

# Разбор первой лекторской контрольной по КТ

## Предисловие:

Для успешного написания контрольной работы рекомендую полностью посмотреть три лекции №3, №6, №7.

Во третьей лекции будет рассказано про алгоритмы планирования в частности про: FCFS (First-Come, First-Served/первым пришёл, первым обслужен), RR (Round Robin), SJF (Shortest-Job-First), понимание работы этих алгоритмов поможет решить первое задание. Третью лекцию можно начинать смотреть примерно с 30 минуты. Но первое задание не стоило бы 14 балов, если бы оно было так просто, поэтому для его решения нам так же понадобится лекция №7 и понимание что такие схемы управления памятью; в ней будут объяснены понятия: first fit, best fit, worst fit. Эту лекцию тоже рекомендую смотреть с 30 минуты для хоть какого-то понимания, но конкретно про методы будет рассказано примерно на 41 минуте.

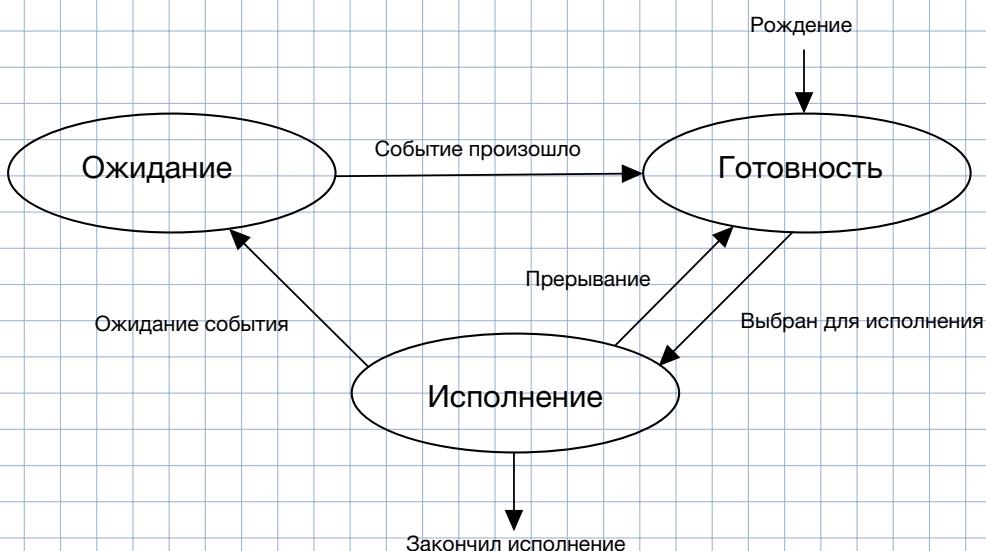
Лекция №6 потребуется для решения второй задачи, рекомендую посмотреть полностью, дабы проникнуться.

И ещё раз нам потребуется лекция №7, дабы совладать с задачей 3. Так же есть 12 минутное видео с разбором задачи прошлого года, которая очень похожа на нынешнюю. Его я так же приложу к этому конспекту.

Ну а четвёртое задание решается всеми любимым методом `ctrl+f`, `ctrl+c`, `ctrl+v`. Вся информация имеется в Карпове/Конькове.

## Теория для первого задания:

Для более эффективного и справедливого распределения ресурсов компьютера используются различные механизмы прерывания процессов. Но все они основаны на том, что раз в какое-то фиксированное число квантов времени планировщик принимает решение переключаться ли с одного процесса на другой по какому-то заданному программистом правилу или нет.



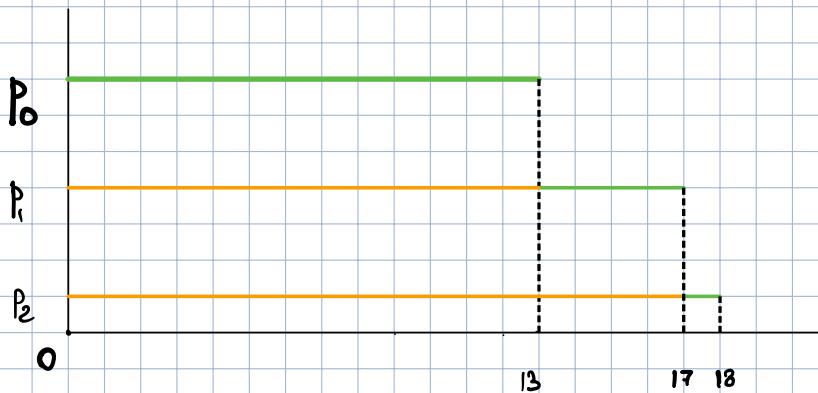
Рассмотрим алгоритмы планирования:

### 1. FCFS (First-Come, First-Served)

Тут все просто: какой процесс пришел первым, тот и будет выполняться пока его CPU burst (время, которое исполняется процесс) не закончится.

Пример:

Процесс	$P_0$	$P_1$	$P_2$
CPU burst	13	4	1



Время ожидания, как не сложно догадаться, — это сумма времени ожидания всех процессов:

$$t_{\text{ожид}} = \frac{13 + 17}{3} = 10$$

$$t_{\text{все}} = \frac{13 + 13 + 4 + 13 + 4 + 11}{3} = 16 \quad \text{— ожидание и исполнение}$$

## 2. Round Robin

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии *готовность*, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, рас-

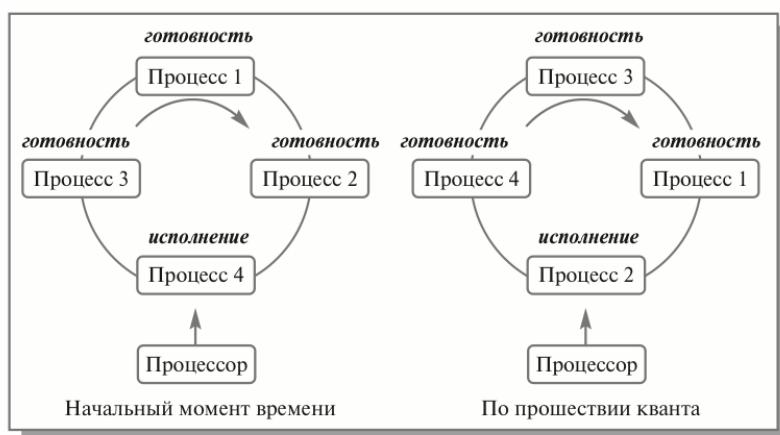


Рис. 3.4. Процессы на карусели

положенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта:

- Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
- Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

### 3. Shortest-Job-First

Всегда берет на исполнение процесс с наименьшим ср. burst.

#### Схемы управления памятью:

- Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.
- Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
- Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

1. (14 баллов) Рассмотрим однопроцессорную вычислительную систему с объемом оперативной памяти 200 МВ, в которой используется схема организации памяти с динамическими (переменными) разделами. Для долгосрочного планирования процессов в ней применен алгоритм FCFS. В систему поступают пять заданий с различной длительностью и различным объемом занимаемой памяти по следующей схеме:

Номер задания	Момент поступления в очередь заданий	Время исполнения (CPU burst)	Объем занимаемой памяти
1	0	3	80 МВ
2	2	4	50 МВ
3	3	5	60 МВ
4	4	2	80 МВ
5	5	1	10 МВ

Вычислите среднее время между стартом задания и его завершением (turnaround time) и среднее время ожидания (waiting time) для следующих комбинаций алгоритмов краткосрочного планирования и стратегий размещения процессов в памяти:

- a) RR (Round Robin) и worst fit (наименее подходящий);
- b) RR и best fit (наиболее подходящий);
- c) FCFS (First Come First Served) и worst fit;
- d) FCFS и best fit.

При вычислениях считать, что процессы не совершают операций ввода-вывода, величину кванта времени принять равной 2. Временами переключения контекста, рождения процессов и работы алгоритмов планирования пренебречь. Освобождение памяти, занятой процессами, происходит немедленно по истечении их CPU burst. Краткосрочное планирование осуществляется после рождения новых процессов в текущий момент времени. Для алгоритма RR принять, что родившиеся процессы добавляются в САМЫЙ конец очереди готовых процессов (ПОСЛЕ процесса, перешедшего в состояние готовность из состояния исполнение в это время).

Рассмотрим пункт а:

$$t_{avg} = \frac{0+3+7+2+4}{5} = \frac{16}{5} = 3,2$$

$$t_{wq} = \frac{3+7+12+4+5}{5} = \frac{31}{5} = 6,2$$

(1) (2)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14



Планшет

200 MB



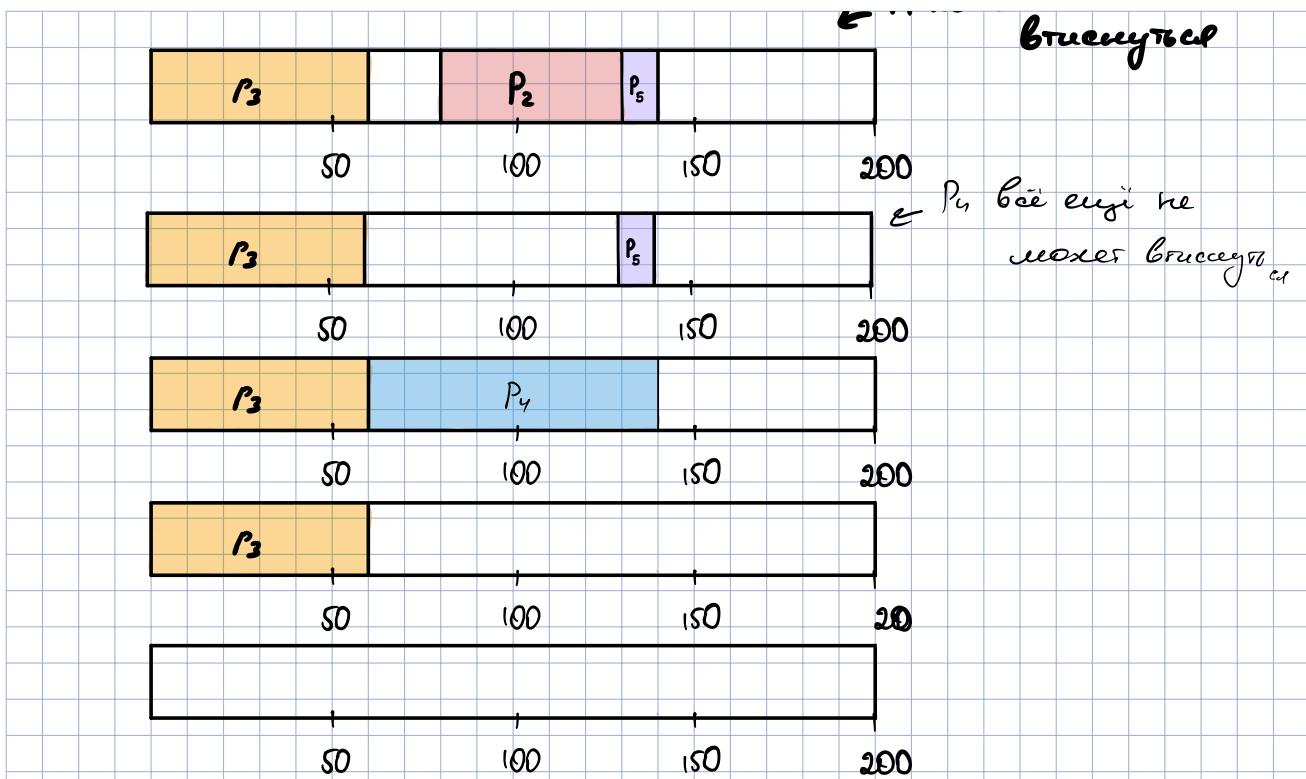
80

100

150

200

Р1 не использован



(1) — обратим внимание, что в момент времени "2" мы отдаём исполнение процессу  $P_1$  т.к. в условии задачи нам сказано, что для RR родившийся процесс добавляется в САМЫЙ конец очереди (см последнюю строку условия)

(2) — обратим внимание, что в алгоритме работы RR сказано, что после окончания выполнения процесса таймер начинается заново

(3) — т.к. мы использовали алгоритм worst fit, то процессу просто не хватило ОП, чтобы родиться, поэтому мы не отмечаем состояние готовности

Рассмотрим путь  $b$ :

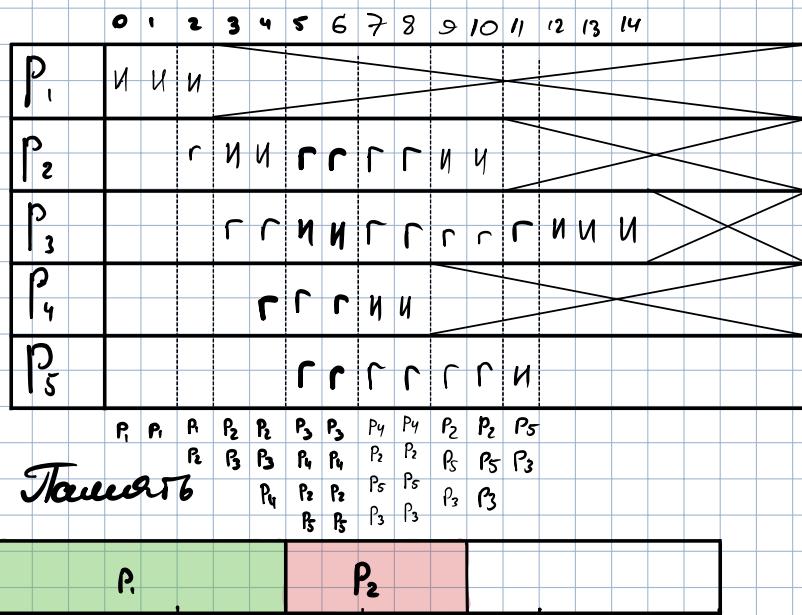
1. (14 баллов) Рассмотрим однопроцессорную вычислительную систему с объемом оперативной памяти 200 МВ, в которой используется схема организации памяти с динамическими (переменными) разделами. Для долгосрочного планирования процессов в ней применен алгоритм FCFS. В систему поступают пять заданий с различной длительностью и различным объемом занимаемой памяти по следующей схеме:

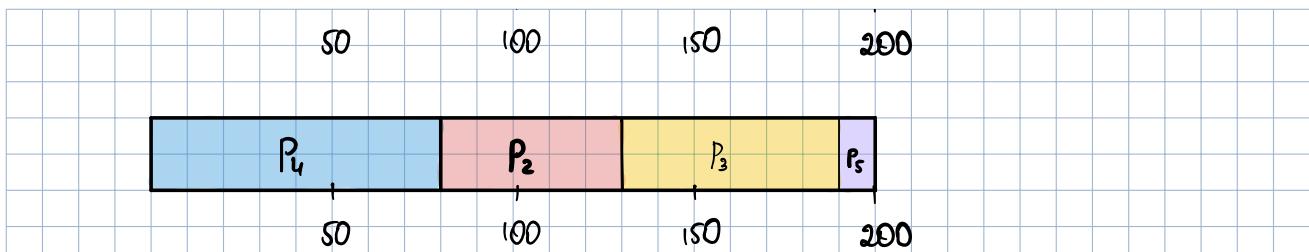
Номер задания	Момент поступления в очередь заданий	Время исполнения (CPU burst)	Объем занимаемой памяти
1	0	3	80 MB
2	2	4	50 MB
3	3	5	60 MB
4	4	2	80 MB
5	5	1	10 MB

Вычислите среднее время между стартом задания и его завершением (turnaround time) и среднее время ожидания (waiting time) для следующих комбинаций алгоритмов краткосрочного планирования и стратегий размещения процессов в памяти:

- a) RR (Round Robin) и worst fit (наименее подходящий);
  - b) RR и best fit (наиболее подходящий);
  - c) FCFS (First Come First Served) и worst fit;
  - d) FCFS и best fit.

При вычислениях считать, что процессы не совершают операций ввода-вывода, величину кванта времени принять равной 2. Временами переключения контекста, рождения процессов и работы алгоритмов планирования пренебречь. Освобождение памяти, занятой процессами, происходит немедленно по истечении их CPU burst. Краткосрочное планирование осуществляется после рождения новых процессов в текущий момент времени. Для алгоритма RR принять, что родившиеся процессы добавляются в **САМЫЙ** конец очереди готовых процессов (**ПОСЛЕ** процесса, перешедшего в состояние **готовность** из состояния **исполнение** в это время).





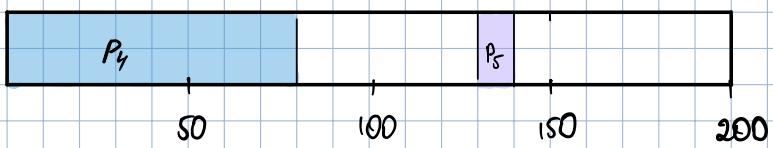
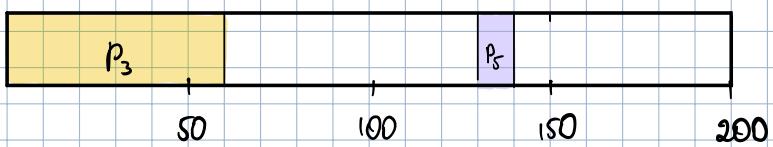
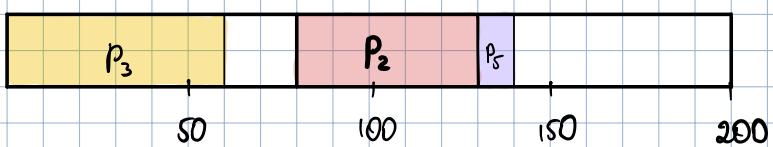
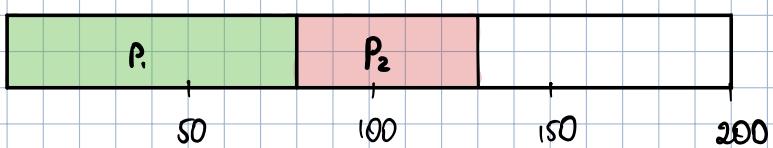
Рассмотрим путь C:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

P <sub>1</sub>	и и и
P <sub>2</sub>	г и и и
P <sub>3</sub>	г г г г и и и
P <sub>4</sub>	х х х х х х х <b>т</b> и и
P <sub>5</sub>	г г г г г г г

P<sub>1</sub> P<sub>1</sub> P<sub>1</sub> P<sub>2</sub> P<sub>2</sub> P<sub>2</sub> P<sub>3</sub> P<sub>3</sub> P<sub>3</sub> P<sub>3</sub> P<sub>3</sub> P<sub>3</sub> P<sub>3</sub> P<sub>4</sub> P<sub>4</sub> P<sub>4</sub> P<sub>4</sub> P<sub>4</sub>

Паспорта: P<sub>3</sub> P<sub>5</sub> P<sub>3</sub> P<sub>5</sub>

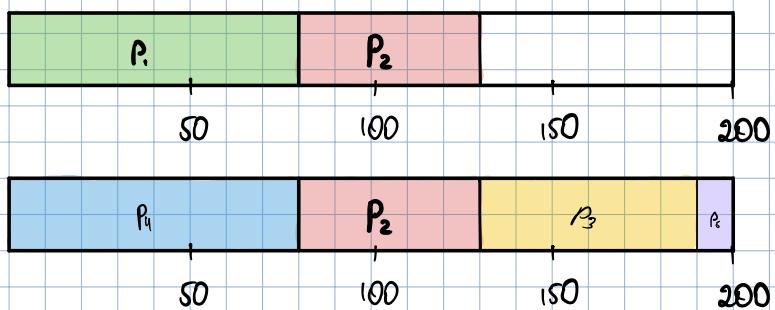


Рассмотрим пучок:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

P <sub>1</sub>	и и и
P <sub>2</sub>	г и и и
P <sub>3</sub>	г г г г и и и
P <sub>4</sub>	г г г г г г г и
P <sub>5</sub>	г г г г г г г г и

Паша:



Поехали с второго задания.

Зачем нужны механизмы синхронизации? Затем, что процессы корректно работали и их ресурсы не пересекались, приводя к неверному выполнению программы. Одним из таких механизмов является семафор. Семафор — целая разделяемая переменная с неотрицательными значениями. Над семафором можно проделывать следующие действия:

P (S): если S == 0, то процесс блокируется,  
иначе S--;

V(S): S++;

При создании может быть инициализирована любым неотрицательным числом.

Используется при решении задачи потребитель-производитель.  
Производитель производит объекты, а потребитель их потребляет и необходимо передать объекты от одного к другому. Плюс надо учесть, что потребитель не может потреблять, когда конвейер пуст, а производитель не может производить, когда конвейер полон.

```
Semaphore mutex = 1;
Semaphore empty = N; /* где N - емкость буфера*/
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;
}
```

Тут семафоры использовались для двух целей: организации взаимоисключения на критическом участке и для синхронизации работы процессов.

## Разберём задачу на семафоры:

2. (12 баллов) В диком каннибальском племени вокруг котла с пищей спят дикари и повар. Изначально в кotle находится  $N$  порций мяса. Дикари по очереди просыпаются, берут из котла порцию мяса, съедают его и засыпают снова. Дикарь, не обнаруживший мяса в котле, будит повара. Повар находит добычу и снова готовит  $N$  порций, не подпуская никого к котлу во время приготовления, после чего тоже засыпает. Используя семафоры Дейкстры и разделяемые переменные, постройте корректную модель происходящего, описав поведение каждого из дикарей и повара с помощью отдельных процессов.

```
int meat = N;  
sem full(N), empty(0);
```

Cook:

```
while (true)  
{  
    P(empty);  
    P(mutex);  
    meat = N;  
    V(mutex);  
    for (i = 0; i < N; ++i)  
    {  
        V(full);  
    }  
}
```

Man:

```
while (true)  
{  
    sleep();  
    P(mutex);  
    if (meat == 0)  
    {  
        V(empty);  
    }  
    V(mutex);  
    P(full);  
    P(mutex);  
    --meat;  
    V(mutex);  
}
```

## Мониторы:

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {  
    описание внутренних переменных ;  
  
    void m1(...){...}  
}  
void m2(...){...}
```

```
    }
    ...
    void m_n(...){...}
}

{
    блок инициализации внутренних переменных ;
}
}
```

времени только один процесс может быть активен, т. е. находиться в состоянии *готовность* или *исполнение*, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам *full* и *empty* в предыдущем примере. Для этого в мониторах было введено понятие *условных переменных* (*condition variables*)\*, над которыми можно совершать две операции *wait* и *signal*, отчасти похожие на операции *P* и *V* над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию *wait* над какой-либо условной переменной. При этом процесс, выполнивший операцию *wait*, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию *signal* над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов ложились

операции `signal` для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции `signal`. Мы будем придерживаться подхода Хансена.

**Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция `signal` всегда должна выполняться после операции `wait`. Если операция `signal` выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции `wait` всегда будет приводить к блокированию процесса.**

Давайте применим концепцию мониторов к решению задачи производитель-потребитель:

```
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;  
}
```

## Сообщение:

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

send(P, message) – послать сообщение message процессу P;  
receive(Q, message) – получить сообщение message от процесса Q.

В случае непрямой адресации мы будем обозначать их так:

send(A, message) – послать сообщение message в почтовый ящик A;  
receive(A, message) – получить сообщение message из почтового ящика A.

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

Producer:

```
while ()  
{  
    produce_item();  
    send(address, item);  
}
```

Consumer:

```
while ()  
{  
    receive(address, item);  
    consume_item();  
}
```

## Разбор:

2. (12 баллов) В орлином гнезде находятся орлята. Птенцы едят из общей миски, где вначале находится N порций еды. Каждый птенец берет порцию еды из миски, съедает ее, затем спит, затем снова лезет за едой. Еду из миски может одновременно брать только один птенец. Орленок, съевший последнюю порцию, зовет мать. Мать приносит в клюве очередные N порций еды, кладет в миску и снова улетает, после чего птенцы могут снова брать пищу. Используя классические очереди сообщений и разделяемые переменные, постройте корректную модель происходящего, описав поведение каждого из птенцов и матери с помощью отдельных процессов. Классические очереди сообщений используют примитивы `send(A, message)` и `receive(A, message)`, где A - имя очереди сообщений.

Parent:

```
while (1)
```

```
{ message = N;  
  send (A, message);  
  receive (C, message);  
  assert (message == 0);  
}  
kill (0, 9);
```

Chick:

```
while (1)
```

```
{ receive (A, message);  
  message --;  
  if (message > 0)  
    send (A, message);  
  else  
    send (C, message);  
  sleep();  
}
```

Очередь работает пришумто так: если канал-го  
очереди свободен и канал занят ее первым  
засыпает с того момента и начинает работать  
программа.

# Задача №3

## Рассмотрим две варианты

### задачи:

3. (6 баллов) В вычислительной системе с сегментно-страничной организацией памяти и 32-х битовым адресом максимальный размер сегмента составляет 4 Mb, а размер страницы памяти 512 Kb. Для некоторого процесса в этой системе таблица сегментов имеет вид:

Номер сегмента	Длина сегмента
0	0x180000
1	0x080000

Таблицы страниц, находящихся в памяти, для сегментов 0 и 1 приведены ниже:

Сегмент 0

Номер страницы	Номер кадра (десятичный)
0	18
3	0

Сегмент 1

Номер страницы	Номер кадра (десятичный)
0	32
1	63

Каким физическим адресам соответствуют логические адреса: 0x000f0236, 0x00470111, 0x00502005?

Адрес в рамках сегмента — это 32 битное поле, разбитое на три области: номер сегмента, номер страницы и на область со следующими внутри страницами. Определены разные задачи областей.

$$4 \text{ Mb} = 2^{22} \text{ байт}, \text{ т.е. на номер}$$

Следующая битка отображается  $32 - 22 = 10$  бит.

Остальные 22 бита на следующем  
шаге мы сдвигаем.

$$512 \text{ KБ} = 2^{19} \text{ байт}$$

$22 - 19 = 3$  бита на конец страницы

а) Переведем адрес страницы:

0000 0000 0.000. | 111. 0000.0010, 0011. 0110  
сдвиг 0 |  $\uparrow^{\text{б}}_1$  170236 - в этот сдвиг

но б 0 сдвигает на страницу №1 -  
→ page fault

б) 0000. 0000. 0100. 0111. 0000. 0001. 0001. 0001  
сдвиги  $\uparrow^{\text{б}}_1$  |  $\uparrow^{\text{б}}_2$

Такой страницы нет, поэтому адрес  
страницы корректный, хотя 32 переведется  
в 10-ричную

$$\begin{array}{r} 32 = 0 \times 20 \\ \hline 16 \end{array}$$

Далее полученный адрес берется  
следующим, для которой

переводится в 16 разрядов следующ

0x00070236  
16

И сдвигается к концу  
кода в 16 разряд, убиваясь  
на 19 бит. Это соответствует  
логической переводе от  
максимальной сдвигаемой  
сравнительной константы  
сдвигом на 16 разрядов.

0x00070236 + 0x200000 =

0x00070236 + 0x01000000 = 0x01070236

3. (3 балла) В вычислительной системе со страницной организацией памяти и 32-битным адресом размер страницы составляет 2 МБ. Для некоторого процесса таблица страниц в этой системе имеет вид:

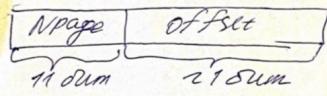
Номер страницы	Адрес начала страницы
1	0x10000000
2	0x00200000
3	0x00600000
4	0x00000000

Каким физическим адресам соответствуют логические адреса: 0x00A03222, 0x00827332, 0x00276543?

3. (9 баллов) Ответьте на следующие вопросы:
- Что такое критическая секция процесса?
  - Что понимается под термином «внешняя фрагментация» в различных схемах выделения процессам оперативной памяти? В каких известных вам схемах она возникает?
  - Что является основной причиной появления мультипрограммных вычислительных систем?

н.3) 32-битная адресация  
размер страницы  $2^15$   $\text{Б} = 2 \cdot 2^{10} \text{Б} = 2 \cdot 2^{20} \text{Б} = 2^{21} \text{Б}$

↗ 15-бит адреса  
идет на следующее  
блоки страниц



1)  $0x00A03222$

0000.0000.101.0.0000.0011.0010.00100010  
 $Npage = 5$

↗  
Page Fault, тк вело к остановке

2)  $0x00827332$

0000.0000.1000.0010.0111.0011.0010  
 $Npage = 4$        $offset_{16} = 27332_{16}$

$Offset = 0x00027332$   
Адрес страницы:  
 $0x00000000$

Физический адрес:  
 $0x00027332$

3)

$0x00276543$

0000.0000.001.0.0111.0110.0101.0100.0011  
 $Npage = 1$        $offset_{16} = 76543_{16}$

$Offset = 0x00076543$

Адрес начала 1й стр.  
 $0x10000000$

Физ. адрес  
 $0x10076543$