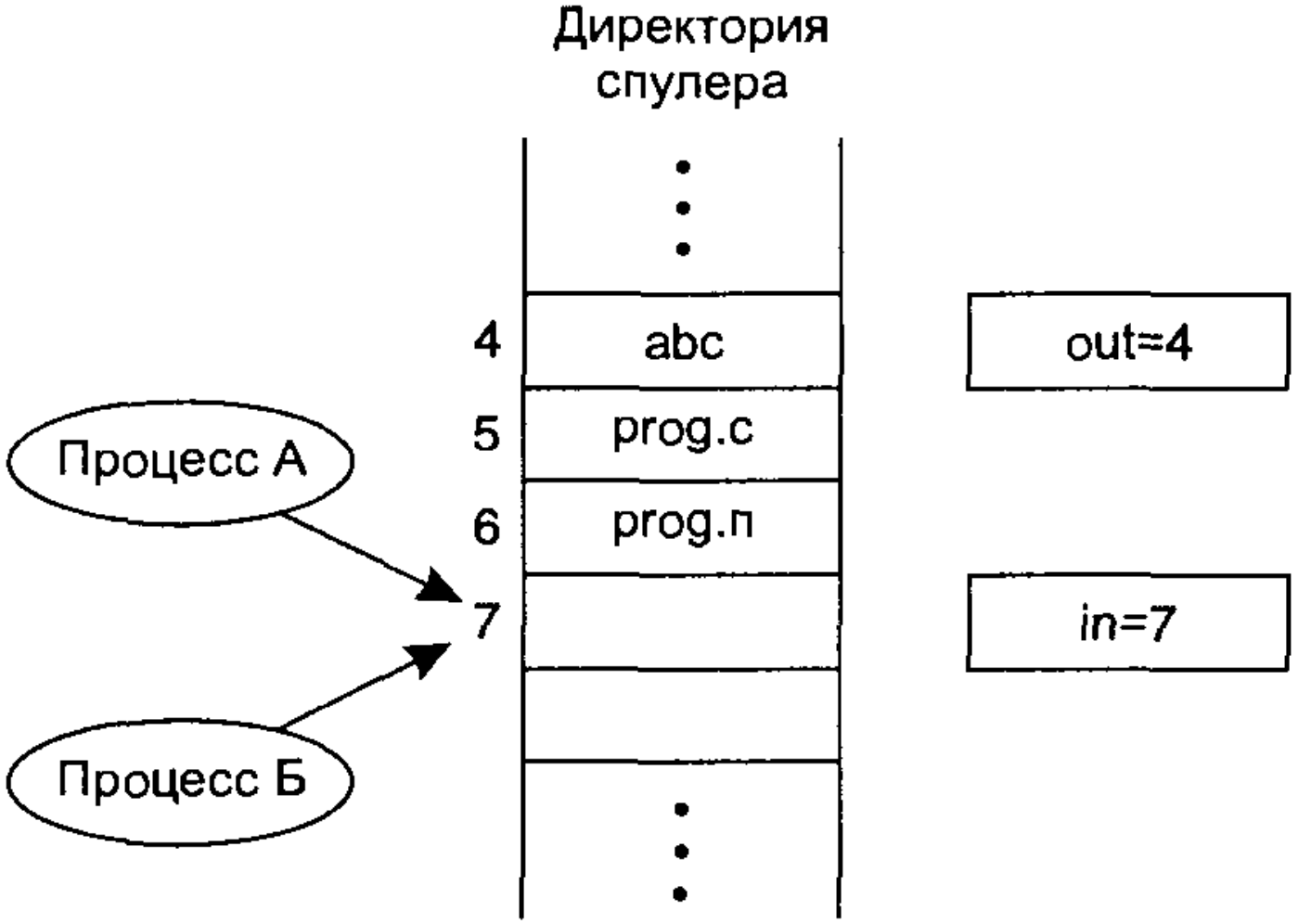
# Операционные системы

Лекция № 6

Синхронизация процессов и потоков

(4 часа)

# Состояние состязания (гонки)



# Состояние состязания (гонки)

- **состояние состязания** - ситуация, в которой два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, в каком порядке процессы получают доступ к данным**.**

# Критические секции

**Критической областью** или **критической секцией (critical section)** называется часть программы, в которой есть обращение к совместно используемому неразделяемому ресурсу.

А сам ресурс - **критическим ресурсом**.

Основным способом решения проблемы состязания является запрет одновременного обращения к критическому ресурсу более чем одним процессом (**взаимное исключение, mutual exclusion**), т.е. запрет на одновременное нахождение двух или более процессов в критической секции.

# Условия правильного использования общих данных

1. Два процесса не должны одновременно находиться в критических областях.
2. Не должно быть предположений об относительных скоростях или количестве процессов.
3. Процессы, находящиеся вне критической области, не должны блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

# Способы реализации взаимного исключения с активным ожиданием

1. **Запрет прерываний**

   Достоинства: простота реализации.

   Недостатки: возможность краха ОС при сбое пользовательского процесса, невозможность использования в многопроцессорных системах.

# Способы реализации взаимного исключения с активным ожиданием

2. **Переменные блокировки**
   С разделяемым ресурсом связана общая переменная со значением, например, 0, если критическая область свободна и 1 – если нет.

**В общем случае проблема не решается!!!**

*** Если действие по проверке и установке нового значения атомарно – решение работает**

# Способы реализации взаимного исключения с активным ожиданием

**3. Строгое чередование**

```
while (TRUE) {
while(turn!=0) ;     /*цикл*/
critical_region() ;
turn=1;
noncritical region(); }
```

```
while (TRUE) {
while(turn!=1) ;     /*цикл*/
critical_region() ;
turn=0;
noncritical region(); }
```

# Способы реализации взаимного исключения с активным ожиданием

**Недостатки**:

Бесцельная загрузка процессора,

Требование строгого чередования доступа к критической секции.

Нарушение 3 требования (один процесс блокирован другим, не находящимся в критической области)

# Способы реализации взаимного исключения с активным ожиданием

Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**, а алгоритмы, ее использующие – **алгоритмами активного ожидания (busy waiting algorithms).**

Блокировка, использующая активное ожидание, называется **спин-блокировкой (spinlock)**.

# Способы реализации взаимного исключения с активным ожиданием

4. а) **Алгоритм Деккера** – первое известное корректное программное решение проблемы взаимного исключения.

   б) **Алгоритм Питерсона** (1981 год) – более простой алгоритм программного взаимного исключения.

   в) **Алгоритм кондитера** – решение для случая n-процессов.

```
#define FALSE 0
#define TRUE  1
int turn;                       /* Чья сейчас очередь? */
int interested[2];              /* Все переменные изначально */
                                /*  равны 0 (FALSE) */
void enter_region(int process) /* Процесс 0 или 1 */
{       int other;              /* Номер второго процесса */
        other = 1 - process; /* Противоположный процесс */
        interested[process] = TRUE; /* Индикатор интереса */
        turn = process;                 /* Установка флага */
        while (turn == process && interested[other] == TRUE);
}
void leave_region(int process) /* process-процесс, покидающий кр.с.*/
{       interested[process] = FALSE; /* индикатор выхода из кр. с. */
}
```

# Способы реализации взаимного исключения с активным ожиданием

5. **Команда TSL** (Test and Set Lock — проверить и заблокировать )

   TSL RX, LOCK
   читает в RX содержимое слова памяти
   LOCK и сохраняет в LOCK некоторое ненулевое значение. Операция чтения и записи **неделима**

```
enter_region:
        TSL REGISTER, LOCK
        CMP REGISTER, #0
        JNE enter_region


        RET


leave_region
        MOVE LOCK, #0
        RET
```

# Резюме:

Решение Питерсона(Деккера, АК) и использование команды TSL корректны, но обладают общим недостатком – используют активное ожидание, что может вызвать **проблему инверсии приоритета** (блокировка высокоприоритетного процесса низкоприоритетным).

# Примитивы синхронизации (реализация взаимного исключения без активного ожидания)

1. Sleep и wakeup.
2. Семафоры.
3. Мьютексы.
4. Мониторы.
5. Передача сообщений.
6. Барьеры синхронизации.

# 1. sleep и wakeup

sleep – системный запрос, который блокирует вызывающий процесс, пока его не запустит другой процесс

wakeup – системный запрос с одним параметром, указывающим процесс, который следует запустить.

Для обхода состояния гонки используется активационный бит (для двух процессов)

# 2. Семафоры

Новый тип целочисленных переменных, предложенный Дейкстрой в 1965 году.

Значение семафора может быть 0 или некоторым положительным числом.

Над семафорами определены две операции: **down** и **up**.

**down** – сравнивает значение семафора с 0, и если семафор больше 0, уменьшает его на 1 и возвращает управление, если = 0, то переводит вызывающий процесс в состояние ожидания.

**up** – увеличивает значение семафора. Если с семафором связаны ожидающие процессы, которые не могли завершить операцию down, один из них выбирается системой и ему разрешается завершить операцию down.

Операции down и up – являются атомарными (все действия выполняются как неделимые).

# 3. Мьютексы

(mutex – **MUT**ual **EX**clusion – взаимное исключение)

Упрощенная версия семафора. Позволяет только управлять доступом к совместно используемым ресурсам или коду.

Может находится в одном из двух состоянии: неблокированном (0) и блокированном (все остальное)

Поддерживает две операции: *mutex_lock* и *mutex_unlock*. (В некоторых реализациях ОС больше, например *mutex_trylock* )

# 4. Мониторы

Примитивы высокого уровня, предложенные в 1974 Хоаром и Бринч Хансеном (в 1973). Представляют собой набор процедур, переменных и других структур данных, объединенных в особый модуль - монитор.

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

В упрощенном варианте реализованы в Java, C#.

# 5. Передача сообщений

Использует два системных вызова:

`send` и `receive`.

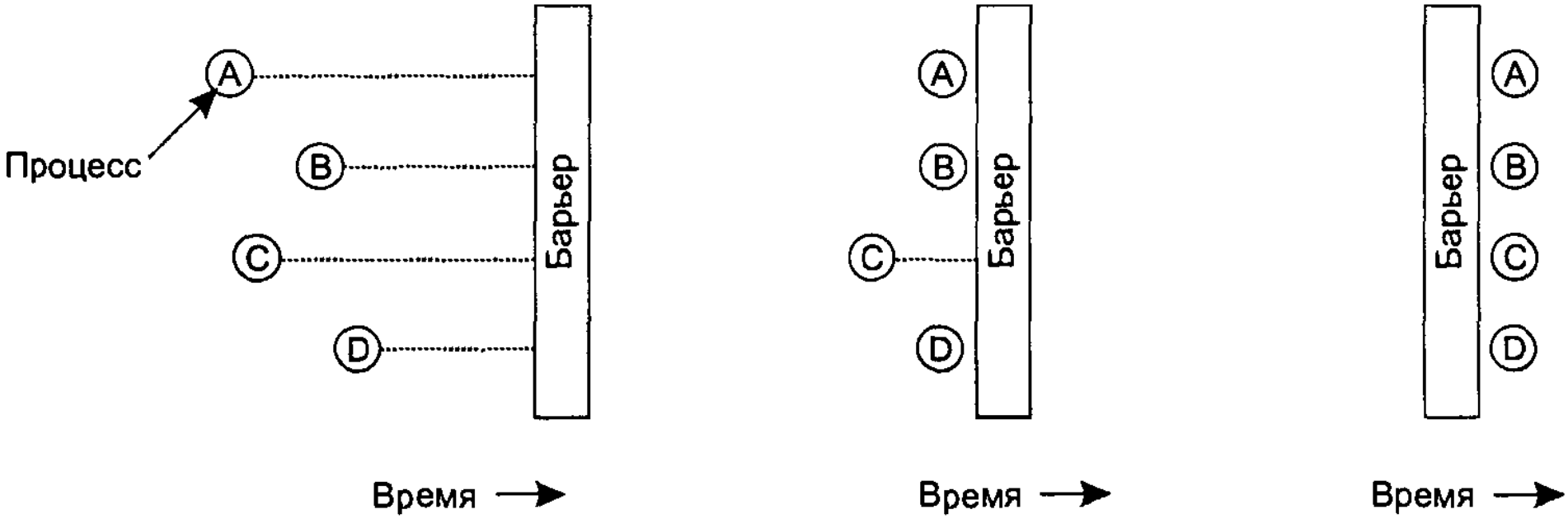
`send` – посылает запрос заданному адресату.

`receive` – получает сообщение от указанного источника. Если сообщения нет, запрос блокируется или сразу возвращает ошибку.

# 6. Барьеры

Барьеры – примитивы синхронизации для групп процессов, блокирующие процессы до завершения последнего из группы.

# Барьеры


A
B
C
D
Процесс
Барьер
Время
A
B
C
D
Барьер
Время
A
B
C
D
Барьер
Время

# Объекты синхронизации

В Windows **объектами синхронизации** называются объекты ядра, которые могут находится в одном из двух состояний: сигнальном (signaled) и несигнальном (nonsignaled).

Объекты синхронизации могут быть разделены на 4 категории

# Объекты синхронизации

1) Объекты синхронизации для параллельных потоков:

- Мьютекс
- Событие
- Семафор

2) Объект синхронизации, переходящий в сигнальное состояние по истечении заданного интервала времени:

- Ожидающий таймер

# Объекты синхронизации

3) Объекты, переходящие в сигнальное состояние по завершению:

- Работа (job)
- Процесс (process)
- Поток (thread)

4) Объекты, переходящие в сигнальное состояние после получения сообщения об изменении содержимого объекта:

- Изменение состояния каталога (change notification)
- Консольный ввод

# Мьютексы

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES
  lpMutexAttributes,BOOL bInitialOwner,
  LPCTSTR lpName);
HANDLE OpenMutex(DWORD
  dwDesiredAccess, BOOL bInheritHandle,
  LPCTSTR lpName);
 BOOL ReleaseMutex(HANDLE hMutex);

WaitForSingleObject(), CloseHandle();
```

# Мьютексы

*NIX:

Тип данных: `pthread_mutex_t`

```
pthread_mutex_init();

pthread_mutex_lock();
pthread_mutex_unlock();

pthread_mutex_destroy();
```

# Семафоры

В Windows:

```
CreateSemaphore()
OpenSemaphore()
ReleaseSemaphore()
WaitForSingleObject()
CloseHandle();
```

# Семафоры

*NIX: два API – System V и POSIX

```
sem_init();
sem_wait();
sem_post();
sem_destroy();
```

# События. Задача уведомления

- **Событием** (event) называется оповещение о некотором выполненном действии.
- События используются для оповещения одного потока о том, что другой поток выполнил некоторое действие.
- Задача  оповещения одного потока о некотором действии другого потока, называется **задачей условной синхронизации** или **задачей оповещения**.

# События . Задача уведомления

События разделяются на:

- События с ручным сбросом;
- События с автоматическим сбросом.

```
CreateEvent();
OpenEvent();
SetEvent();
ResetEvent();
PulseEvent();
```

# Критические секции CRITICAL_SECTION

- Реализуют концепцию критических областей кода;
- CRITICAL_SECTION не являются объектами ядра;
- Не имеют дескрипторов и не могут разделяться процессами;
- При входе в критическую секцию блокируют поток, если в критической области уже есть другой поток.

# Критические секции CRITICAL_SECTION

```
// инициализация критической секции
VOID InitializeCriticalSection
  (LPCRITICAL_SECTION
  lpCriticalSection);
// разрушение объекта критическая секция
VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

# Критические секции CRITICAL_SECTION

```
// вход в критическую секцию
VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
// попытка войти в критическую секцию
BOOL TryEnterCriticalsection(
  LPCRITICAL_SECTION lpCriticalSection);
// выход из критической секции
VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

# Тупики

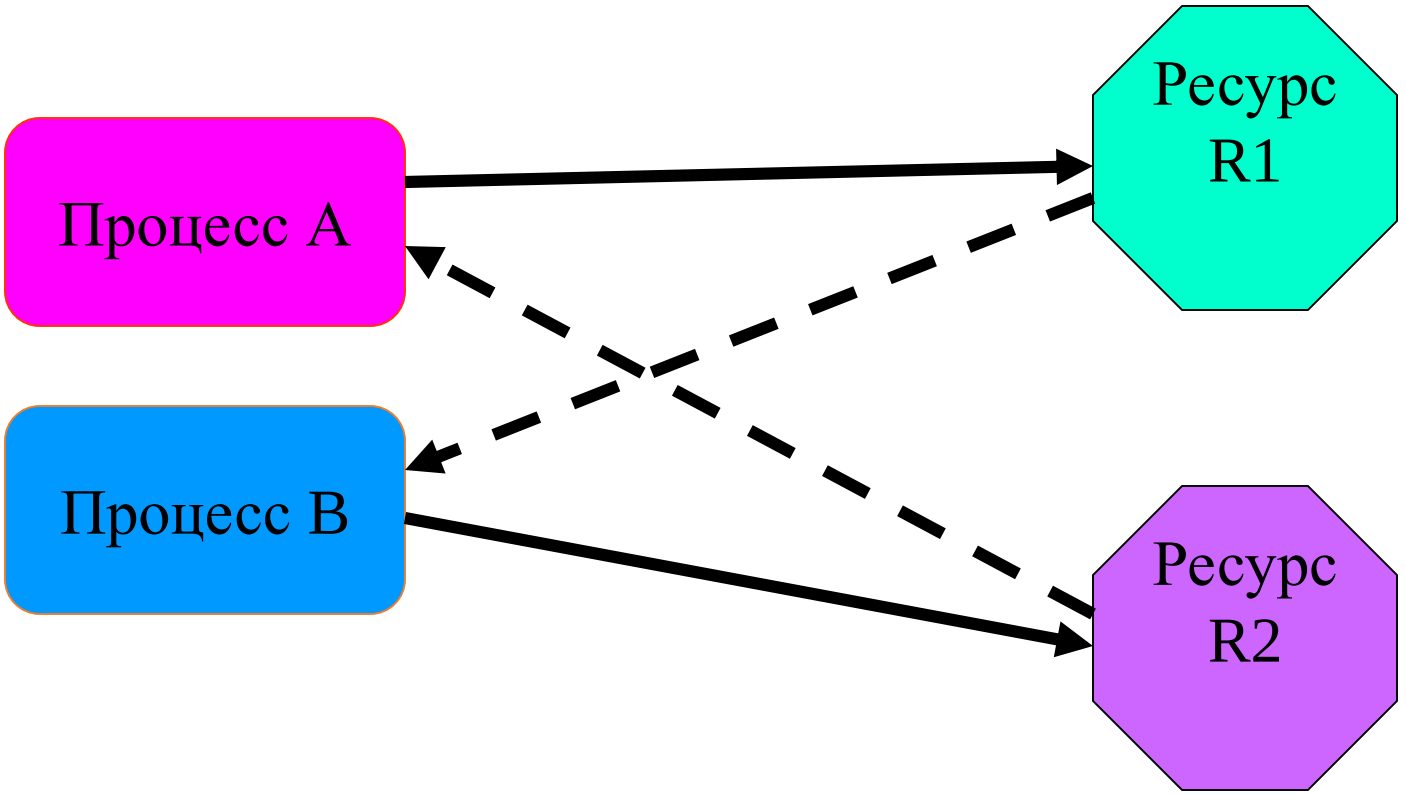
Говорят, что поток находится в **тупике (deadlock)**, если он ждет событие, которое никогда не произойдет.

Событие может никогда не произойти по следующим двум причинам:

1. не существует потока, который оповещает о наступлении ожидаемого события;
2. поток, оповещающий о наступлении ожидаемого события, существует, но сам находится в тупике.

Если в тупике находится хотя бы один из потоков процесса, то считается, что этот процесс также находится в тупике.

# Тупики



# Условия возникновения тупиков

Были сформулированы Коффманом, Элфиком и Шошани в 1970 г.:

1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.

# Условия возникновения тупиков

3. Условие неперераспределяемости (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

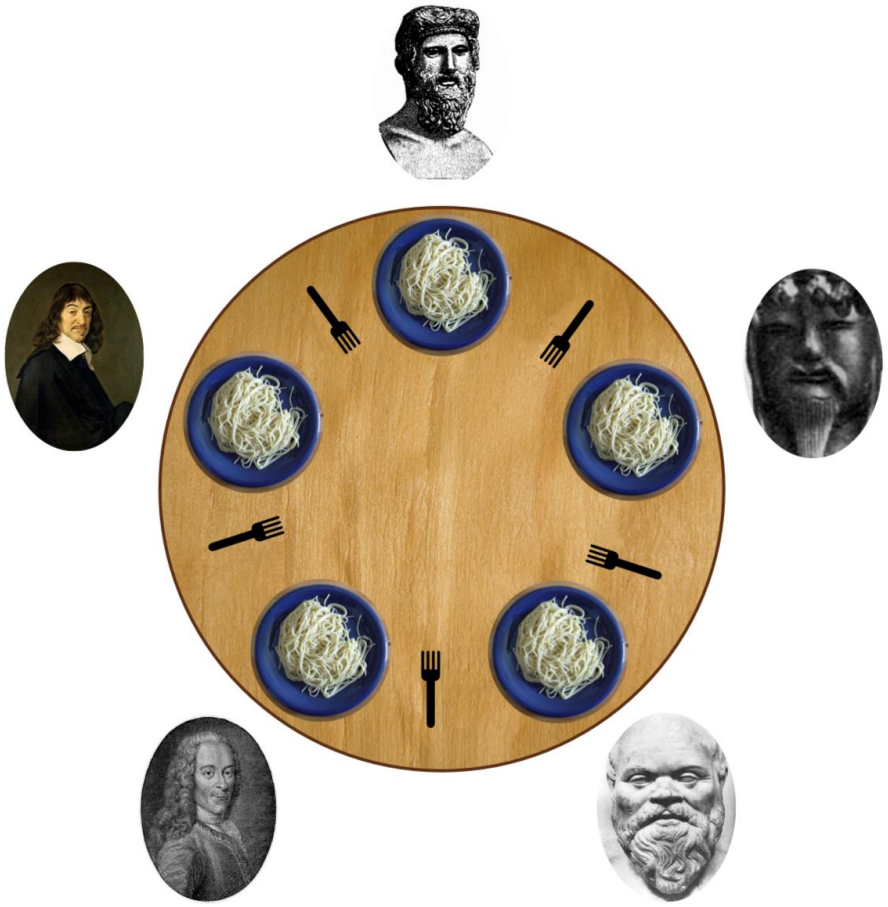
# Направления борьбы с тупиками

- Игнорирование проблемы в целом
- Предотвращение тупиков (тщательное распределение ресурсов – алгоритм банкира, нарушение условии возникновения тупиков)
- Обнаружение тупиков
- Восстановление после тупиков (termination, preemption, rollback).

# Классические проблемы IPC

- Проблема обедающих философов (1965 г., Дейкстра);
- Проблема взаимного исключения
- Проблема читателей и писателей;
- Проблема производителей и потребителей;
- Проблема спящего брадобрея

# Проблема обедающих философов



# Проблема обедающих философов

- 5 философов сидят за круглым столом, перед каждым на столе стоит тарелка со спагетти, между каждыми двумя соседями лежит вилка;
- Каждый философ некоторое время размышляет, потом берет две вилки и ест спагетти, затем кладет вилки на стол и размышляет дальше
- Найти алгоритм, который позволит философам организовать доступ к вилкам таким образом, чтобы каждый имел возможность насытиться и никто не умер с голоду.

# Проблема читателей и писателей

- Задана некоторая разделяемая область памяти
- К этой области могут обращаться некоторое количество «читателей» и «писателей»
- «Читатели» могут получить доступ на чтение одновременно, при этом «писатели» не допускаются
- Одновременно только один «писатель» может записывать данные, другие «писатели» и «читатели» должны ожидать

# Проблема производителей и потребителей (проблема ограниченного буфера)

- Дан буфер заранее фиксированного размера
- Имеется один (или более) процесс-производитель, который помещает данные в буфер. Если буфер полностью заполняется – производители «засыпают»
- Имеется один (или более) процесс-потребитель, который считывает данные из буфера. При опустошении буфера потребители блокируются.

# Проблема спящего брадобрея

- В парикмахерской имеется одно кресло для посетителя, которого обслуживают и N мест для ожидающих
- Если посетителей нет – брадобрей спит
- Если приходит посетитель и кресло свободно – посетитель садится в него, будит брадобрея и брадобрей бреет его
- Если кресло занято – посетитель садится на одно из мест для ожидающих
- Если все места заняты – посетитель уходит