## Проблемы распределения ресурсов ОС

Многие задачи взаимодействия процессов можно свести к *задачам борьбы за ресурсы ЭВМ*. В частности, таким ресурсом может быть память ЭВМ или, напри- мер, очередь процессов на печать.

В борьбе за общий разделяемый ресурс, процессы входят в *состояние состя- зания*.

**Состояние состязания** - *ситуация*, в которой два или более процессов счи- тывают или записывают данные *одновременно*, а результат зависит от того, *кто из них был первым*.

Основной способ решения этих проблем - *запрет* на одновременное исполь- зование разделяемого ресурса или *взаимное исключение*.

**Взаимное исключение** предполагает, что одновременно не могут выполнять- ся две *критические секции* разных процессов.

**Критической секцией** (*критической областью*) называется *часть кода про- цесса*, который обеспечивает доступ к разделяемым ресурсам.

Чтобы сформулировать общую проблематику использования любых разделя- емых ресурсов, были сформулированы *четыре необходимых условия*:

* два процесса *не должны находиться* в критических областях;
* в программе *не должно быть предположений* о скорости или количестве процессов;
* процесс, находящийся в критической области, *не может блокировать* другие процессы;
* невозможна ситуация, когда процесс *вечно ждет попадания* в критическую область.

Среди известных способов, реализующих *взаимное исключение*, можно выде- лить следующие:

* *запрет всех прерываний*, когда процесс вошел в критическую область;
* *переменные блокировки*, когда процесс, прежде чем войти в критическую об- ласть, использует значение переменной, которая доступна всем процессам.

Если значение переменной *равно 0,* то процесс изменяет ее значение на 1 и входит в критическую область, а если значение переменной *равно 1*, то про- цесс ждет, пока ее значение не *изменится на 0*;

* *строгое чередование*, когда процесс имеющий и использующий критическую область становится в очередь и получает номер этой очереди, в цикле про-

веряя некоторую общую переменную, значение которой указывает номер оче- реди с разрешенным входом в критическую область. Когда процесс выходит из критической области, он изменяет значение этой переменной, тем самым, разрешая другому процессу войти в критическую область.

* *алгоритм Петерсона*, который использует две функции *enter\_region(int process)* и *leave\_region(int process)*. Вызов первой, из этих функций, является

обязательным условием, при входе процесса в критическую область. Она так- же сохраняет дополнительные данные о процессах, вызвавших ее, поэтому только один процесс может войти в критическую область, а другому необ- ходимо ждать, когда выходящий из критической области процесс вызовет функцию *leave\_region(int process)*, освобождающую доступ к нужной облас- ти.

* *команда TSL* (*Test and Set Lock — проверить и заблокировать*), которая на аппаратном уровне гарантирует неделимость операций чтения и записи слова

памяти, обеспечивая однозначность отражения блокировки процессом разде- ляемого ресурса.

Не вдаваясь в детали возможных реализаций приведенных выше алгоритмов, мы рассмотрим пример *простейшей проблемы* разделения общих ресурсов.

**Проблема производителя и потребителя**, которая соотвествует взаимодей- ствию двух процессов, использующих *общий буфер данных* (*проблема ограничен- ного буфера*).

Более конкретно:

* *один* из процессов, *производитель*, помещает данные в буфер;
* *другой* процесс, *потребитель*, читает данные из буфера;
* *трудности начинаются*, когда производитель хочет поместить данные в бу- фер, а он уже заполнился; тогда производитель должен ждать, когда второй

процесс, полностью или частично, не освободит буфер;

* *аналогично*, когда потребитель хочет прочитать данные из буфера, а он пуст, процесс уходит в ожидание, пока другой процесс не положит данные в буфер и не разбудит потребителя.

**Замечание** Указанная задача очень часто встречается даже в прикладном программировании, не го- воря уже о системном, и требует *разработки примитивов*, обеспечивающих такое меж- процессное взаимодействие.

Ситуация значительно осложняется, когда процессу необходимо захватить

несколько разделяемых ресурсов. В этом случае, возникает *состояние взаимной блокировки*.

**Состояние взаимной блокировки** возникает тогда, когда процесс успел зах- ватить только часть разделяемых ресурсов, необходимых ему для выполнения рабо- ты, а другую часть захватили другие процессы. В такой ситуации, все процессы переходят в *состояние бесконечного ожидания* необходимых им ресурсов.

Примером задачи, приводящей к взаимным блокировкам, является «*Задача об обедающих философах*», которую мы подробно рассмотрим в подразделе 1.5. Здесь же отметим, что в общем случае, необходимы дополнительные системные средства, обеспечивающие процессы инструментами надежной фиксации фактов входа в критические области и выхода из них.

Одним из первых системных средств, обеспечивающих программистов на- дежными инструментами работы с взаимодействующими процессами, стал пакет, имеющий название *System V IPC*. В последующем, его идеи были положены в сред- ства сетевого взаимодействия ЭВМ.

**Замечание** Следует хорошо осознать и запомнить, что все проблемы синхронизации процессов не могут быть решены алгоритмически *только на уровне отдельного процесса*. Это свя- зано с тем, что алгоритм работы процесса прерывается ядром ОС, а захват нескольких нужных ресурсов процесса может осуществляться только некоторой последователь- ностью команд (системных вызовов), которые могут не завершиться до остановки про- цесса и передачи управления другому процессу.

## Системный пакет IPC

С целью обеспечить системных и прикладных программистов надежным инструментом, устранения проблем взаимодействующих процессов, было разрабо- тано несколько системных средств ядра ОС, известных под названием *System V IPC*.

**IPC** (*Inter Process Communication*) — ориентирован на решение трех проб-

лем:

* *надежная и простая передача данных и сообщений* от одного процесса дру- гому;
* *контроль над деятельностью* процессов в критических ситуациях;
* *согласование действий* между процессами.

Непосредственная реализация пакета разделена на *четыре части*:

* *средства адресации IPC*
* *семафоры*,
* *разделяемая память* и *очереди сообщений*

Создавая пакет IPC, разработчики сразу же предусмотрели возможность асинхронного взаимодействия процессов. Для этих целей была разработана систе- ма адресации, которой должен пользоваться каждый процесс, использующий сред- ства пакета IPC.

Необходимость такой адресации обусловлена тем, что *процессы должны раз- личать* разные виды взаимодействия, обусловленные их алгоритмами работы и прикладной направленностью самих процессов.

В качестве информационной основы формирования таких адресов исполь- зуются следующие два объекта:

* *имена файлов*, доступные в файловой системе ОС;
* *произвольное целое число*, отличное от нуля, интерпретируемое как номер проекта.

В случае удачного завершения вызова возвращается значение созданного ключа типа *key\_t*. При ошибке возвращается *-1*, а в переменную *errno* записывается код ошибки, согласно системному вызову *stat*(2).

**Замечание** Тип данных *key\_t* обычно представляет собой 32-битовое знаковое целое — возвращае- мое значение, которое — одинаково для всех имен, указывающих на один и тот же файл, при одинаковом значении аргументов *proj\_id* и *pathname*. Возвращаемые значения функ- ции должны различаться, когда одновременно существующие файлы или идентификато- ры проекта различаются.

Ядро ОС хранит информацию обо всех средствах System V IPC, используе- мых в системе вне контекста пользовательских процессов. В прикладном плане, ключ, генерируемый функией *ftok(...)*, рассматривается как *средство связи* между взаимодействующими процессами.

При создании нового средства связи или получении доступа к уже существу- ющему, процесс получает неотрицательное целое число — *дескриптор* (иденти- фикатор) *этого средства связи*, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор *должен передаваться в качестве пара- метра* всем системным вызовам, осуществляющим дальнейшие операции над со- ответствующими средствами System V IPC.

## Утилиты управления средствами пакета IPC

У пользователя ОС, для работы со средствами межпроцессного взаимодей- ствия *System V IPC*, имеются три основные утилиты, которыми следует пользо- ваться по мере необходимости:

* *ipcmk* — создание различных ресурсов средств IPC;
* *ipcs* - вывод отчета о состоянии средств межпроцессного взаимодействия;
* *ipcrm* - удаление очередей сообщений, наборов семафоров и разделяемых сег- ментов памяти.

### Утилита ipcmk

ipcmk [options]

*ipcmk* позволяет открывать разделяемые сегменты памяти, очереди сообще- ний и массивы семафоров.

**Имеются следующие опции**:

-M, --shmem size

Открытие разделяемого сегмента памяти размером *size* байт;

-Q, --queue

Открытие очереди сообщений;

-S, --semaphore number

Открытие массива семафоров с *number* элементами;

-p, --mode mode

Установка доступа к ресурсу; по-умолчанию = *0644*;

-V, --version

Отображение версии пакета;

-h, --help

### Утилита ipcs

ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]

*ipcs* выводит информацию о системных средствах межпроцессного взаимо- действия *System V IPC*.

**Имеются следующие опции** (используйте *ipcs -h*, для конкретной системы):

-a Показать максимально возможное количество информации во время вывода данных об активных семафорах, очередях сообщений и разделяемых сегмен- тах памяти. Это эквивалентно указанию опций -b, -c, -o, -p и -t.

-b Показать максимально допустимые размеры активных семафоров, очередей сообщений и разделяемых сегментов памяти. «Максимальный допустимый размер» -- это максимальное количество байт в сообщении в очереди сообщений, размер разделяемого сегмента памяти или количество семафо-

ров в наборе семафоров.

-c Показать имя и группу создателя активных семафоров, очередей сообщений и разделяемых сегментов памяти.

-m Вывести информацию об активных сегментах разделяемой памяти.

-o Показать пиковое использование активных очередей сообщений и раз- деляемых сегментов памяти. «Пиковое использование» -- это количество сообщений в очереди сообщений, или количество процес- сов, подключенных к разделяемому сегменту памяти.

-p Показать информацию об идентификаторе процесса для активных сема-

форов, очередей сообщений и разделяемых сегментов памяти.

«Идентификатором процесса» является последний процесс, отпра- вивший или получивший сообщение из очереди сообщений, процесс, создавший семафор, или последний процесс, подключившийся или отключившийся от разделяемого сегмента памяти.

-q Вывести информацию об активных очередях сообщений.

-s Вывести информацию об активных семафорах.

-t Показать время доступа к активным семафорам, очередям сообщений и разделяемым сегментам памяти. Время доступа -- это время послед- ней операции управления IPC объектом, последняя отправка или приём сообщения, последнее подключение или отключение от разделя- емого сегмента памяти, или последняя операция с семафором.

-C дамп

Извлечь значения из списка имён (namelist) указанного дампа памяти ядра, вместо определённого по умолчанию /dev/kmem. Подразумевает -y.

-M Вывести системную информацию о разделяемой памяти.

-N система

Извлечь список имён из указанной системы, вместо определённой по умолчанию /boot/kernel/kernel. Подразумевает -y.

-Q Вывести системную информацию об очередях сообщений.

-S Вывести системную информацию о семафорах.

-T Вывести системную информацию о разделяемой памяти, очередях сооб- щений и семафорах.

-y Использовать интерфейс kvm вместо интерфейса sysctl для извлечения необходимой информации. Если ipcs запущена на работающей системе, использование kvm(3) потребует привилегии чтения из /dev/kmem.

-u пользователь

Вывести информацию о механизмах IPC для указанного пользователя. Пользователь может быть задан либо числовым идентификатором UID, либо регистрационным именем.

**Замечание** Если не указана ни одна из опций -M, -m, -Q, -q, -S, или -s, то выводится информация обо всех активных средствах IPC.

### Утлита ipcrm

ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey] [-S semkey] ...

*ipcrm* удаляет указанные очереди сообщений, семафоры и разделяемые сег- менты памяти из системы. Требуемые объекты System V IPC задаются идентифика- тором их создания или любым связанным с ними ключом.

Для выбора объектов IPC, которые будут удалены, используются следующие опции, которых может быть задано любое число и любая комбинация:

-q msqid

Удалить из системы очередь сообщений, связанную с идентификатором msqid.

-m shmid

Пометить для удаления разделяемый сегмент памяти, связанный с идентификатором shmid. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.

-s semid

Удалить из системы набор семафоров, связанный с идентификатором semid.

-Q msgkey

Удалить из системы очередь сообщений, связанную с ключом msgkey.

-M shmkey

Пометить для удаления разделяемый сегмент памяти, связанный с ключом shmkey. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.

-S semkey

Удалить из системы набор семафоров, связанный с ключом semkey.

**Замечание** Идентификаторы и ключи, связанные с этими объектами System V IPC, могут быть най- дены с помощью утилиты ***ipcs***.

## Семафоры

Для решения проблем *взаимого исключения* и *взаимной блокировки* были предприняты значительные усилия. Естественно, что указанные проблемы косну- лись и стандарта POSIX, в частности, стандарта *POSIX-2001*. Со временем, вопро- сы согласования действий между взаимодействующими процессами стали называть задачами *синхронизации*.

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. При разработке средств System V IPC семафоры вошли в их состав как неотъ- емлемая часть.

**Замечание**

Следует отметить, что набор операций над семафорами System V IPC отличается от

классического набора операций, предложенного Дейкстрой.

**Набор действий** над семафорами System V IPC включает три операции:

1. *A(S, n) — семафор S увеличивается на n;*
2. *D(S, n) – пока значение семафора S < n, процесс блокируется. Иначе процес не блокируется и выполняется операция S = S - n;*
3. *Z(S) – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.*

**Основная идея** реализации этого механизма, предполагает, что:

* + семафор - это *минимальный примитив синхронизации*, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе;
  + у семафора есть *значение*, которое представляется целым числом в диапа- зоне *от 0 до 32767*;
  + прикладная реализация механизма синхронизации обеспечивается *набором (массивом) семафоров*, операции над этими наборами, для приложений яв- ляются *атомарными*;
  + *гарантом атомарности* операций на наборами является *ядро ОС*, в котором и реализованы механизмы синхронизации.

Непосредственно в пакете IPC, работа с семафорами осуществляется с помощью трех системных вызовов:

//Необходимо для совместимости со старыми версиями

//#include <sys/types.h>

//#include <sys/ipc.h>

#include <sys/sem.h>

int *semget* (key\_t key,

int

nsems, int

semflg);

int *semop* (int semid, struct sembuf \*sops, size\_t nsops); int *semctl* (int semid, int semnum, int cmd, ...);

Рассмотрим отдельно каждый из этих вызовов.

Системный вызов *semget(...)* предназначен для выполнения операции доступа к массиву IPC-семафоров или создание такого массива.

В случае успешного завершения функции, возвращается дескриптор System V IPC для этого массива, который является целым неотрицательным числом, одно- значно характеризующим массив семафоров внутри вычислительной системы и ис- пользуется в дальнейших операциях над ним:

int *semget* (key\_t key, int nsems, int semflg);

где *key* — ключ, ассоциированный с семафором и генерируемый функцией *ftok(...)*; *nsems* - задает число семафоров в наборе;

*semflg* — флаг семафора; обычно ассоциируется с константами:

* *IPC\_CREAT* — если массив для указанного ключа не существует, то он дол- жен быть создан;
* *IPC\_EXCL* — применяется совместно с флагом *IPC\_CREAT*; при совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка; при этом, переменная

*errno*, описанная в файле *<errno.h>*, примет значение *EEXIST*.

**Дополнительные права доступа** могут иметь значения:

* 0400 — разрешено чтение для пользователя, создавшего массив;
* 0200 — разрешена запись для пользователя, создавшего массив;
* 0040 — разрешено чтение для группы пользователя, создавшего массив;
* 0020 — разрешена запись для группы пользователя, создавшего массив;
* 0004 — разрешено чтение для всех остальных пользователей;
* 0002 — разрешена запись для всех остальных пользователей.

**В случае ошибки**, возвращается *-1*, а переменной *errno* присваивается ее номер:

* *EACCES* - набор семафоров существует для ключа *key*, но вызывающий про- цесс не имеет прав на доступ к набору;
* *EEXIST* - набор семафоров существует для ключа *key*, а в *semflg* не включены флаги *IPC\_CREAT* и *IPC\_EXCL*;
* *ENOENT* - набора семафоров для ключа *key* не существует, а в *semflg* не включен флаг *IPC\_CREAT*;
* *EINVAL* - значение *nsems* меньше *0* или больше максимально возможного для набора количества семафоров (*SEMMSL*), или набор семафоров, соответст- вующий *key* уже существует и *nsems* больше, чем количество семафоров в

этом наборе;

* *ENOMEM* - набор семафоров должен быть создан, но недостаточно памяти для создания новой структуры;
* *ENOSPC* - набор семафоров должен быть создан, но при этом будет превы- шен системный лимит количества наборов семафоров (*SEMMNI*) или систем- ный лимит количества семафоров (*SEMMNS*).

**Замечание**

Вновь созданные семафоры *инициируются нулевым значением*.

Системный вызов *semop(...)* предназначен для выполнения операций ***A***, ***D*** и ***Z***

на основе ключа, который получен системным вызовом *semget(...)*:

int *semop* (int semid, struct sembuf \*sops, size\_t nsops);

где *semid* — идентификатор набора семафоров, созданный функцией *semget(...)*; *sops* — указатель на массив структур, с числом элементов *nsops*, состоящим из структур типа *sembuf*;

*nsops* - число семафоров, указанное при вызове *semget(...)*.

При успешном завершении возвращает *0*, иначе — возвращает *-1*, а перемен- ной *errno* присваивается номер ошибки:

Действия, которые выполняются над набором семафоров, определяются вто- рым аргументом системного вызова *semop(...)*, являющимся указателем на массив сируктур типа *sembuf*.

**Каждая структура** типа *sembuf* содержит, по крайней мере, следующие три

поля:

unsigned short sem\_num; // Номер семафора в наборе (нумерация с нуля)

short

sem\_op; // Запрашиваемая операция над семафором

short

sem\_flg; // Флаги операции: 0 — операции с блокировкой

**Операция** над конкретным семафором определяется значением поля *sem\_op*:

* *положительное значение* предписывает увеличить значение семафора на ука- занную величину, соответствует операции ***A(S,n)***;
* *отрицательное* - уменьшить, соответствует операции ***D(S,n)***;
* *нулевое* - сравнить с нулем, соответствует операции ***Z(S)***.

**Замечание** Вторая операция не может быть успешно выполнена, если в результате значение сема- фора становится отрицательным, *а третья* - если значение семафора ненулевое. В та- ких случаях, процесс выполняющий операцию — *блокируется*.

С точки зрения пользовательского процесса, выполнение операций над мас- сивом семафоров является неделимым действием. Это значит:

* *если операции выполняются*, то только все вместе;
* *никакой другой процесс не может получить доступ* к промежуточному сос- тоянию набора семафоров, когда часть операций из массива уже выполни- лась, а другая еще не успела.

Сама ОС выполняет операции из массива семафоров - по очереди, причем по- рядок их обработки - не оговаривается.

Если очередная операция не может быть выполнена, то:

* эффект предыдущих операций *аннулируется*;
* вызов функции *semop(...)* приостанавливается (операция с блокировкой, когда

*sem\_flg=0*) или немедленно завершается неудачей, когда выполняется *опера-*

*ция без блокировки*.

**Замечание**

В случае неудачного завершения вызова *semop(...)* значения всех семафоров в наборе останутся неизменными.

Последний изучаемый в данной теме системный вызов *semctl(...)* предназна- чен *для управления набором семафоров*. Он может иметь три или четыре аргумен- та:

int *semctl* (int semid, int semnum, int cmd, [union semun arg]);

где *semid* - идентификатор набора семафоров;

*semnum* - номер семафора в наборе, определяющий объект, над которым выполня- ется управляющее действие, задаваемое значением третьего аргумента *cmd*; если объектом является набор, значение *semnum* — *игнорируется*;

*cmd* - набор команд выполняемых на семафорами; для некоторых команд исполь- зуется четвертый аргумент *arg*, имеющий вид:

union semun { int

val;

// Значение для SETVAL

struct semid\_ds \*buf;

unsigned short

} arg;

\*array;

// Буфер

// Массив

для IPC\_STAT, IPC\_SET

для GETALL,

SETALL

Полный перечень значений аргумента *cmd* следует изучать по руководству:

***man semctl***.

Далее рассмотрим наиболее важные значения *cmd*.

**Одиночные операции над семафором**:

* *GETVAL* - получить значение семафора и выдать его в качестве результата;
* *SETVAL* - установить значение семафора равным **arg.val**.

val = semctl (semid, semnum, GETVAL);

arg.val = ...;

if (semctl (semid, semnum, SETVAL, arg) == -1) ...;

**Групповые операции над набором семафоров**:

* *GETALL* - прочитать значения всех семафоров набора и поместить их в мас- сив **arg.array**;
* *SETALL* - установить значения всех семафоров набора равными значениям элементов массива.

arg.array = (unsigned short \*) malloc (nsems \* sizeof (unsigned short));

err = semctl (semid, 0, GETALL, arg);

for (i = 0; i < nsems; i++) arg.array [i] = ...; err = semctl (semid, 0, SETALL, arg);

**Информационные операции**:

* *GETPID* - узнать PID процесса, выполнившего последнюю операцию над се- мафором;
* *GETNCNT/GETZCNT* — узнать число процессов, ожидающих *увеличения / обнуления* значения семафора.

lpid = semctl (semid, semnum, GETPID);

ncnt = semctl (semid, semnum, GETNCNT); zcnt = semctl (semid, semnum, GETZCNT);

**Управляющие команды над семафорами**:

* *IPC\_STAT* - получить информацию о состоянии набора семафоров ;
* *IPC\_SET* - переустановить характеристики;
* *IPC\_RMID* - удалить набор семафоров.

arg.buf = (struct semid\_ds \*) malloc (sizeof (struct semid\_ds); err = semctl (semid, 0, IPC\_STAT, arg);

arg.buf->sem\_perm.mode = 0644;

err = semctl (semid, 0, IPC\_SET, arg);

err = semctl (semid, 0, IPC\_RMID);

В случае ошибки, *semctl(...)* вернет значение *-1*, а переменная *errno* будет иметь одно из значений:

* *EACCES* - вызывающий процесс не имеет права доступа, необходимого для запуска *cmd*;
* *EFAULT* - адрес, указанный *arg***.buf** или *arg***.array** , недоступен;
* *EIDRM* - набор семафоров был удален;
* *EINVAL* - неверное значение *cmd* или *semid*;
* *EPERM* - аргумент *cmd* имеет значение *IPC\_SET* или *IPC\_RMID*, но вызыва- ющий процесс не имеет достаточных привилегий на выполнение команды;
* *ERANGE* - аргумент *cmd* имеет значение *SETALL* или *SETVAL*, или значение, присваиваемое *semval*, для некоторых семафоров в наборе, меньше нуля или больше, чем стандартное значение *SEMVMX*.

## Задача об обедающих философах

Как уже было отмечено ранее, семафоры:

* *обеспечивают* программиста наддежными *средствами синхронизации* про- цессов;
* *требуют* от программиста разработки и реализации алгоритмов, обеспечива- ющих *взаимное исключение* процессов и устранение их *взаимных блокировок*.

В сложных реальных задачах, в которых взаимодействуют множество асинх- ронно выполняющихся процессов, разработка надежных алгоритмов синхрони- зации также превращается в самостоятельную проблему.

В таких случаях, программисту требуется правильно провести декомпозицию алгоритма работы приложений и реализовать его по частям.

В данном подразделе, на классическом примере «*Задачи об обедающих фило- софах*», мы рассмотрим основные этапы подготовки и реализации проекта по син- хронизации процессов, которые могут приводить к *взаимным блокировкам*.

### Описание задачи

**Класический вариант** «*Задачи об обедающих философах*» демонстрируется рисунком 1.2 и следующим описанием:

* За круглым столом сидит несколько философов.
* В каждый момент времени каждый из них *либо беседует*, *либо ест*.
* Для процесса еды одновременно *требуются две вилки*. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу *необходимо до-*

*ждаться, пока освободятся обе вилки* - слева и справа от него, и взять их в руки.

* Немного поев, философ *кладет вилки на стол* и вновь присоединяется к бе- седе.
* Требуется разработать программную модель обеда философов.

**Главная проблема в этой задаче,** - *корректная дисциплина захвата и осво- бождения вилок*, иначе например, если каждый из философов, одновременно с дру- гими, возьмется за вилку, лежащую слева от него, и будет ждать освобождения пра- вой, то - *обед не завершится никогда*.

Прежде, чем приступить к реализации алгоритма синхронизаци *процессов- философов*, определимся с общей стратегией решения задачи.

### Выбор стратегии решения

Одним из общих подходов решения задач синхронизации множества асинх- ронно взаимодействующих процессов является разделение структуры всего алго- ритма на *два уровня иерархии*:

* первый — *нижний уровень* реализует сами действующие процессы, выделяя те общие ресурсы, которые нужны ему для обеспечия функционирования;
* второй — *верхний уровень* (программа-монитор) обеспечивает нижний уро- вень необходимыми данными, проводит первоначальную подготовку разделя- емых ресурсов и, по возможности, устраняет возникающие проблемы синх-

ронизации процессов нижнего уровня.

Для нашего случая:

* *нижний уровень* — программы *процессов-философов*, которые борятся за ре- сурс в виде двух вилок;
* *верхний уровень* — *программа-монитор*, которая подготавливает набор сема- форов для синхронизации всех процессов-философов, создает и запускает эти процессы и отслеживает их завершение.

Далее, переходим к построению и реализации моделей каждого уровня.

### Модель философа

Каждый философ ведет себя независимо от других: он ест и беседует, причем время, необходимое для беседы и поедания порции (части) своего обеда, не привя- зано к аналогичным действиям других философов.

Для моделирования такой независимости *предполагаем*, что:

* время *одной порции разговора* философа - случайное значение ***trnd***;
* время *одной порции поедания* обеда философа - случайное значение ***ernd***;
* общее *время поедания всего обеда* — ограничено, поэтому философ может завершить обед в несколько приемов.

**Описание действий философа**:

* философ какое-то время беседует (***trnd***), затем пытается взять вилки слева и справа от себя;
* когда ему это удается, он некоторое время ест (**ernd**), после чего освобождает вилки;
* так продолжается до тех пор, пока не будет съеден весь обед, после чего про- грамма, моделирующая действия философа, закончит свою работу.

**Таким образом**, мы видим, что для нормальной работы философа необходи- мы два ресурса, которые обеспечиваются двумя семафорами.

Чтобы реализовать корректную модель *программы-философа*, необходимо уточнить исходные данные, передаваемые ей в качестве аргументов.

Учитывая, что отдельный философ использует только два разделяемых ре- сурса из общего их колличества, равного общему числу философов, то для запуска программы достаточно только трех аргументов:

* *argv[1]* — идентификатор набора семафоров;
* *argv[2]* — общее число философов;
* *argv[3]* — порядковый номер конкретного философа за столом.

**Замечание**

Философы нумеруются, начиная с 1.

Вилки нумеруются, начиная 0. Номер вилки — номер семафора.

Значение семафора = 1, когда вилка — свободна. Значение семафора = 0, когда вилка — занята.

### Программа-монитор

Для рассматриваемой задачи, учитывая, что количество обедающих филосо- фов задано некоторой константой ***QPH***, алгоритм работы достаточно прост:

* *порождается* набор семафоров ***QPH***: по одному семафору на каждую вилку;
* *устанавливаются* начальные значения семафоров: занятой вилке будет соот- ветствовать значение 0, свободной — 1;
* запускаются ***QPH*** процессов, каждый из которых представляет одного фило- софа, передавая им в качестве аргументов: идентификатор набора семафоров,

общее количество всех философов и место конкретного философа за столом: философы нумеруются от 1 до ***QPH***;

* *ожидается* завершение работы всех процессов: когда все философы съедят свой обед;
* *удаляется* весь набор семафоров.