# Операционные системы
# Лекция 5
# Синхронизация при межпроцессовом взаимодействии

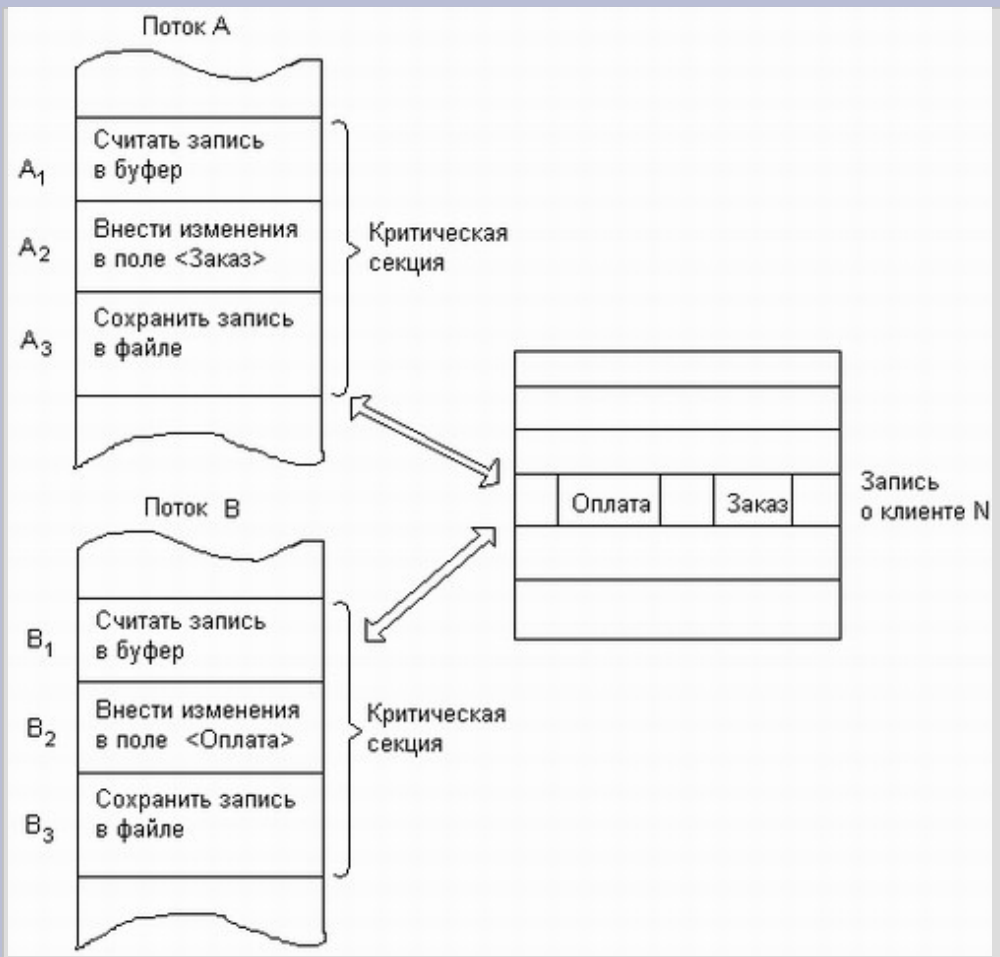
Боровцов Е.Г., 2016 г.

# Проблемы межпроцессового взаимодействия

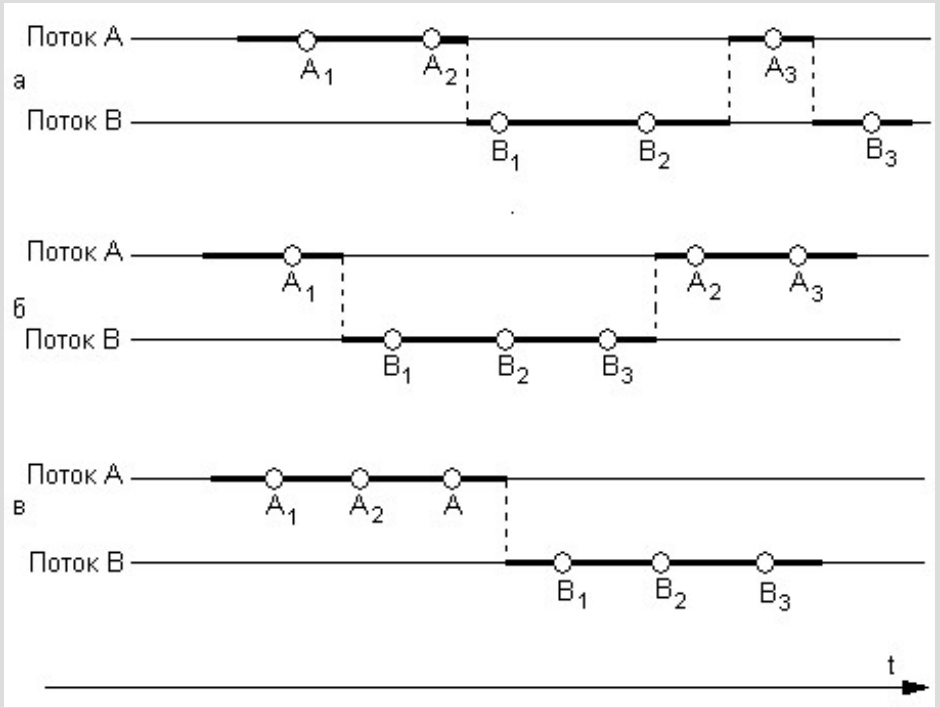
**Проблема синхронизации:**

- независимые процессы должны сохранять независимость;
- обеспечение механизма гарантированной доставки данных;
- исключение возможности взаимных блокировок при доступе к общим ресурсам.

# Проблемы межпроцессового взаимодействия: гонки (состязательная ситуация)




а) потеряна информация об оплате
б) потеряна информация о заказе
в) нормальное завершение операций



# Понятие критической секции

Критическая секция — это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими процессами (потоками) в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты. Во всех процессах(потоках), работающих с критическими данными, должна быть определена критическая секция. В общем случае в разных процессах(потоках) критическая секция может состоять из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один процесс(поток) в активном или в приостановленном состоянии. Этот прием называют **взаимным исключением**.

# Усллoвия безконфликтного доступа к общим данным

Хотя выполнение требования нахождения единственного процесса в критической секции позволяет избежать состязательных ситуаций, его недостаточно для того, чтобы параллельные процессы правильно выстраивали совместную работу и эффективно использовали общие данные. Для приемлемого решения необходимо соблюдение четырех условий:

**1. Два процесса не могут одновременно находиться в своих критических областях.**
**2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.**
**3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.**
**4. Процессы не должны находиться в вечном ожидании входа в свои критические области.**

# Реализация взаимного исключения: запрет прерываний

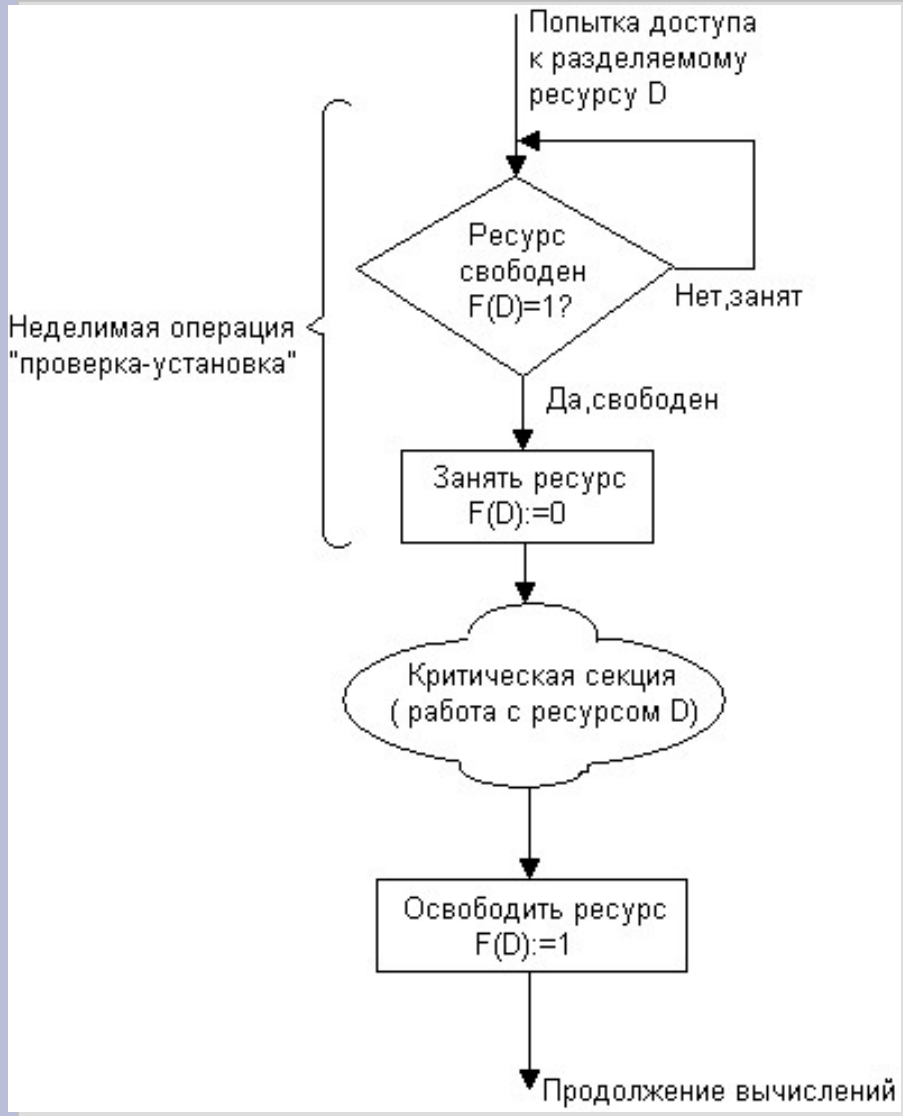
В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области.

Поскольку процесс запретил прерывания, он может исследовать и обновлять общую память, не опасаясь вмешательства со стороны любого другого процесса.

Недостатки:

1. возможность краха процесса, запретившего прерывания (прежде, чем он их опять разрешит);
2. снижение эффективности приема (или невозможность его реализации) в мультипроцессорных системах с общей памятью;
3. отрицательный эффект от запрета прерываний на длительный период времени(«потеря» событий);

# Реализация взаимного исключения: блокирующие переменные




Недостатки:

1. При отсутствии атомарной операции «проверка-установка» возможно возникновение гонок при доступе к блокирующим переменным. (на процессорах 80x86 решается с помощью команд семейства Bit_Test_and...(Clear|Reset|Set) ***BTC,BTR,BTS)***
2. В течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время.

# Реализация взаимного исключения: строгое чередование

```
turn=0;
```

```
while(true) {
        while(turn!=0); /*ждём turn=0*/
        critical_section();
        turn=1;
        noncritical_section();
}
```

```
while(true) {
        while(turn!=1); /*ждём turn=1*/
        critical_section();
        turn=0;
        noncritical_section();
}
```

Постоянная проверка значения переменной, пока она не приобретет какое-нибудь значение,называется **активным ожиданием.** Этого ожидания следует избегать,поскольку оно тратит впустую время центрального процессора. Активное ожидание используется только в том случае, если есть основание полагать, что оно будет недолгим. Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

# Реализация взаимного исключения: поддержка критической секции аппаратурой

```
Enter_region:
    MOVE REGISTER,#1            ; помещение 1 в регистр
    XCHG REGISTER,LOCK          ; обмен содержимого регистра и переменной lock
    CMP REGISTER,#0             ; было ли значение lock нулевым?
    JNE enter_region            ; если оно было ненулевым, значит, блокировка
                                ; уже установлена и нужно войти в цикл
    RET                         ; возврат управления вызывающей программе;
                                ; вход в критическую область осуществлен


leave region:
    MOVE LOCK,#0                ;  присвоение переменной lock нулевого значения
RET                             ; возврат управления вызывающей программе
```

В WinAPI существует набор примитивов для реализации синхронизации процессов через критическую секцию:

```
CRITICAL_SECTION cs;

InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
LeaveCriticalSection(&cs);
DeleteCriticalSection(&cs);
TryEnterCriticalSection(&cs);
```

# Средства синхронизации: семафоры

Семафоры представляют собой одну из форм IPC и используются для синхронизации доступа нескольких процессов к разделяемым ресурсам.

Классический семафор (generic) представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов. Объект System V IPC представляет собой **набор семафоров.**

Использование семафоров в качестве средства синхронизации доступа к другим разделяемым объектам предполагает следующую схему:

**- с каждым разделяемым ресурсом связывается один семафор из набора;**
**- положительное значение семафора означает возможность доступа к ресурсу (ресурс свободен), неположительное – отказ в доступе (ресурс занят);**
**- перед тем как обратиться к ресурсу, процесс уменьшает значение соответствующего ему семафора, при этом, если значение семафора после уменьшения должно оказаться отрицательным, то процесс будет заблокирован до тех пор, пока семафор не примет такое значение, чтобы при уменьшении его значение оставалось неотрицательным;**
**- закончив работу с ресурсом, процесс увеличивает значение семафора (при этом разблокируется один из ранее заблокированных процессов, ожидающих увеличения значения семафора, если таковые имеются);**
**- в случае реализации взаимного исключения используется двоичный семафор, т.е. такой, что он может принимать только значения 0 и 1: такой семафор всегда разрешает доступ к ресурсу не более чем одному процессу одновременно.**

# Средства синхронизации: логика работы семафора

Следующая функция иллюстрирует логику работы семафора в зависимости от значения переменной состояния (semvalue) и управляющей переменной sem_op.

```
void semaphore (int sem_op)
{
  static int semvalue; // Внутренняя переменная
  if (sem_op != 0) {
    if (sem_op < 0) while (semvalue < ABS(sem_op));
    semvalue += sem_op;
  }
  else while (semvalue != 0);
}
```

Отрицательное значение sem_op соответствует операции проверки доступности ресурса и вызывает приостановку потока, если доступ к ресурсу заблокирован.Положительное значение сигнализирует о высвобождении ресурса.

# Пример использования семафоров при доступе к разделяемой памяти-сервер[5]

*Процесс-сервер — создает блок разделяемой памяти и обеспечивает прием текстовых строк и их печать от программы-клиента*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "semtypes.h"
int main(int argc, char * argv[]) {
  key_t key;
  int shmid, pid, semid;
  struct sembuf buf[2];
  struct memory_block * mblock;
  key = 1234567;
  semid = semget(key, 3, 0666|IPC_CREAT);
  buf[0].sem_num = 0;
  buf[0].sem_flg = SEM_UNDO;
  buf[1].sem_num = 1;
  buf[1].sem_flg = SEM_UNDO;
  semctl(semid, 0, SETVAL, 0);
  shmid = shmget(key, sizeof(struct memory_block), 0666 | IPC_CREAT);
  mblock = (struct memory_block *) shmat(shmid, 0, 0);
  strcpy(mblock->string, "Hello!");
  buf[0].sem_op = -1;
  buf[1].sem_op = 1;
  semop(semid, (struct sembuf*) &buf[1], 1);
  while (strcmp("q\n", mblock->string) != 0)  {
      semop(semid, (struct sembuf*) &buf, 1);
      printf("String sent by the client is: %s", mblock->string);
      if (strcmp("q\n", mblock->string) != 0)
      strcpy(mblock->string, "Ok!");
      buf[0].sem_op = -1;
      buf[1].sem_op = 1;
      semop(semid, (struct sembuf*) &buf[1], 1);
  }
  printf("Server got q and exits\n");
  shmdt((void *) mblock);
  shmctl(shmid, IPC_RMID, 0);
  semctl(semid, 2, IPC_RMID);
  return EXIT_SUCCESS;
}
```

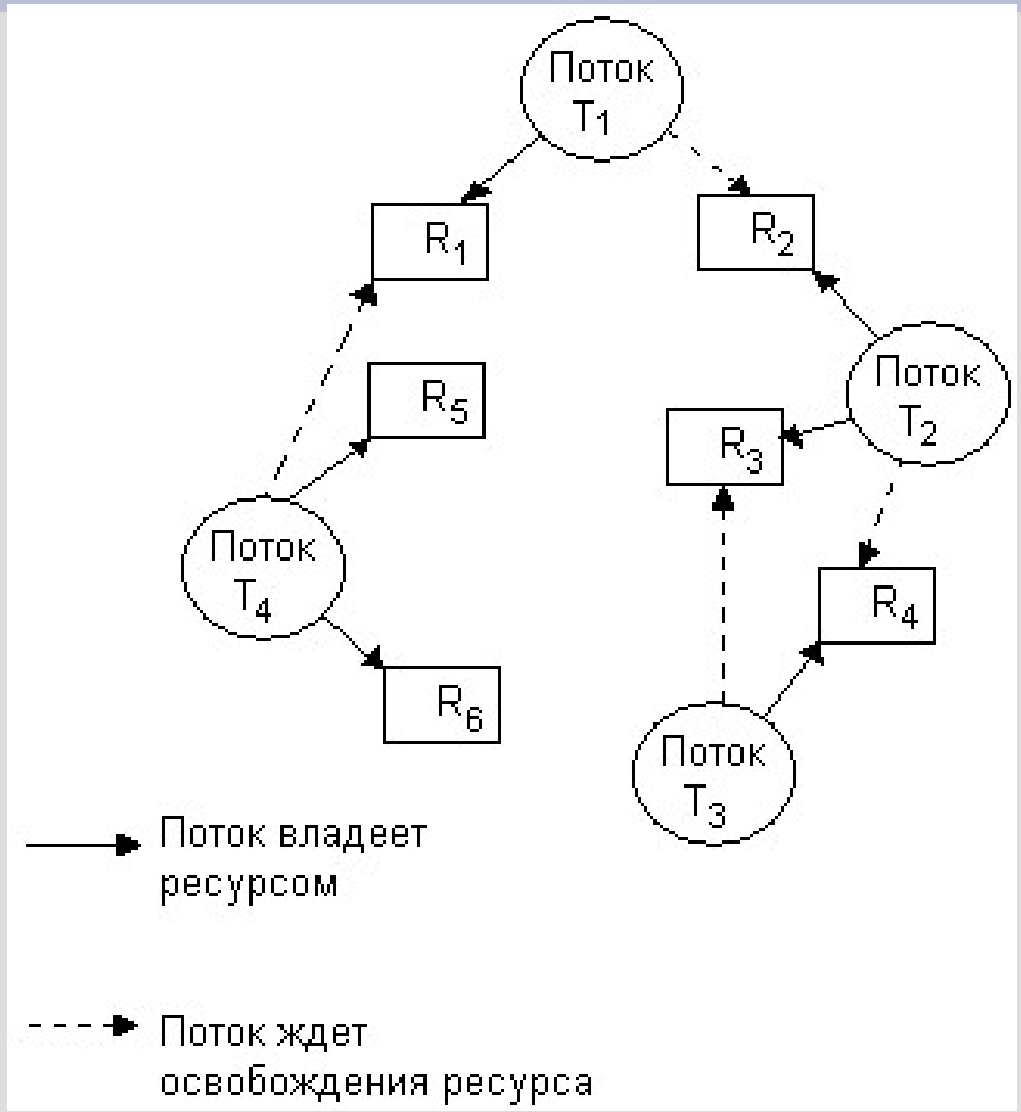
# Пример использования семафоров при доступе к разделяемой памяти-клиент[5]

*Процесс-клиент — считывает строку со стандартного ввода и передает ее серверу через разделяемую память*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "semtypes.h"
int main(int argc, char * argv[]) {
  key_t key;
  int shmid;
  struct memory_block * mblock;
  int semid;
  struct sembuf buf[2];
  key = 1234567; // генерация ключа
  shmid = shmget(key, sizeof(struct memory_block), 0666);
  if (shmid == -1)  {
     printf("Server is not running!\n");
     return EXIT_FAILURE;
  }
  semid = semget(key, 2, 0666);
  buf[0].sem_num = 0;
  buf[0].sem_flg = SEM_UNDO;
  buf[1].sem_num = 1;
  buf[1].sem_flg = SEM_UNDO;
  mblock = (struct memory_block *) shmat(shmid, 0, 0);
  buf[1].sem_op = -1;
   while (strcmp("q\n", mblock->string) != 0)  {
     int i = 0;
     semop(semid, (struct sembuf*) &buf[1], 1);
     printf("Server sends %s\n", mblock->string);
     while ((i < (MAXLEN - 1)) && ((mblock->string[i++] =
  getchar()) !=  '\n') );
     mblock->string[i] = 0;
     buf[0].sem_op = 1;
     buf[1].sem_op = -1;
     semop(semid, (struct sembuf*) &buf, 1);
   }
   printf("Client exits\n");
   shmdt((void *) mblock);
   return EXIT_SUCCESS;
  }
```

```
// Semtypes.h
#ifndef SHMEM_TYPES
#define SHMEM_TYPES
#define MAXLEN 512
struct memory_block {    char string[MAXLEN]; };
#endif
```

# Средства синхронизации: Взаимная блокировка(тупик)



# Средства синхронизации: другие примитивы синхронизации

Помимо рассмотренных, в современных ОС существуют и другие примитивы синхронизации процессов:

1. Двоичные семафорры — мьютексы (mutex);
2. обьекты синхронизации в пользовательском пространстве — фьютекс(futex) (fast user space mutex - Linux);
3. системные вызовы ожидания: семейство Wait (Unix);

Вызовы Windows API:

WaitForSingleObject()

WaitForMultiplyObject()

# Методы межпроцессового взаимодействия: литература

1.Таненбаум Э., Бос Х**.** Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил. — (Серия «Классика computer science»).
2. Основы операционных систем.Курс лекций. Учебное пособие / В.Е.Карпов, К.А.Коньков / Под редакцией В.П.Иванникова. - М.:ИНТУИТ.РУ «Интернет-Университет Информационных Технологий» - 2005, 536 с.
3. Вдовикина Н.В., Машечкин И.В., Терехин А.Н.,Томилин А.Н.  Операционные системы: взаимодействие процессов:учебно-методическое пособие. Издательский отдел факультета ВМиК МГУ 2008, - 215 с.
4. Иванов Н.Н.Программирование в Linux. Самоучитель. - СПб.:БХВ-Петербург, 2007.- 416с.:ил.
5. Андрей Боровский. Цикл интернет-статей. Программирование для Linux. ЧАСТЬ 3. Объекты SVID IPC.
6. Андрей Боровский. Цикл интернет-статей. Программирование для Linux. ЧАСТЬ 7,8. Потоки.
7. В.Г. Олифер, Н.А. Олифер. Сетевые операционные системы.- СПб.:Питер.-2002. - 544 с.