Продолжение про взаимодействие параллельных процессов.

```
P1: ...
    fl1 = 1;
    do while (fl2 == 1)
    {
        if (que == 2) then
        {
            fl1 = 0;
            do while (que == 2); // цикл ожидания по пер-ной que;
            fl1 = 1;
        }
    }
    CR1;
    fl1 = 0;
    que = 2; // отдаёт очередь др. процессу
    pf1;
end P1
```

Активное ожидание на процессоре – время непроизводительно расходуется на проверку флага другого процесса;

→ введение пер-ной „чья очередь"

Это решение устраняет попадание процессов в тупик и беск. отталкивание

Процесс сразу захватывает свой флаг;

Но это решение для 2 процессов ⇒ цена его невелика

Есть ещё алгоритм Петерсона; (аналогичное для 2 процессов)

race condition (гонки, условия гонок) - процессы пытаются получить доступ к разделяемым пер-ным;

Аппаратная реализация взаимоисключения.

В IBM 370 появилась команда test-and-set;

Она явл. неделимой (атомарной) и реализует как неделимое действие проверку и установку значения в памяти;

Бит блокировки

test-and-set читает значение лог. пер-ной b, копирует его в a, присваивает b значение true;

Использование test-and-set (спец. предусмотренная команда) — аппаратная реализация (простая блокировка у Вахалии, стр. 305);

Классическая аппаратная реализация:

```
program we_testandset;
  fl,c1,c2 : logical;
P1; while(1) do
    {
        c1=1;
        while(c1) do
            testandset(c1,fl);
        Ch1;
        fl=0;
    }
    pr1;
  //end p1;
P2; while (1) do
    {
        c2=1;
        while(c2) do
            test and set(c2,fl);//(критическая блокировка - spin lock, simple lock, simple mutex)
        Ch2;
        fl=0;
        pr2;
    }
//end P2;

void main()
{
  fl=0;
```

```
parbeg;n
    P1; P2;
parent;
}
```

fl = true, когда любой из процессов находится в своей критической участке (initial false);

Пусть P1 хочет войти в свой крит. участок, а P2 уже находится в своём критическом участке. P1 устанавливает пер-ную C1 и заходит в цикл проверки флага 2-го процесса (fl) командой test_and_set;

Т.к. P2 находится в своём критическом участке, fl = 1. Команда test_and_set обнаруживает это и устанавливает C1

В рез-те P1 наход. в своём цикле активного ожидания до тех пор, пока P2 не выйдет из своего критического участка и не сбросит fl;

Считается, что данный способ аппаратной реализации взаимоисключения не приведёт к бесконечному откладыванию. Оно возможно, но его вероятность очень мала;

Считается, что когда процесс выходит из своего критического участка и сбрасывает флаг, то, скорее всего, другой процесс сможет перехватить инициативу и установить свой флаг;

test_and_set активно используется в ядре ОС;

Использование test_and_set в цикле проверки значения пер-ной, наз. циклической блокировкой (spin lock);

```
void spin_lock(spin_lock_t *c)        → condition
{
    while (test-and-set(*c) != 0)
        /* ресурс занят */
}

void spin_unlock (spin_lock_t *c)
{
    *c = 0;
}
```

Реализация команды test-and-set во многих архитектурах связана с блокировкой шины памяти;

Врем-тедлительные циклы ожидания могут привести к занятию шины одним потоком на длительное время
⟹ может понизить отзывчивость ОС;

Решение проблемы: 2 вложенных цикла;

```
void spin_lock(spin_lock_t *c)
{
    while (test_and_set(*c) != 0)
        while (*c != 0);
}
```

если пер-ная занята, то начинает выполняться обычный цикл проверки без захвата шины данных;

Использование команды test_and_set для реализации взаимоисключения наз. аппаратной реализацией

ЭКЗАМЕН!

Для всех рассм. вариантов характерно наличие цикла ожидания (активного ожид-я в цикле проверки значения пер-ной). Это неэффективное использование процессорного времени.

Семафоры.

1965 – Дейкстра (Dijkstra E.W.) предложил использовать семафоры как средство взаимоисключения;

Семафор Дейкстры – неотрицательная защищённая пер-ная (S), на которой были определены две неделимые операции

P(S)
↳ passeren (пропустить)

V(S)
↳ vrijgeven (освободить);

P(S) уменьшает значение семафора S на 1;

P(S): S = S-1, если S>0;
если S=0, то процесс блокируется до тех пор, пока движение не станет возможным;

V(S): S = S+1 ⇒ разблокировка процесса, ожидающего освобождения семафора;

К семафору м.б. организована очередь: когда он будет освобождён, его сможет захватить 1-й процесс из очереди;

если на семафоре опр. числа 0 и 1, то семафор бинарный, а если целые положительные числа — считающий;

Семафор освобождает другой процесс;

В чём суть инновации:
Если мы видим, что процесс не может захватить семафор, то он блокируется ⇒
⇒ нет активного ожидания (т.к. не расх. процессорное время). Другой процесс, который освободит семафор, разблокирует процесс, и он сможет захватить семафор.

И блокировка, и разблокировка возможны только ядром ОС посредством сист. вызовов, т.е. переходом в режим ядра, т.е. все действия выполняются ядром ОС. Это (сами переключения) затратно, т.е. мы минимизировали активное ожидание, но пришли к сист. вызовам и переходу в режим ядра;

P_1 :	P_2 :	Нач. знач. $S=1$
... P(S); CR1; V(S); ...	... P(S); CR1; V(S); ...	

Пример использования считающего семафора:
Решение задачи производства – потребления (producer – consumer) на 3 семафорах (предложено Дейкстрой);
Имеется буфер размерами N и 2 типа процессов:

Производитель может только производить и класть в буфер единицу данных, т.е. заполнять ячейки буфера;

Потребитель может только выбирать данные из буфера;

3 семафора: 2 считающих и 1 бинарный:
se – empty (число пустых ячеек буфера);
sf – full (число заполненных ячеек буфера);
sb – binary;

```
program semaphore
        sc, sf, sb : int;
producer: while(1)
          {
                создать запись;
                P(se);
                P(sb);
                n = n + 1;
                V(sb);
                V(sf);
          }
```

```
consumer: while(1)
          {
                P(sf);
                P(sb);
                n = n - 1;
                V(sb);
                V(se)
          }

начало
        se = n; sf = 0; sb = 1;
```

Задача производства-потребления - задача, в которой не только осуществляется взаимоисключение процессов (обеспечивается монопольный доступ к разделяемому ресурсу, которым явл. буфер), но и реализуется синхронизация процессов;

Producer ничего не сможет положить в буфер, если нет свободных ячеек (т.е. если se=0). На операции P(se) Producer будет блокирован до тех пор, пока consumer не освободит ячейку буфера. Освободив ячейку буфера, consumer инкрементирует se;

Аналогично, consumer не сможет ничего взять из буфера, если буфер пуст. Он будет блокирован на sf до тех пор, пока producer не положит единицу данных в буфер. При этом он инкрементирует sf.

Процессы явл. асинхронными, т.е. выполняются с собственной скоростью

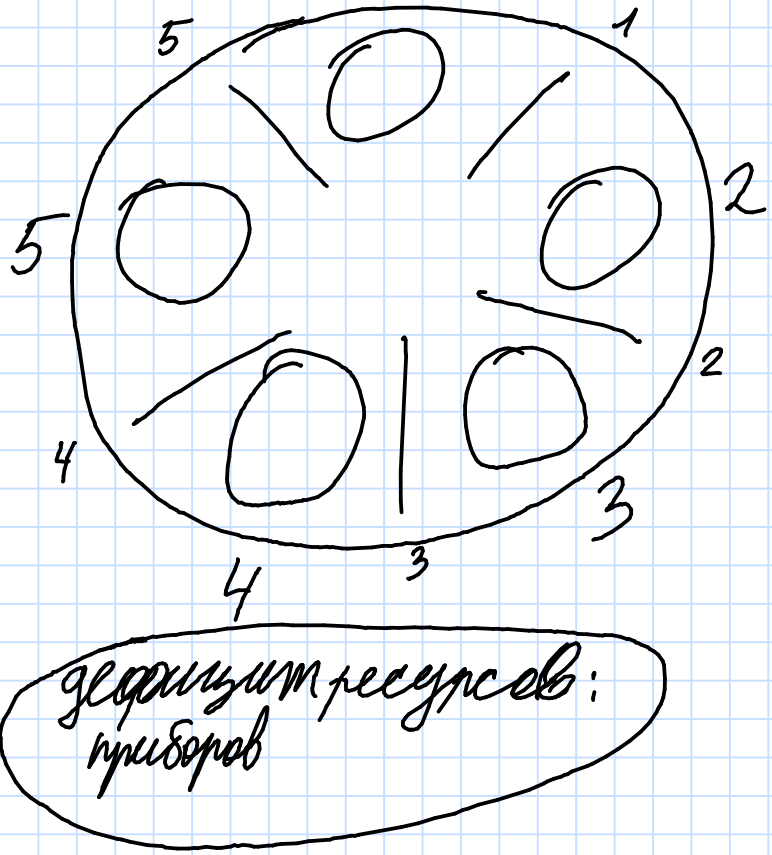
Нельзя предсказать, когда какой процесс придёт в опр. точку

Простое взаимоисключение само по себе не предполагает синхронизацию;

В совр. ОС UNIX/Linux реализованы наборы семафоров, наиболее общих, многоместными семафорами;

Рассм. возм-ти симафорных семафоров.

Задача Дейкстры об обедающих философах:
На столе стоит 5 тарелок, между тарелками по одному прибору;



дерадлок ресурсов: приборов

3 действия философов:
1) каждый сразу пытается взять обе вилки; если удалось, то они едят и кладут сразу обе вилки (возвр. сразу оба ресурса);
2) каждый пытается взять правую, удерживает её и пытается взять левую;
3) каждый пытается взять правую, удерживает её и пытается взять левую; если не получилось взять левую, то кладёт правую (возвращает ресурс);

К чему это приведёт?
Сколько философов умрёт голодной смертью при подходе 1? (бесконечное откладывание)

Эту задачу записывают с помощью множественных семафоров:

```
program example5ft; //1-й сценарий
var
    forks: array[1..5] of semaphore;
    left, right, i: 1..5;
      ---
P1:  ---
    left = (i+1) mod 5; right = i mod 5;
    while(1) do
    begin
        <размышляет>
        P(fork[left], fork[right]);
        <ест>
        V(fork[left], fork[right]);
    end;
//P1
      ---
P5:  ---
    left = 4, right = 5;
    while(1) do
    begin
        <размышляет>
        P(fork[left], fork[right]);
        <ест>
        V(fork[left], fork[right]);
    end;
//P5
```

Св-во набора семафоров: одной неделимой операцией (=) м.б. изменено значение всех или

части семафоров набора.

Почему были введены наборы семафоров и почему на них определено именно такое правило?

Это решило большую проблему для 1965 года — активное ожидание на процессоре, была решена задача взаимоисключения (обеспечения монопольного доступа).
Но в результате получилось так, что чем больше инструментов получает программист, тем больше у него возникает проблем с ними. Использование в программах большого числа разных семафоров, обеспечивающих доступ к разным критическим секциям, приводило к их частому пересечению. Стало возможным попадание процессов в тупик.

Пример:

P1:	P2;
...	...
P(S1)	P(S2);
P(S2)	P(S1);
Cr1;	Cr2;
V(S2);	V(S1);
V(S1);	V(S2);

Два процесса в разном порядке захватывают семафоры.

Отловить такое очень сложно;
Чтобы структурировать всё, были введены наборы семафоров;