

Операционные системы

Курс лекций для гр. 4057/2

Лекция №4

Содержание

Раздел 2. Взаимодействие процессов

2.1 Синхронизация

Виды взаимодействия процессов

Сотрудничество: несколько процессов решают общую задачу

Часто это лучше, чем один большой процесс:

- структуризация большой задачи - разделение ее на автономные части
- повышение производительности:
 - в однопроцессорной системе - путем совмещения операций в разных устройствах компьютера во времени
 - пример: для вв/вы порождается сыновний процесс, а основной продолжается - не блокируется
 - в многопроцессорной (или многоядерной) системе - путем распараллеливания задачи на подзадачи, выполняемые на разных процессорах (ядрах)

В распределенной системе процессы в разных локациях взаимодействуют, если решают общую задачу

Конкуренция: и независимые, и сотрудничающие процессы часто соперничают за доступ к *неразделяемым (критическим)* ресурсам, владение которыми разрешается только одному процессу за раз (напр, к принтеру или файлу, открытому для записи)

Три проблемы:

- взаимное исключение
- взаимная блокировка (тупики)
- голодание

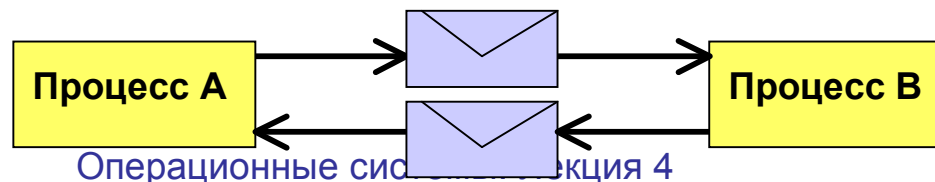
NB: процессор и диск – не критические, а разделяемые ресурсы

Как ОС поддерживает взаимодействие процессов

1. Взаимодействующие процессы должны *синхронизировать* свои действия, т.е. иногда *ждать*, пока другие процессы не выполнят то, что позволит им продолжиться, и, в свою очередь, сигнализировать

Примитивы синхронизации (в ядре ОС): <i>семафоры, замки, условные переменные, мониторы, события, ...</i>
Аппаратная поддержка: <i>система прерываний, атомарные инструкции</i>

2. Сотрудничающие процессы, кроме того, обмениваются информацией. Для этого служат средства *коммуникации* – обмена сообщениями



Виды синхронизации

1. *Взаимное исключение (mutual exclusion)*

одновременного доступа к критическому ресурсу

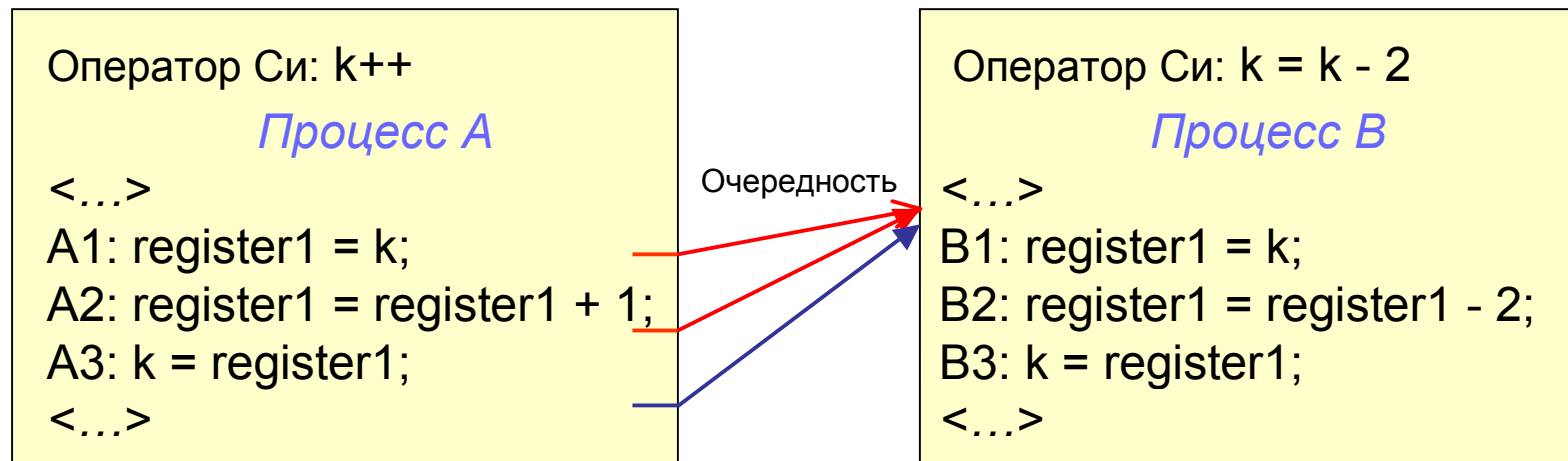
- Фрагмент кода, на котором процесс должен обладать монопольным доступом к критическому ресурсу, называется *критическим участком (critical section)*

2. *Условная синхронизация* – ожидание выполнения некоторого события или условия, наложенного на результат другого процесса

Пример задачи взаимного исключения

Тип задачи: обновление данных

Частный случай: счетчик k – критический ресурс; критические участки:

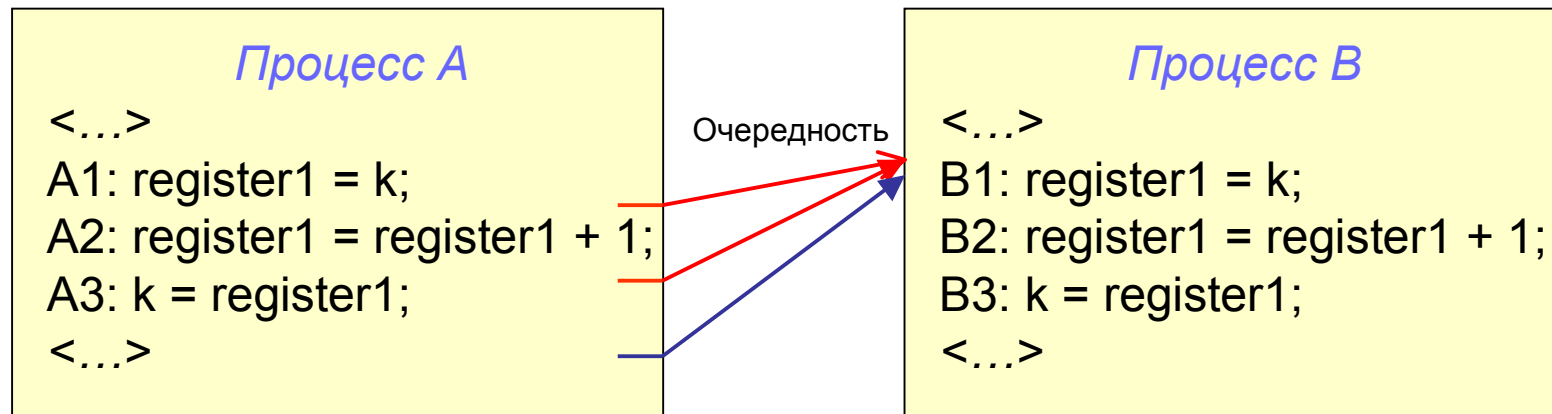


Если сразу после A1 или A2 выполнится B1 (очередность по стрелке \rightarrow), то конечный результат k – неправильный, отличный от случая \rightarrow , т.е. последовательного выполнения фрагментов (симметрично то же для B)

(Вопрос 1)

Не важно, выполняются эти процессы на одном процессоре (при вытесняющей диспетчеризации) или же на разных процессорах в многопроцессорной системе !

Непредсказуемость и невоспроизводимость результата



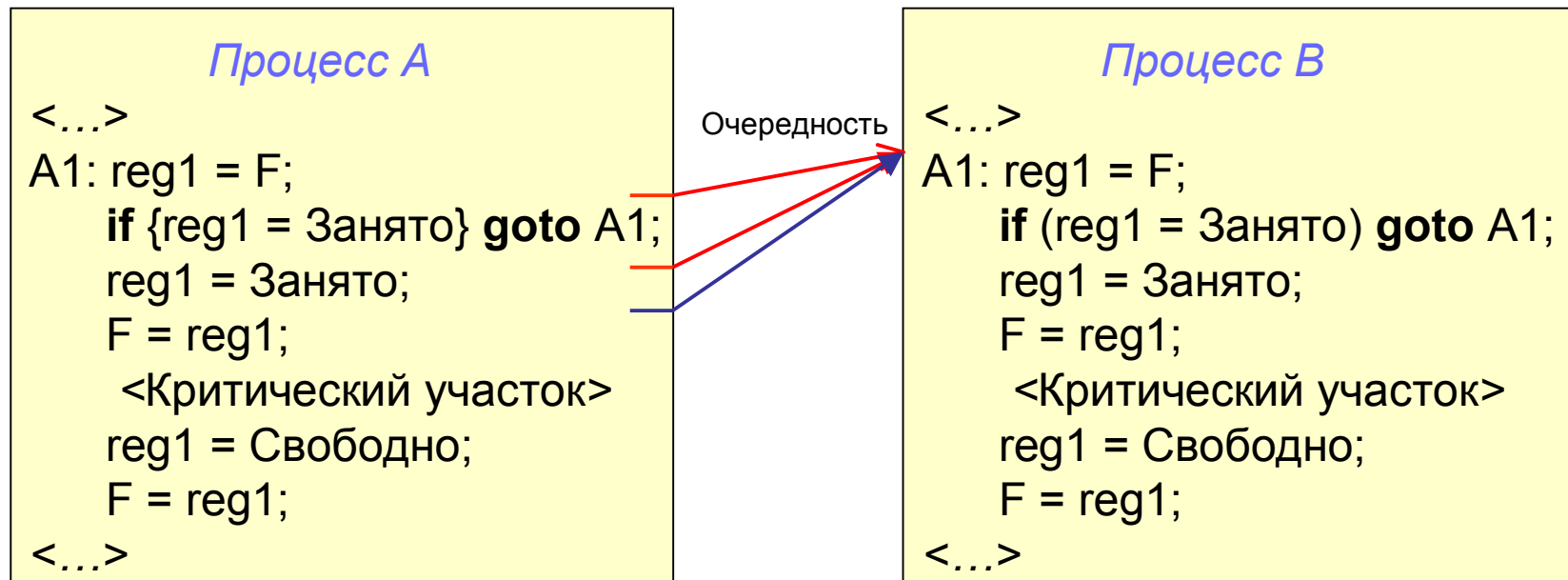
Никаких предположений о скоростях процессов и моментах их переключения делать нельзя!

Ошибки из-за одновременного доступа могут происходить очень редко → их трудно обнаруживать при тестировании !

**Выполнение процессами критических участков должно быть
взаимоисключающим во времени**

Программные решения проблемы

Простая проверка флага занятости F – неверное решение:



Причина ошибки: флаг занятости – тоже критический ресурс !

Программные решения: алгоритмы Деккера, Петерсена, Дейкстры, Кнута, Лэмпорта, ...

Идея алгоритма «пекарни» (bakery) взаимного исключения n процессов (Л.Лэмпорт, 1974)

Аналогия с очередью в пекарню, где раздаются билеты с номерами

- ❖ Номера билетов последовательно возрастают на 1
- ❖ Каждый процесс, желающий войти в критический участок, получает очередной билет
- ❖ Далее он сам определяет, когда может войти в критический участок – когда его номер билета становится минимальным. Отличия от реальной пекарни:
 - ❖ процессы могут получить одинаковые билеты – тогда приоритет отдается процессу с минимальным номером
 - ❖ В реальной пекарне – централизованное определение текущего минимального номера: он высвечивается на табло

Достоинства алгоритма

- ✓ Простота: длина псевдокода - минимальная из известных решений
- ✓ Подходит для использования в многопроцессорных системах, т.к. каждый элемент двух совместно используемых массивов изменяется только одним процессом, и поэтому нет нужды во взаимном исключении доступа к ним
- ✓ Алгоритм работает даже при отказе одного или нескольких процессов (при условии, что их элементы массивов установлены в 0)

Алгоритм «пекарни» Лэмпорта

```
boolean choosing[n]; //массив: какие процессы получают билет; инициализ. false
int ticket[n];       //номера билетов для каждого процесса; инициализируется 0
void main() {
    x = threadNumber(); // сохранить номер текущего процесса
    while (true) {
        choosing[x] = true;           // начать выбор билета
        ticket[x] = maxValue(ticket) + 1; // получить билет; maxValue возвращает
                                         // наибольший номер билета из числа выданных
        choosing[x] = false;          // выбор билета закончен
        for (int i = 0; i < n; i++) { // цикл проверки состояния всех процессов
            if(i == x) {continue};    // нет нужды проверять собственный билет
            while (choosing[i] != false); // занятое ожидание, пока i-ый процесс
                                         // выбирает билет (Вопрос 2)

            // занятое ожидание, пока значение текущего билета не станет минимальным:
            while (ticket[i] != 0 && ticket[i] < ticket[x]);
            // при одинаковых номерах - предпочтение меньшему номеру процесса:
            if (ticket[i] == ticket [x] && i < x )
                while (ticket[i] != 0); // занятое ожидание, пока i-ый процесс
                                         // не покинет критический участок
        }
        // критический участок
        ticket[x] = 0 // выход из взаимного исключения
        25.09.09 //код вне критического участка
    }
}
```

Аппаратное взаимное исключение – 1-й способ

Процесс запрещает прерывать себя на время выполнения критического участка

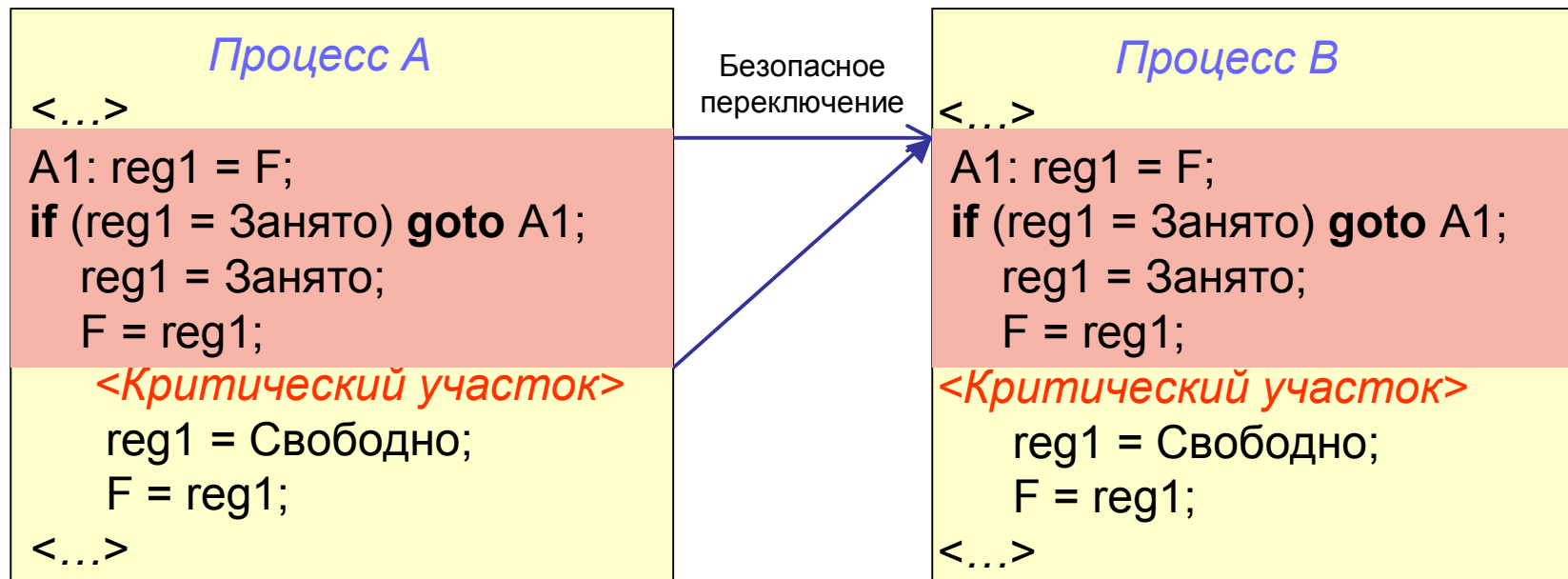
➤ Кто первым вошел в критический участок, тот работает в нем, монопольно владея процессором до конца участка

Недостатки

- Противоречие с диспетчеризацией процессов, особенно если критический участок – длинный (квантование? вытеснение?)
- Проблемы в случае многопроцессорной системы

Аппаратное взаимное исключение – 2-й способ

Сделать непрерываемой проверку и установку флага занятости



Специальные машинные команды - *атомарные инструкции*:
проверка/установка значения - одна неделимая операция

В многопроцессорной системе – блокировка шины памяти на это время

Атомарные инструкции процессора

Test&Set: прочесть значение и записать вместо него 1

Exchange (Intel x86): обменять местами значения регистра и памяти

Compare&Swap (М 68000): прочесть значение в памяти; если оно равно регистру1, обменять значения регистра2 и памяти

и т.п.

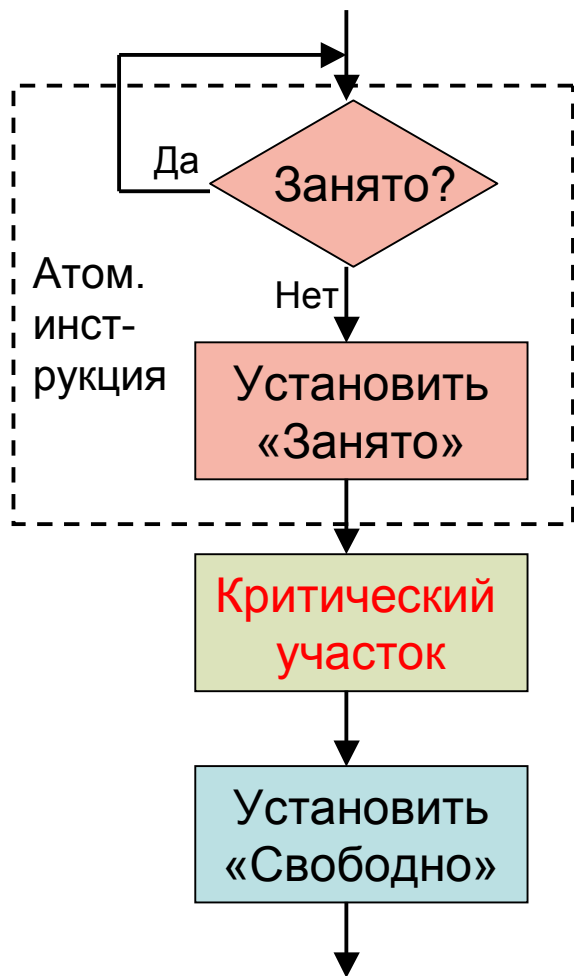
Действие инструкции Test&Set описывается так:

```
bool TS(bool lock) {  
    bool current = lock; //сохранить текущее значение lock  
    lock = true;         //установить lock (запереть замок)  
    return current;      //возвратить текущее значение  
}
```

Код взаимного исключения в каждом процессе в программе с глобальной переменной bool lock, иницируемой в false:

```
некритический участок;  
while (TS(lock));  
критический участок;  
lock = false;  
некритический участок;
```

Взаимное исключение с помощью атомарных инструкций



Недостатки

- Активное ожидание (пережидание занятости – busy waiting) – бессмысленная трата времени процессора
- В приоритетных системах есть опасность инверсии приоритета, когда высокоприоритетный процесс занят активным ожиданием
 - то же при любом программном решении !
- Порядок входа процессов в критические участки недетерминирован → опасность голодания процессов, ждущих освобождения ресурса

Для более эффективных решений ОС содержат функции, настроенные над аппаратными средствами синхронизации – *примитивы синхронизации: замки, семафоры, мониторы* и др.

Замок, или мьютекс

(Mutex, Mutual exclusion)

Объектная модель замка

```
class Lock {  
    public:  
        void Acquire();    // ждать, пока замок не откроется, и тогда запереть его  
        void Release();    // отпереть замок и разблокировать процесс, ждущий в  
                           // Acquire  
    private:  
        bool Closed;  
        Queue Q;  
}
```

Конструктор

```
Lock::Lock {  
    Closed = False;    //замок открыт  
    Q = 0;             //очередь пуста  
}
```

Методы замка

Захватить критический ресурс

```
Lock::Acquire() {  
    запретить прерывания;  
    if ( Closed == True) {  
        заблокировать текущий процесс  $P_i$  и поставить его в очередь Q,  
        разрешить прерывания;  
    }  
    else {  
        Closed = True;  
        разрешить прерывания;  
    }  
}
```

Освободить критический ресурс

```
Lock::Release() {  
    запретить прерывания;  
    if (Q не пуста) {  
        взять  $P_k$  из очереди Q; разблокировать  $P_k$ ;  
    }  
    else Closed = False;  
    разрешить прерывания;  
}
```

*Красный шрифт –
действия ОС*

Применение замка

Zamok = new Lock; // глобальный объект

Процесс 1

```
while (true) {  
    не критический участок;  
    Zamok. Acquire();  
    критический участок;  
    Zamok. Release();  
    не критический участок;  
}
```

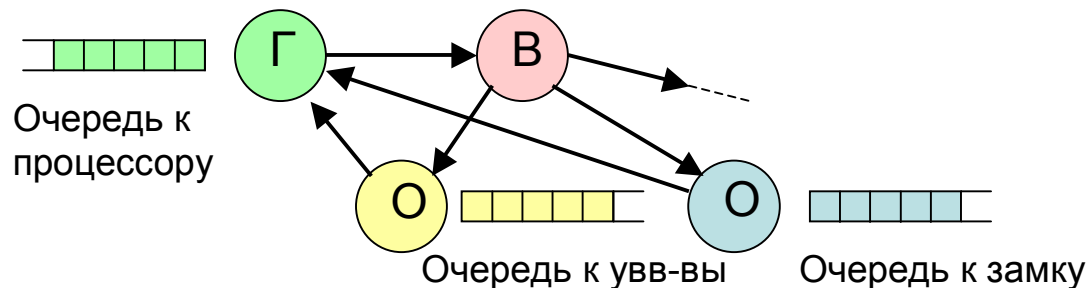
Процесс 2

```
while (true) {  
    не критический участок;  
    Zamok. Acquire();  
    критический участок;  
    Zamok. Release();  
    не критический участок;  
}
```

- Произвольное число процессов
- Критические участки выполняются строго последовательно, а не критические – могут параллельно

Преимущества замка

- ❑ Вместо активного ожидания процесс блокируется – т.е., работает диспетчеризация очереди процессов, ждущих открытия замка как общего ресурса
 - ❑ нет бессмысленной траты времени процессора и инверсии приоритета
 - ❑ можно регулировать дисциплину очереди, избегая голодания



- ❑ Прерывания запрещаются только на короткое время проверки замка, а не на все время работы с критическим ресурсом, как при непосредственном запрете прерываний

Другая реализация замка использует атомарные инструкции, так что запрета прерываний не требуется; это годится для многопроцессорных систем

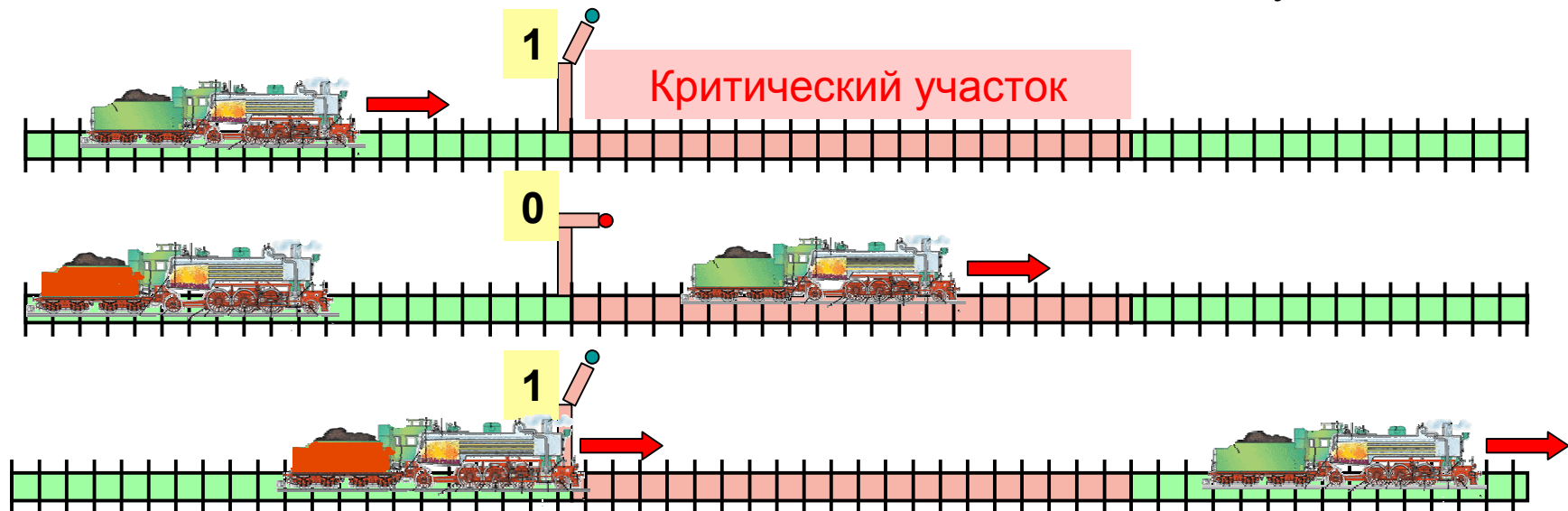
Семафоры (Дейкстра, 1965)

Счетный семафор – это специальный тип глобальной переменной: инкапсулированная неотрицательная целая переменная с двумя атомарными операциями: Wait и Signal

При $S > 0$: вход в критический участок свободен

При $S = 0$: вход в критический участок закрыт

Двоичный семафор отличается от счетного тем, что S может принимать только два значения: 0 и 1; по действию он эквивалентен замку:



Автоблокировка на ЖД транспорте

Эдсгер Дейкстра (1930-2002)

Отец программирования как науки

Основные заслуги



- Участие в разработке языка Алгол-60 и компилятора для него
- Понятия критического участка, взаимного исключения и семафора
- Структурное программирование (без goto) и поуровневая структуризация программ
- Логическое доказательство правильности программ
- Алгоритмы
 - Нахождение кратчайшего пути в графе
 - Алгоритм банкира для предупреждения тупиков
 - и др.
- 1300 статей и заметок

Вычислительные машины – самый большой вызов человеческому разуму.

Э.Дейкстра

Псевдокод семафора как класса

```
class Semaphore {  
    public:  
        void Wait();  
        void Signal();  
    private:  
        int S;  
        Queue Q;  
}
```

```
Semaphore::Wait() {  
    if(S > 0)  
        S = S - 1;  
    else {добавить этот процесс в  
        очередь Q; заблокировать процесс}  
}
```

```
Semaphore::Semaphore(int k) {  
    S = k;        //инициализация  
    Q = 0;        //очередь пуста  
}
```

```
Semaphore::Signal() {  
    if(очередь пуста)  
        S = S + 1;  
    else{удалить процесс из очереди  
        Q; разблокировать процесс}  
}
```

В реализации семафора, как и замка, атомарность операций с переменной S обеспечивается запретом прерываний или атомарными инструкциями, а управление очередью ждущих процессов осуществляется ОС

Применение семафоров

Взаимное исключение

В каждом из N процессов критический участок обрамляется двумя операциями с семафором M, инициализируемым 1:

M.Wait();

Критический участок;

M.Signal();

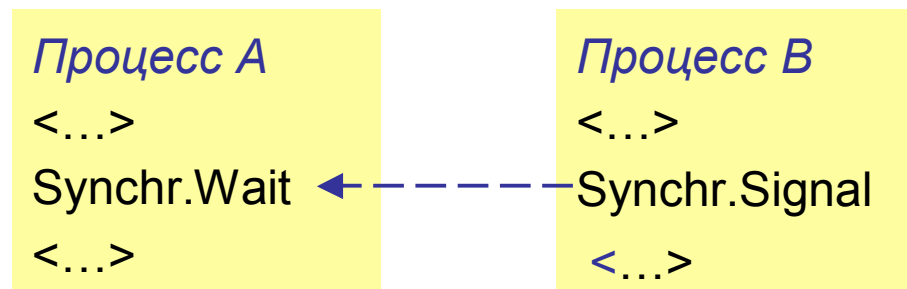
Полная аналогия с
замком

Условная синхронизация

Выполняя операцию Signal, процесс устанавливает флаг условия, а при Wait – ждет установки флага и сбрасывает его

Пример: процесс A ждет, пока B не вычислит некоторые данные:

Synchr = new Semaphore(0)



Управление очередью к семафору

Возможные дисциплины очереди

1. FCFS («сильный» семафор)
 - плюс: гарантированный доступ к критическому интервалу, т.е. нет голодания
 - минус: задержка приоритетных процессов; возможна инверсия приоритетов
2. Недетерминированный выбор («слабый» семафор) – нет гарантии доступа, т.е. опасность голодания
3. Приоритетная очередь: выбор в соответствии с приоритетами процессов

Счетные семафоры

В отличие от замка, счетный семафор S может представлять критический ресурс с несколькими доступными единицами ресурса:

- два принтера в сети
- буфер на N порций данных
- и т.п.

Он инициализируется количеством единиц ресурса и позволяет процессам продолжаться, пока есть доступные единицы

Например, задача «производитель-потребитель» - синхронизация обмена данными через буфер. Производитель записывает порции данных в буфер, а потребитель считывает их, причем скорости процессов непредсказуемы.

Случай циклического буфера с ограниченным числом ячеек на одну порцию:

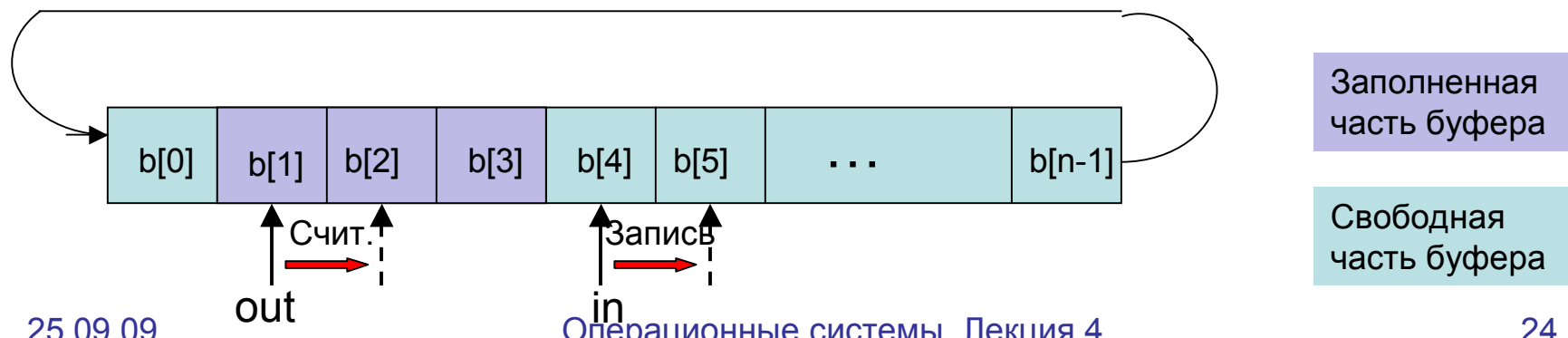
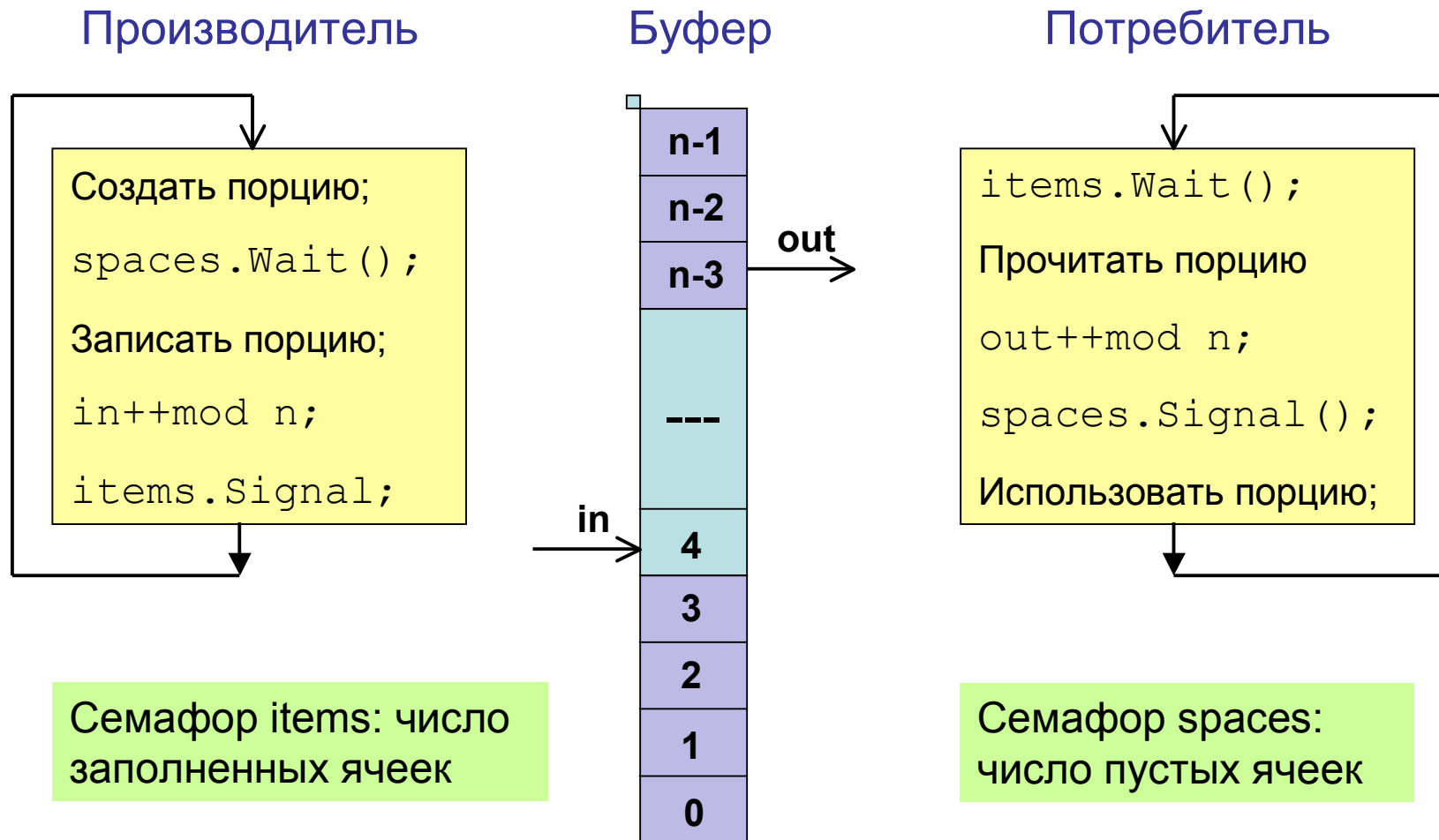


Схема взаимодействия «производитель – потребитель»



Семафор для простого буфера – массива СИМВОЛОВ

```
class Buffer() {  
    static int n = 50;    //размер  
                          //буфера  
    char array buffer[n];  
    int in, out;  
    Semaphore items, spaces;  
public:  
    void insert(char item);  
    void take();  
}
```

```
Buffer::Buffer() {  
    in = 0;  
    out = 0;  
    items = new Semaphore(0);  
    // число порций в буфере  
    spaces = new Semaphore(n);  
    // число свободных мест в  
    буфере  
}
```

```
Buffer::insert(char item) {  
    spaces.Wait();  
    buffer[in] = item;  
    in++mod n;  
    items.Signal();  
}
```

```
char Buffer::take() {  
    char item;  
    items.Wait();  
    item = buffer[out]  
    out++mod n;  
    spaces.Signal();  
    return item  
}
```

Задача «читатели-писатели»

Несколько процессов совместно используют общий файл (таблица, база данных и т.п.)

- процессы-*читатели* читают его – это разрешается несколькими одновременно
- *писатели* же хотят писать в файл – это разрешается только одному за раз, при этом читателем доступ запрещен

Простейшее решение с усиленным ограничением: у каждого процесса - исключительный доступ к файлу:

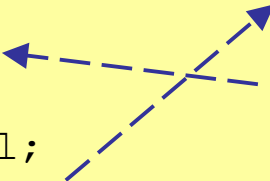
```
rw = new Semaphore(1); //семафор взаимного исключения
//читатель:
while (true) {
    rw.wait;
    читать файл;
    rw.signal;
    другие действия;
}
//писатель:
while (true) {
    rw.wait;
    писать в файл;
    rw.signal;
    другие действия;
}
```

Решение задачи с приоритетом читателей

Ослабим ограничения, чтобы читатели могли работать одновременно:

- Читатели как группа должны блокировать работу писателей
- *первый* читатель должен захватить блокировку взаимного исключения
- *последний* читатель должен снимать ее, закончив работу

```
rw = new Semaphore(1); //семафор взаимного исключения
int nr = 0;             //число активных читателей
//читатель              //писатель
while (true) {          while (true) {
nr++;                   rw.wait;
if (nr == 1) rw.wait;   писать в файл;
читать файл;           rw.signal;
nr--;                  другие действия;
if (nr == 0) rw.signal;
другие действия;      }
}
```



(Здесь есть ошибка: переменная `nr` – критический ресурс для нескольких читателей, который должен быть защищен от одновременного доступа. (Вопрос 5)).

Данный алгоритм дает преимущество читателям: ни один писатель не может начать запись, если есть хоть один ждущий читатель. Значит, при интенсивном поведении читателей писатели будут голодать. (Вопрос 6).

Заключение

1. Независимые процессы взаимодействуют, конкурируя за общие ресурсы, а сотрудничающие (cooperative) процессы, кроме того, обмениваются информацией (коммуникация)
2. Два вида синхронизации взаимодействующих процессов: взаимное исключение доступа к критическому ресурсу и ожидание/сигнализация о событиях или о выполнении условий
3. Программные решения задачи взаимного исключения сложны и неэффективны по времени, но в распределенных системах им нет альтернативы
4. Аппаратной поддержкой взаимного исключения служат атомарные инструкции типа «проверить-установить» и запрет прерываний
5. Большинство ОС содержат низкоуровневые примитивы (функции) синхронизации: замки (мьютексы) и семафоры. Двоичный семафор работает как замок, а счетный может регулировать доступ к группе однородных ресурсов
6. Задачи «производитель-потребитель» и «читатели-писатели» - типичные модели взаимодействующих процессов, используемые в различных контекстах.

Вопросы к лекции 4

1. Предположим, непосредственно перед выполнением одного (любого) из этих двух участков $k = 10$. Сколько (и каких) различных результатов k можно получить после выполнения обоих участков, учитывая все возможные варианты чередования во времени команд этих параллельных процессов?
2. Зачем нужно это ожидание?
3. Недостаток инструкции TS в том, что она при каждом вызове записывает значение в lock, даже если это значение не изменяется (это сильно замедляет выполнение TS в многопроцессорных системах). Как изменить код взаимного исключения для преодоления этого недостатка?
4. Решите задачу взаимного исключения с применением инструкций Exchange и Compare&Swap, описав предварительно их семантику на псевдокоде.
5. Расширьте данное решение на случай многих потребителей и производителей с общим буфером. (Понадобится взаимное исключение работы с буфером – отдельное для потребителей и производителей.)
6. Исправьте эту ошибку.
7. Напишите псевдокод решения задачи «читатели-писатели» с приоритетом писателей: ни один читатель не должен начать чтение, если есть хоть один ждущий писатель. Используйте сильные семафоры. (Не забудьте прокомментировать объявления семафоров и глобальных переменных.)

Подсказка: одна из возможных схем алгоритма – на следующем слайде

Возможная схема решения задачи с приоритетом писателей

