# МЕХАНИЗМЫ СИНХРОНИЗАЦИИ

10

Курс лекций

**«Системное программное обеспечение»**
**«System Software»**
**«Операционные системы»**

для студентов специальностей АСОИ и ИИ

**Павел Кочурко**
доцент кафедры ИИТ, к.т.н.

БрГТУ

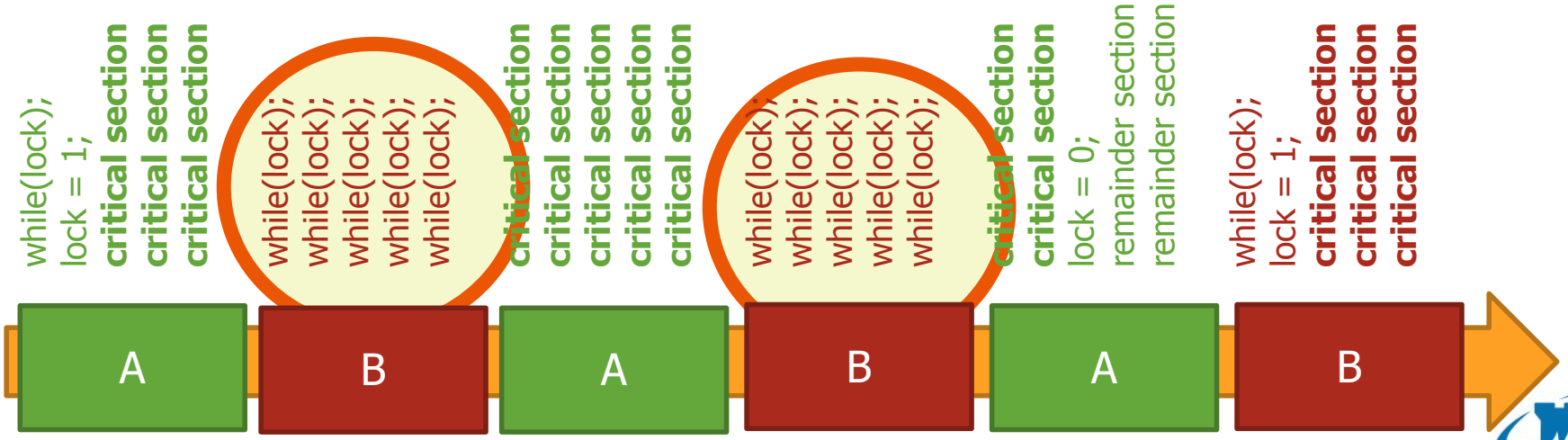
# Недостаток алгоритмов реализации взаимоисключения

**Process A**

```
shared int lock = 0;
while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
      remainder section
}
```

**Process B**

```
shared int lock = 0;
while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
      remainder section
}
```



# Недостаток алгоритмов реализации взаимоисключения. Вывод

- Бесполезная работа в прологе критической секции ☐
- ЦПУ вместо полезных вычислений производит проверку условия, которое изменится только в один из следующих квантов времени ☐
- Существенное снижение КПД ☐
- ОС могла бы следить за тем, когда кого пускать в критическую секцию ☐
- Процессы должны сообщать ОС о входе в критическую секцию и выходе из неё

# Семафоры Дейкстры

**Семафор** – это целая неотрицательная переменная, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P и V.

**P**roberen

P(S): пока S == 0 процесс блокируется;

S = S – 1;

**V**erhogen

V(S): S = S + 1;

**Мьютекс** – двоичный семафор (0 или 1)

# Задача производителя-потребителя

**Producer**

```
while(1) {
   produce_item;
   put_item;
}
```

**Consumer**

```
while(1) {
   get_item;
   consume_item;
}
```

- Буфер размера N
- Ограничения:
  - Одновременно к буферу не могут обращаться никакие два процесса
  - Производитель не может писать в полный буфер, должен ждать освобождения хоть одной ячейки буфера
  - Потребитель не может читать из пустого буфера, должен ждать заполнения хоть одной ячейки буфера

# Решение при помощи семафоров

Semaphore mutex = 1;
Semaphore empty = N;
Semaphore full = 0;

**Producer**

```
while(1) {
   produce_item;

   P(empty);
   P(mutex);
   put_item;
   V(mutex);
   V(full);

}
```

**Consumer**

```
while(1) {

   P(full);
   P(mutex);
   get_item;
   V(mutex);
   V(empty);

   consume_item;
}
```

+: нет бесполезной работы системы по проверке в цикле
-: сложность отслеживания корректности размещения семафоров

# Мониторы Хоара

- **Монитор** – тип данных в ЯВУ (аналог класса в ООП), содержит переменные, определяющие его состояние, и функции-методы.
- В любой момент времени **только один процесс** может быть активен, т. е. находиться в состоянии готовность или исполнение, **внутри данного монитора**
- Условные переменные – переменные монитора, над которыми могут производиться две примитивные операции:
  - **wait** – процесс блокируется на данной переменной
  - **signal** – процесс, заблокированный на данной переменной, разблокируется

# Общая структура монитора

```
monitor monitor_name {
    описание внутренних переменных ;

    void m1(...){...
    }
    void m2(...){...
    }
    ...
    void mn(...){...
    }

    {
      блок инициализации
       внутренних переменных;
    }
}
```

# Решение при помощи мониторов

```
monitor ProducerConsumer {

    condition full, empty;
    int count;

    void put() {
        if(count == N) full.wait;
        put_item();
        count += 1;
        if(count == 1) empty.signal;
    }

    void get() {
        if (count == 0) empty.wait;
        get_item();
        count -= 1;
        if(count == N-1) full.signal;
    }

    {
        count = 0;
    }

}
```

**Producer**:

```
while(1) {
    produce_item;
    ProducerConsumer.put();
}
```

**Consumer**:

```
while(1) {
    ProducerConsumer.get();
    consume_item;
}
```

+: семафоры расставит компилятор, использование монитора простое

-: необходима поддержка мониторов языком/компилятором

# Сообщения

Примитивные операции над сообщениями:

**прямая адресация**

send(P, message)

послать сообщение message процессу P

receive(Q, message)

получить сообщение message от процесса Q

**непрямая адресация**

send(A, message)

послать сообщение message в почтовый ящик A

receive(A, message)

получить сообщение message из почтового ящика A

# Решение при помощи сообщений

**Producer**

```
while(1) {
   produce_item;
   send(buf, item);
}
```

**Consumer**

```
while(1) {
  receive(buf, item);
  consume_item;
}
```

+: встроенный механизм взаимоисключения, тривиальность использования

-: невозможность реализации сложных алгоритмов синхронизации

# ВОПРОСЫ?

http://iit.bstu.by/ss