситуация, когда часть процессов оказывается блокированными из-за ожидания ресурсов, принадлежащих другим процессам, которые, в свою очередь, ожидают освобождения ресурсов, принадлежащих первой части процессов.

Такая ситуация называется ***тупиковой*** (или ***дедлоком*** (deadlock), или ***клинчем*** (clinch)).



Понятие ресурса

Ресурс – объект предоставления доступа.
Ресурсом может быть аппаратное устройство или часть информации.



Последовательность событий, необходимых для использования ресурса:

1. Запрос ресурса.
2. Использование ресурса.
3. Возврат ресурса.

Условия взаимоблокировки

1. ***Условие взаимного исключения.*** Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. ***Условие удержания и ожидания.*** Процессы, в данный момент удерживающие полученный ресурсы, могут запрашивать новые ресурсы.
3. ***Условие отсутствия принудительной выгрузки ресурса.*** У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. ***Условие циклического ожидания.*** Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждёт доступа к ресурсу, удерживаемому следующим членом последовательности.

Для того чтобы произошла взаимоблокировка, должны выполниться все четыре условия. Если хоть одно из них отсутствует, тупиковая ситуация невозможна!

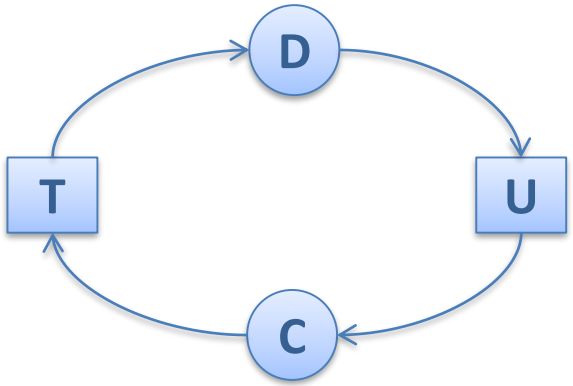
Схематичное представление использования ресурсов

Для схематичного представления использования ресурсов несколькими процессами используются **направленные графы**, в которых:

- два вида узлов:
  - – процессы показаны кружками;
  - – ресурсы нарисованы квадратами;
- ребро, направленное от узла ресурса к узлу процесса, означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом;
- ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу.

Ресурс R отдан процессу A

Процесс B ждёт ресурс S



Взаимоблокировка

Стратегии борьбы с взаимоблокировками

При столкновении с взаимоблокировками используются четыре стратегии:

1. Пренебрежение проблемой в целом.
2. Обнаружение и восстановление.
3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.
4. Предотвращение с помощью структурного опровержения одного из условий, необходимых для взаимоблокировки.

Страусовый алгоритм

Самым простым подходом является ***страусовый алгоритм***: воткните голову в песок и притворитесь, что проблемы вообще не существует.

Причина применения такого алгоритма: считается, что тупики возникают относительно редко, а затраты на решение высоки.

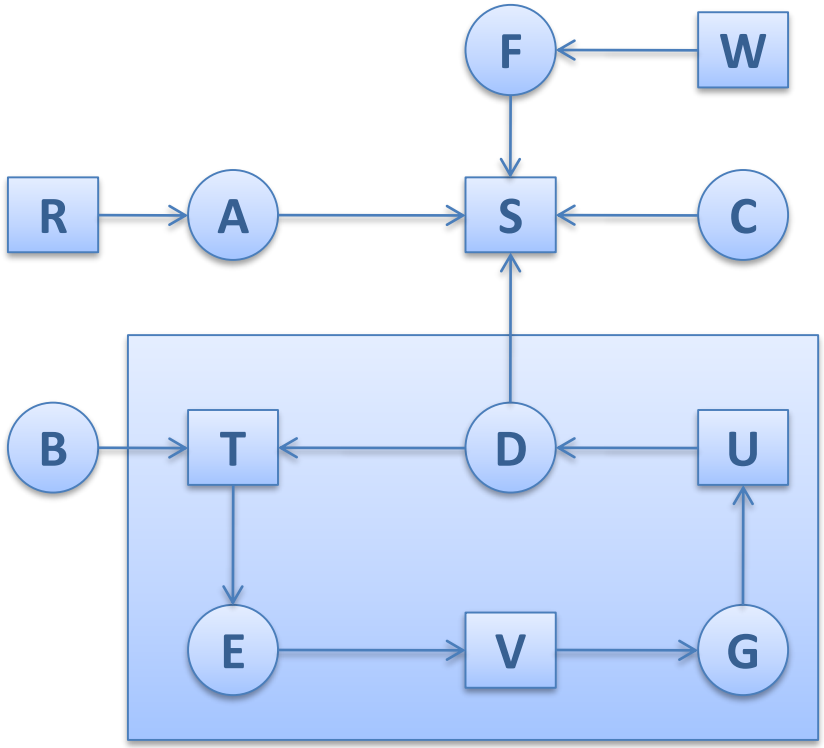
Обнаружение и устранение взаимоблокировок

При использовании этого метода система не пытается предотвратить попадание в тупиковые ситуации. Вместо этого она позволяет взаимоблокировке произойти, старается определить, когда это случилось, и затем совершает некие действия к возврату системы к состоянию, имевшему место до того, как система попала в тупик.

Обнаружение взаимоблокировки при наличии одного ресурса каждого типа

Пример. Состояние системы описывается следующим списком:

1. Процесс *A* занимает ресурс *R* и хочет получить ресурс *S*.
2. Процесс *B* ничего не использует, но хочет получить ресурс *T*.
3. Процесс *C* ничего не использует, но хочет получить ресурс *S*.
4. Процесс *D* занимает ресурс *U* и хочет получить ресурсы *S* и *T*.
5. Процесс *E* занимает ресурс *T* и хочет получить ресурс *V*.
6. Процесс *F* занимает ресурс *W* и хочет получить ресурс *S.*
7. Процесс *G* занимает ресурс *V* и хочет получить ресурс *U*.



Алгоритм поиска циклов (тупиков) в графе

Алгоритм использует одну структуру данных – список узлов *L*.

1. Для каждого узла *N* в графе выполняются следующие пять шагов, где *N* является начальным узлом.
2. Задаём начальные условия: *L* – пустой список, все рёбра не маркированы.
3. Текущий узел добавляем в конец списка *L* и проверяем количество появлений узла в списке. Если узел присутствует в двух местах, граф содержит цикл, и работа алгоритма завершается.
4. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 5, если нет, то переходим к шагу 6.
5. Случайным образом выбираем любое немаркированное исходящее ребро и отмечаем его. Затем по нему переходим к новому текущему узлу и возвращаемся к шагу 3.
6. Удаляем последний узел из списка и возвращаемся к предыдущему узлу. Обозначаем его текущим узлом и возвращаемся к шагу 3. Если это первоначальный узел, граф не содержит циклов, и алгоритм завершается.

Обнаружение взаимоблокировки при наличии нескольких ресурсов каждого типа

Существующие ресурсы

$E = \{E_1, E_2, \ldots, E_m\}$

Доступные ресурсы

$A = \{A_1, A_2, \ldots, A_m\}$

Матрица текущего распределения

$$\begin{bmatrix} C_{11}, C_{12}, \ldots, C_{1m} \\ C_{21}, C_{22}, \ldots, C_{2m} \\ \ldots \\ C_{n1}, C_{n2}, \ldots, C_{nm} \end{bmatrix}$$

Матрица запросов

$$\begin{bmatrix} R_{11}, R_{12}, \ldots, R_{1m} \\ R_{21}, R_{22}, \ldots, R_{2m} \\ \ldots \\ R_{n1}, R_{n2}, \ldots, R_{nm} \end{bmatrix}$$

m – число классов ресурсов, n – количество процессов, E_i – количество имеющихся в наличии ресурсов класса i, A_i – количество ресурсов класса i, доступных в текущий момент, C_{ji} – количество ресурсов класса i, занятое процессом j, R_{ji} – количество ресурсов класса i, которое хочет получить процесс j.

$$\sum_{j=1}^{n} C_{ji} + A_i = E_i$$

Алгоритм обнаружения взаимоблокировок при наличии нескольких ресурсов каждого типа

1. Ищем немаркированный процесс, для которого j-ая строка матрицы запросов R меньше вектора доступных ресурсов A или равна ему.

2. Если такой процесс найден, прибавляем j-ую строку матрицы C к вектору доступных ресурсов A, маркируем процесс и возвращаемся к шагу 1.

3. Если таких процессов не существует, работа алгоритма заканчивается.

Завершение алгоритма означает, что все немаркированные процессы, если таковые есть, попали в тупиковую ситуацию.

Пример

$E = \{4, 2, 3, 1\}$ $A = \{2, 1, 0, 0\}$

$$C = \begin{bmatrix} 0, 0, 1, 0 \\ 2, 0, 0, 1 \\ 0, 1, 2, 0 \end{bmatrix} \quad R = \begin{bmatrix} 2, 0, 0, 1 \\ 1, 0, 1, 0 \\ 2, 1, 0, 0 \end{bmatrix}$$

Последовательность предоставления ресурсов: 3, 2, 1.

Если процесс 3 потребует один экземпляр ресурса 4, то будет тупик.

Выход из взаимоблокировки

После обнаружения взаимоблокировки требуется восстановить работу системы. Это можно сделать следующими способами:

- ***Восстановление при помощи принудительной выгрузки ресурса***. Ресурс временно отбирается у текущего владельца и передаётся другому процессу

- ***Восстановление через откат***. Процессы периодически создают контрольные точки (запись состояния процесса). Процесс, занимающий необходимый ресурс, откатывается к тому моменту времени, перед которым он получил данный ресурс. Процесс запускается вновь с более раннего момента, когда он не занимал тот ресурс, который теперь предоставляется одному из процессов, попавших в тупик.

- ***Восстановление путём уничтожения процессов***. Грубейший и простейший способ выхода из ситуации взаимоблокировки. Один (или несколько) процессов уничтожается с освобождением всех выделенных ресурсов.

Избежание взаимоблокировок

В этому случае требуется, чтобы при удовлетворении запросов на использовании ресурсов ОС учитывала статистику их использования.

Состояние безопасно, если оно не находится в тупике и существует некоторый порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы вдруг захотят немедленно получить максимальное количество ресурсов.

Пример.

	Есть	Max			Есть	Max			Есть	Max			Есть	Max			Есть	Max	
A	3	9		A	3	9		A	3	9		A	3	9		A	3	9	**Состояние безопасно при последовательности предоставления ресурсов B, C, A**
B	2	4		B	4	4		B	0	-		B	0	-		B	0	-	
C	2	7		C	2	7		C	2	7		C	7	7		C	0	-	
Свободно: 3																			
A	3	9		A	4	9		A	4	9		A	4	9					**Состояние небезопасно при последовательности предоставления ресурсов A(1), B**
B	2	4		B	2	4		B	4	4		B	0	-					
C	2	7		C	2	7		C	2	7		C	7	7					

Алгоритм банкира

Алгоритм поиска последовательности предоставления ресурсов, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra) и носит название ***алгоритма банкира***.

Модель алгоритма основана на примере банкира в маленьком городке, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведёт ли выполнение каждого запроса к небезопасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу.

В случае обслуживания банкиром нескольких ресурсов (т.е. он выдает не только денежные кредиты, но и, например, материальные), то кредит выдаётся в том случае, если условие нахождения безопасного состояния выполняется для всех видов ресурсов.

Предотвращение взаимоблокировок

Предотвращение взаимоблокировок – это гарантия того, что ни одно из четырёх условий появления тупика никогда не возникнет.

Условие взаимного исключения можно избежать, если выделять ресурс, только когда это является абсолютно необходимым, и пытаться обеспечить ситуацию, в которой фактически претендовать на ресурс может минимальное количество процессов.

Условие удержания и ожидания можно избежать, требуя:

- чтобы любой процесс запрашивал все необходимые ресурсы до начала работы;
- чтобы процесс сначала временно освободил все используемые им в данный момент ресурсы. Затем этот процесс может попытаться сразу получить всё необходимое.

Предотвращение взаимоблокировок

Условие отсутствия принудительной выгрузки ресурса можно исключить, разрешив операционной системе «отбирать» ресурсы у процессов в любом случае. Для этого в ОС должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления хода вычислений.

Условие циклического ожидания можно устранить несколькими способами:

- следовать правилу, что процессу дано право только на один ресурс в конкретный момент времени. Если нужен второй ресурс, процесс обязан освободить первый.
- поддержка общей нумерации всех ресурсов. Процессы могут запрашивать ресурс, когда хотят этого, но все запросы должны быть сделаны в соответствии с нумерацией ресурсов.

Предотвращение взаимоблокировок

Предотвращение тупиков сопровождается высокими накладными расходами. Поэтому используется крайне редко.

Условие взаимного исключения. Чаще всего одновременное использование ресурсов несколькими процессами организовать сложно. Можно использовать выгрузку процессов на диск, однако это может также привести к проблеме тупика.

Условие удержания и ожидания. Многие процессы не знают заранее, сколько ресурсов им понадобится. При этом методе ресурсы не будут использоваться оптимально.

Условие отсутствия принудительной выгрузки ресурса. Принудительное освобождение некоторых ресурсов (например, принтера) может быть в лучшем случае сложно, а в худшем – невозможно.

Условие циклического ожидания. Существуют ситуации, когда невозможно найти порядок ресурсов, удовлетворяющий всех.

Сопутствующие вопросы

Двухфазовое блокирование. В первой фазе процесс пытается заблокировать все требуемые ресурсы по одному за раз. Если операция успешна, процесс переходит ко второй фазе , выполняя изменения и освобождение ресурсов. Никакой настоящей работы на первой фазе не совершается. Если во время первой фазы необходимый ресурс оказывается уже заблокированным, процесс просто освобождает все свои блокировки и начинает первую фазу заново.

Тупики без ресурсов. Может случиться, что два процесса заблокировали друг друга: каждый ждёт, когда другой выполнит некое действие. Такое часто случается с семафорами при неправильном порядке выполнения операций *down*.

«Голодание». Некоторые процессы никогда не получают требуемое, хотя они и не заблокированы. «Голодания» можно избежать, если использовать стратегию распределения ресурсов по принципу «первым пришёл – первым обслужен».