



# Средства межпроцессного взаимодействия

# Цели и задачи

- Обеспечить средства взаимодействия между процессами
- Исключить нежелательное влияние одного процесса на другой



# Задачи, решаемые взаимодействующими процессами

- Передача данных
- Совместное использование  
данных
- Извещения

# Используемые средства

- Сигналы
- Каналы (именованные и неименованные)
- Сообщения
- Семафоры
- Разделяемая память
- Сокеты



# Средства IPC System V

## Inter-Process Communication

- Сообщения
- Семафоры
- Разделяемая память

# Пространство имен

Используется ключ — некоторый числовой идентификатор, позволяющий, с одной стороны, двум процессам обратиться к одному и тому же ресурсу, а с другой — двум другим процессам работать с другим ресурсом.



# СИСТЕМНЫЙ ВЫЗОВ

```
key_t ftok(char * filename, char  
(int) project);
```

Файл не может быть  
временным, так как для  
генерации ключа используется  
номер i-node.

# Специальный ключ IPC\_PRIVATE

При его использовании всегда создается новый ресурс. Ответственность за его удаление несет процесс-создатель.



# Идентификатор

Имеет стандартный тип `int`.

Используется при всех обращениях процесса к ресурсу.

Пространства идентификаторов отдельные для всех трех типов объектов ИРС.

Идентификатор имеет смысл не только в контексте процесса.

# Ключ и идентификатор (1)

Ключ генерирует процесс. Ключ уже может принадлежать существующему ресурсу, а может быть создан новый ресурс. Любому ресурсу присваивается идентификатор аналогично назначению PID новому процессу.



# Ключ и идентификатор (2)

Генерация ключа с использованием стандартной функции гарантирует, что «неродственные» процессы не получают одинаковый ключ. С другой стороны, позволяет «родственным» процессам иметь несколько разных ключей.

# Структура `irc_perm`

`uid` – идентификатор владельца

`gid` – группа владельца

`cuid` – идентификатор создателя

`cgid` – группа создателя

`mode` – права доступа

`key` – ключ



# Права доступа

Используются только права на чтение и запись. Максимальный набор прав 0666 в восьмеричном виде. Маска при создании объекта не применяется.

# Префиксы

msg – очереди сообщений

sem – семафоры

shm – разделяемая память



# СИСТЕМНЫЕ ВЫЗОВЫ

xxx – префикс

xxxget – получение  
идентификатора ресурса по  
ключу (возможно создание)

xxxor – выполнение стандартных  
операций с ресурсом

xxxctl – выполнение операций по  
управлению ресурсом

# Очереди сообщений

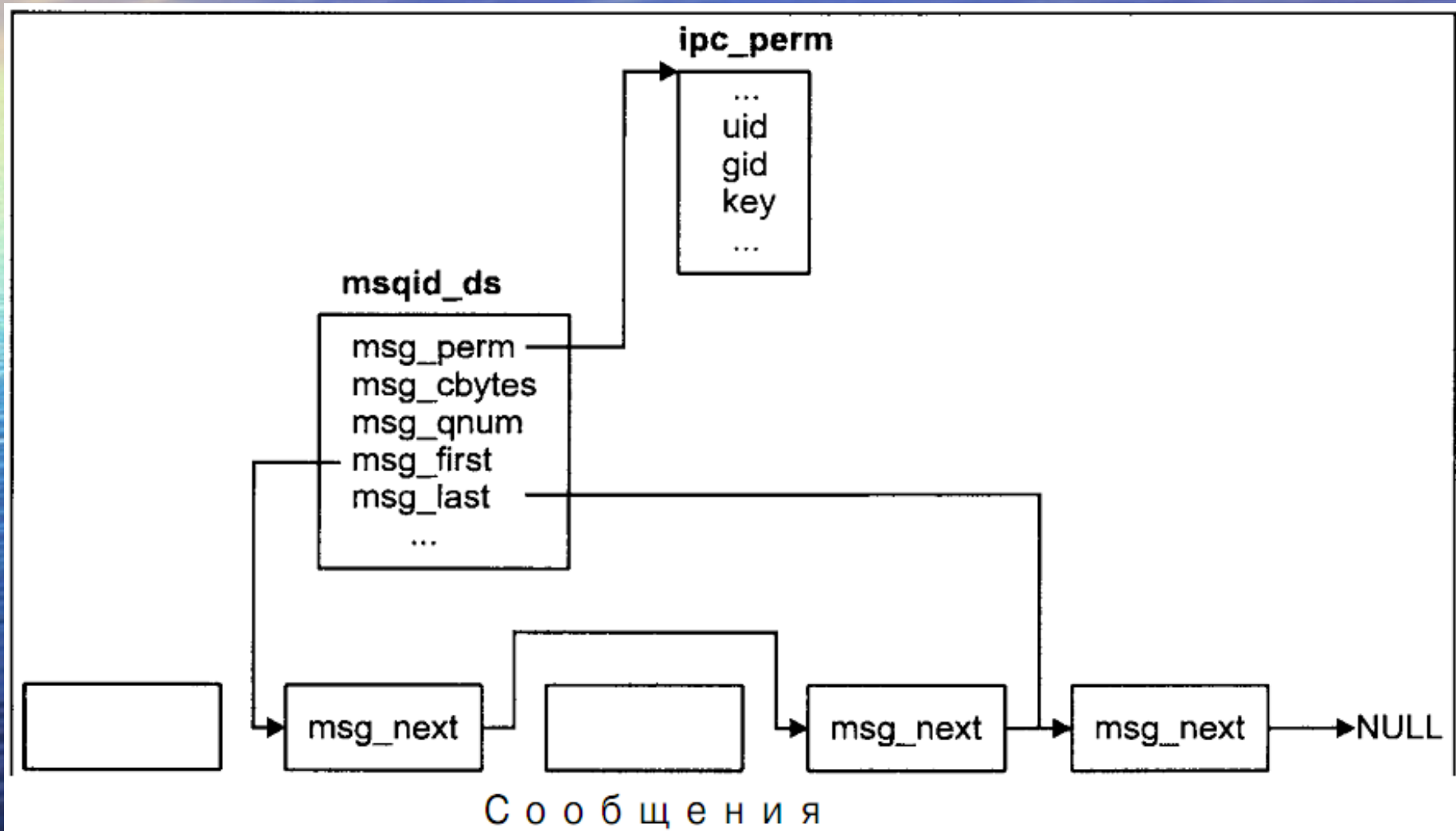
Предоставляют возможность  
процессам обмениваться  
структурированными данными  
– сообщениями.



# Сообщение

- Тип сообщения (положительное число)
- Текст сообщения (может быть нулевое сообщение)

# Структура очереди сообщений





# Структура msgid\_ds (1)

msg\_perm – права доступа и ключ

msg\_rtime – время последнего  
извлечения сообщения из  
очереди

msg\_stime – время последней  
отправки сообщения в очередь

msg\_ctime – время последнего  
изменения атрибутов очереди

# Структура msgid\_ds (2)

`msg_qnum` – текущее количество сообщений в очереди

`msg_qbytes` – максимальный размер всех сообщений в очереди



# Структура msgid\_ds (3)

msg\_lspid – идентификатор  
процесса, отправившего  
последнее сообщение в очередь

msg\_lrpid – идентификатор  
процесса, последним  
извлекавшего сообщение

# Получение идентификатора

```
int msgget(key_t key, int msgflg);
```

msgflg – perm | flags

IPC\_CREAT

IPC\_EXCL



# Отправка сообщения в очередь

```
int msgsnd(int msqid,  
    const void * msgp,  
    size_t msgsz,  
    int msgflg);
```

IPC\_NOWAIT

# Извлечение сообщения из очереди (1)

```
size_t msgrcv(int msqid,  
    void *msgp,  
    size_t msgsz,  
    long msgtyp,  
    int msgflg);
```



# Извлечение сообщения из очереди (2)

`msgtyp`

- $>0$  – первое сообщение указанного типа
- $=0$  – первое сообщение в очереди
- $<0$  – сообщение наименьшего типа не больше, чем  $|\text{msgtyp}|$

# Извлечение сообщения из очереди (3)

Флаги

IPC\_NOWAIT

MSG\_EXCEPT

MSG\_NOERROR



# Управление очередью сообщений

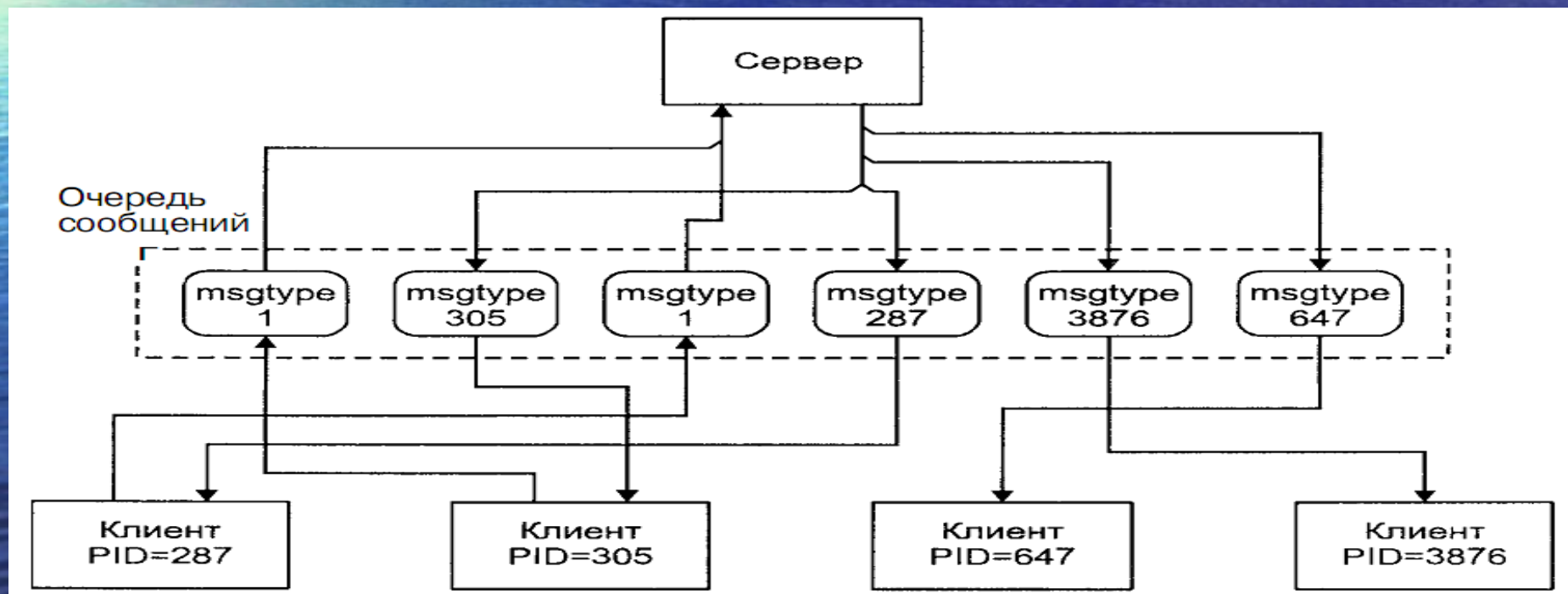
```
int msgctl(int msgid, int cmd,  
           struct msgid_ds * buf);
```

IPC\_STAT

IPC\_SET

IPC\_RMID

# Мультиплексирование сообщений в одной очереди





# Мультиплексирование сообщений

Сервер и несколько клиентов. Сервер адресует сообщение каждому клиенту, используя тип сообщения. (Например PID процесса). Клиенты посылают сообщения серверу с типом 1. Клиент идентифицирует себя в теле сообщения.

# Семафоры

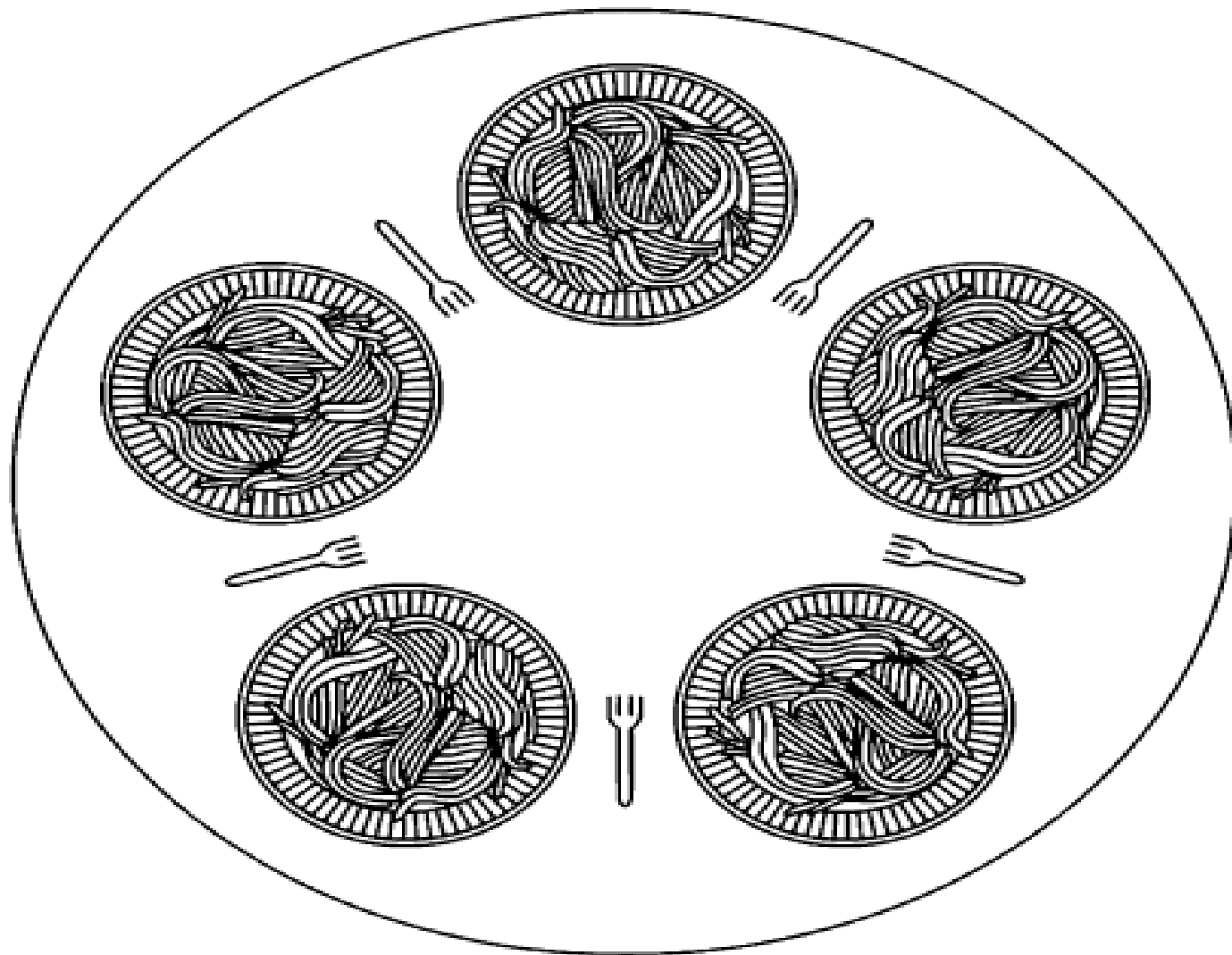
Предоставляют возможность синхронизации процессов при доступе к разделяемому ресурсу.



# Лирическое введение

- Обедающие философы
- Читатели и писатели
- Спящий брадобрей

# Обедающие философы





# Ошибочное решение (1)

```
#define N 5                                /* количество философов */

void philosopher(int i)                    /* i: номер философа (от 0 до 4) */
{
    while (TRUE) {
        think( );                          /* философ размышляет */
        take_fork(i);                      /* берет левую вилку */
        take_fork((i+1) % N);              /* берет правую вилку; */
                                           /* % - оператор деления по модулю */
        eat();                             /* есть спагетти */
        put_fork(i);                       /* кладет на стол левую вилку */
        put_fork((i+1) % N);               /* кладет на стол правую вилку */
    }
}
```

# Ошибочное решение (2)

Если все пять философов возьмут одновременно левую вилку, то возникнет взаимная блокировка, так как ни один философ не сможет взять правую вилку и начать есть.



# Верное решение (1)

```
#define N          5          /* количество философов */
#define LEFT      (i+N-1)%N   /* номер левого соседа для i-того философа */
#define RIGHT     (i+1)%N     /* номер правого соседа для i-того философа */
#define THINKING  0          /* философ размышляет */
#define HUNGRY    1          /* философ пытается взять вилки */
#define EATING    2          /* философ ест спагетти */

typedef int semaphore; /* Семафоры – особый вид целочисленных переменных */
int state[N];          /* массив для отслеживания состояния каждого философа */
semaphore mutex = 1;   /* Взаимное исключение входа в критическую область */
semaphore s[N];        /* по одному семафору на каждого философа */

void philosopher(int i) /* i - номер философа (от 0 до N-1) */
{
    while (TRUE) {       /* бесконечный цикл */
        think();         /* философ размышляет */
        take_forks(i);   /* берет две вилки или блокируется */
        eat();           /* ест спагетти */
        put_forks(i);    /* кладет обе вилки на стол */
    }
}
```

# Верное решение (2)

```
void take_forks(int i)      /* i - номер философа (от 0 до N-1) */
{
    down(&mutex);           /* вход в критическую область */
    state[i] = HUNGRY;      /* запись факта стремления философа поесть */
    test(i);                /* попытка взять две вилки */
    up(&mutex);             /* выход из критической области */
    down(&s[i]);            /* блокирование, если вилки взять не удалось */
}

void put_forks(i)          /* i - номер философа (от 0 до N-1) */
{
    down(&mutex);           /* вход в критическую область */
    state[i] = THINKING;   /* философ наелся */
    test(LEFT);            /* проверка, готовности к еде соседа слева */
    test(RIGHT);           /* проверка, готовности к еде соседа справа */
    up(&mutex);            /* выход из критической области */
}

void test(i)               /* i - номер философа (от 0 до N-1) */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```



# Читатели и писатели

Общий доступ в базу данных.

Разрешено одновременное чтение из базы. Но если хотя бы один процесс модифицирует базу данных, то любые другие операции запрещены.

# Решение проблемы? (1)

```
typedef int semaphore;          /* напрягите свое воображение */
semaphore mutex = 1;           /* управляет доступом к 'rc' */
semaphore db = 1;              /* управляет доступом к базе данных */
int rc = 0;                    /* количество читающих или желающих читать процессов */

void reader(void)
{
    while (TRUE) {              /* бесконечный цикл */
        down(&mutex);           /* получение исключительного доступа к 'rc' */
        rc = rc + 1;            /* теперь на одного читателя больше */
        if (rc == 1) down(&db); /* если это первый читатель ... */
        up(&mutex);             /* завершение исключительного доступа к 'rc' */
        read_data_base( );      /* доступ к данным */
        down(&mutex);           /* получение исключительного доступа к 'rc' */
        rc = rc - 1;            /* теперь на одного читателя меньше */
        if (rc == 0) up(&db);    /* если это последний читатель ... */
        up(&mutex);             /* завершение исключительного доступа к 'rc' */
        use_data_read();         /* не критическая область */
    }
}
```



# Решение проблемы? (2)

```
void writer(void)
{
    while (TRUE) {
        think_up_data( );
        down(&db);
        write_data_base( );
        up(&db);
    }
}
```

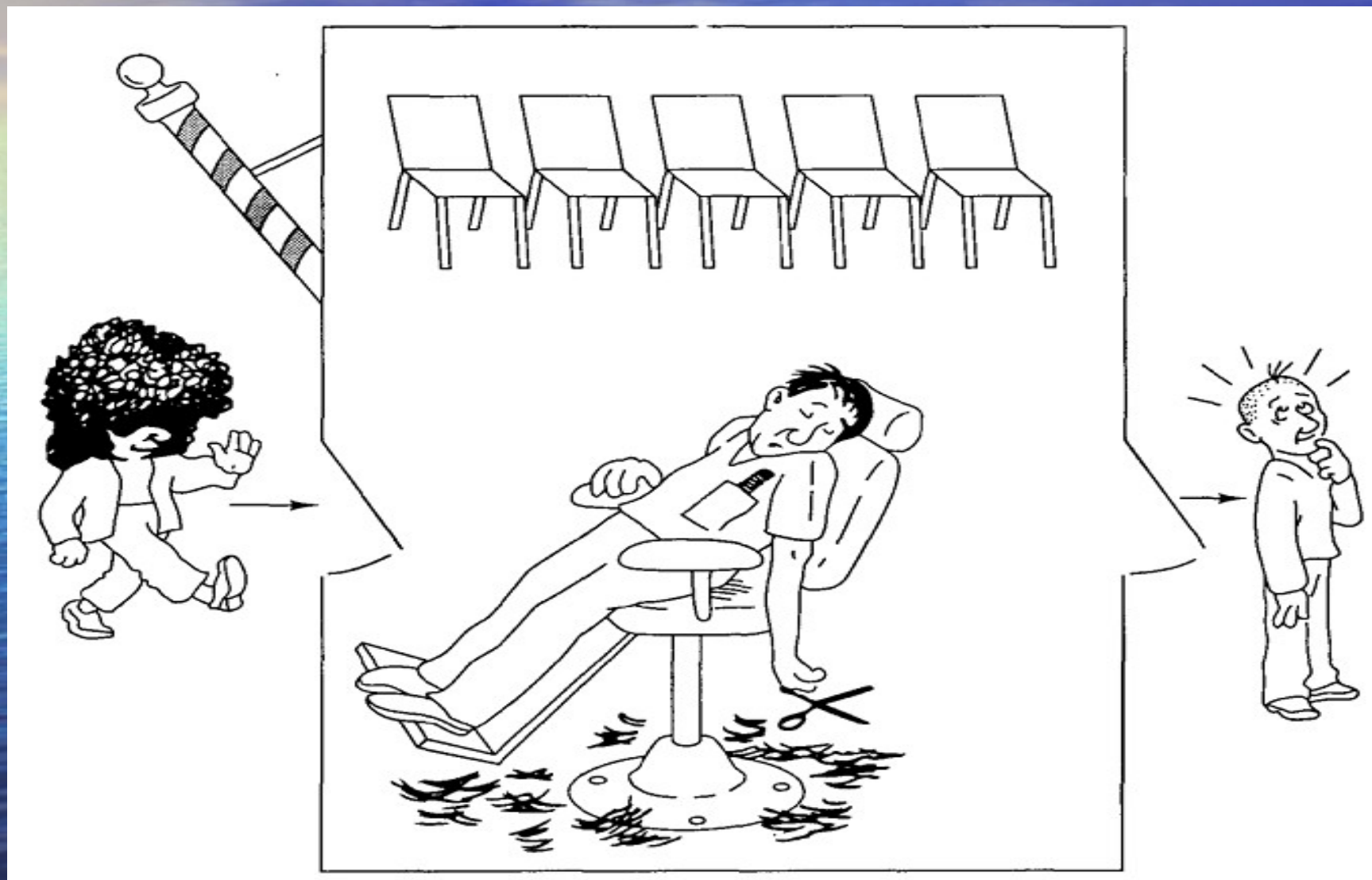
/\* бесконечный цикл \*/  
/\* не критическая область \*/  
/\* получение исключительного доступа \*/  
/\* обновление данных \*/  
/\* завершение исключительного доступа \*/

# Недостаток

Если не установить  
«писателям» более высокий  
приоритет, то при некоторой  
интенсивности работы  
«читателей» «писатели»  
никогда не смогут получить  
доступ к ресурсу.



# Спящий брадобрей



# Решение проблемы (1)

```
#define CHAIRS 5                /* Количество стульев для посетителей */

typedef int semaphore;         /* Догадайтесь сами */

semaphore customers = 0;       /* Количество ожидающих посетителей */
semaphore barbers = 0;        /* Количество бравобреев, ждущих клиентов */
semaphore mutex = 1;          /* Для взаимного исключения */
int waiting = 0;              /* Ожидющие (не обслуживаемые) посетители */

void barber(void)
{
    while (TRUE) {
        down(&customers);      /* Если посетителей нет, уйти в состояние ожидания */
        down(&mutex);          /* Запрос доступа к waiting */
        waiting = waiting - 1; /* Уменьшение числа ожидающих посетителей */
        up(&barbers);          /* Один бравобрей готов к работе */
        up(&mutex);            /* Отказ от доступа к waiting */
        cut_hair();            /* Клиента обслуживают (вне критической области) */
    }
}
```



# Решение проблемы (2)

```
void customer(void)
{
    down(&mutex);           /* Вход в критическую область */
    if (waiting < CHAIRS) {  /* Если свободных стульев нет, придется уйти */
        waiting = waiting + 1; /* Увеличение числа ожидающих посетителей */
        up(&customers);      /* При необходимости, разбудить брадобрея */
        up(&mutex);          /* Отказ от доступа к waiting */
        down(&barbers);      /* Если брадобрей занят, уйти в состояние ожидания */
        get_haircut();        /* Клиента усаживают и обслуживают */
    } else {
        up(&mutex);          /* Много посетителей, из парикмахерской придется уйти */
    }
}
```

# Требования к семафорам

- Значение семафора должно быть доступно различным процессам
- Операция проверки и изменения значения семафора должна быть атомарна по отношению к другим процессам



# Семафоры IPC System V

- Семафор представляет собой группу с единой управляющей структурой
- Каждый семафор группы может принимать любое неотрицательное значение (предел определен системой)

# Структура semid\_ds

sem\_perm – права доступа и ключ

sem\_nsems – количество  
семафоров в группе

sem\_otime – время последней  
операции над семафором

sem\_ctime – время последнего  
изменения атрибутов группы  
семафоров



# Структура семафора

`semval` – значение семафора

`sempid` – идентификатор процесса,  
выполнившего последнюю  
операцию над семафором

`semncnt` – количество процессов,  
ожидающих увеличения семафора

`semzcnt` – количество процессов,  
ожидающих обнуления семафора

# Получение идентификатора

```
int semget(key_t key, int nsems,  
int semflg);
```

semflg – perm | flags

IPC\_CREAT

IPC\_EXCL



# Операции над семафором

```
int semop(int semid,  
          struct sembuf * semop,  
          size_t nops);
```



# Структура sembuf

sem\_num – номер семафора в  
группе

sem\_op – операция

sem\_flg – флаги операции

# Операция

`sem_op`

$>0$  – текущее значение семафора  
увеличивается на `sem_op`

$=0$  – ожидание обнуления  
семафора

$<0$  – ожидание и затем  
уменьшение на  $|\text{sem\_op}|$



# Флаги операции

sem\_flg

IPC\_NOWAIT

SEM\_UNDO



# 1-й случай

Ресурс свободен – 0

Ресурс занят – 1

```
struct sembuf sop_lock[2]={
```

```
    0, 0, 0,
```

```
    0, 1, 0 };
```

```
struct sembuf sop_unlock[1]={
```

```
    0, -1, 0};
```

## 2-й случай

Ресурс свободен – 1

Ресурс занят – 0

```
struct sembuf sop_lock[1]={  
    0, -1, 0 };
```

```
struct sembuf sop_unlock[1]={  
    0, 1, 0};
```



# Управление семафорами

```
int semctl(int semid, int semnum,  
            int cmd, ...);
```

Если есть четвертый параметр, то это `union semun`.



# union semun

int val – значение семафора

struct semid\_ds \* buf –

управляющая структура семафора

struct short \* array – массив

значений семафоров

# Флаги

IPC\_STAT IPC\_SET IPC\_RMID

GETALL GETNCNT GETPID

GETVAL GETZCNT

SETALL SETVAL

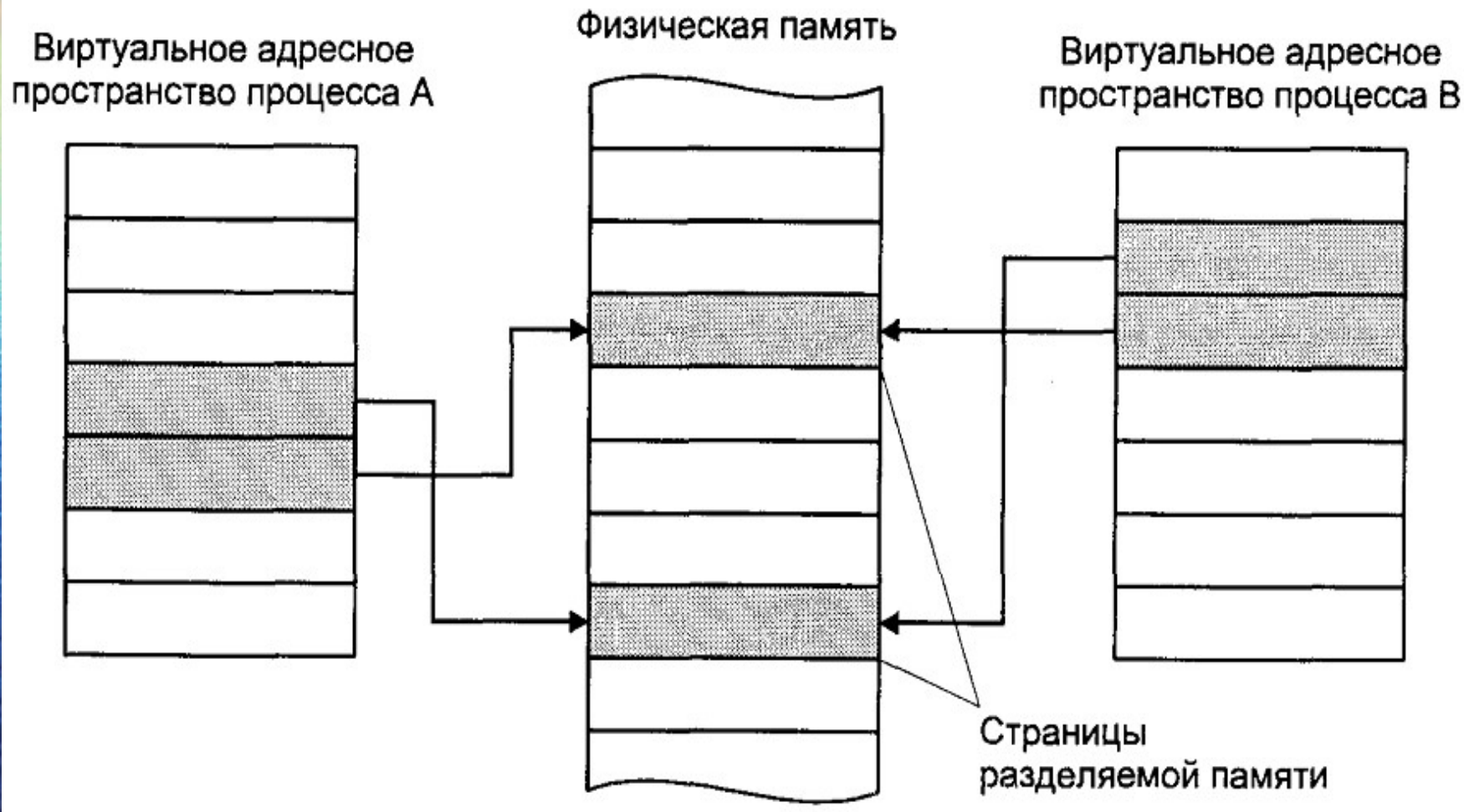


# Разделяемая память

Обеспечивает доступ нескольким процессам к одним и тем же страницам физической памяти. Т.е. одни и те же страницы отображаются в виртуальные адресные пространства нескольких процессов.



# Совместное использование разделяемой памяти



Разделяемая память не содержит встроенных средств синхронизации доступа. Обычно используется совместно с семафорами. Разделяемая память является самым быстрым способом обмена информацией между процессами.



# Структура shmid\_ds (1)

shm\_perm – права доступа и ключ

shm\_segsz – размер выделяемой  
памяти

shm\_atime – время последнего  
присоединения

shm\_dtime – время последнего  
отключения

# Структура shmid\_ds (2)

shm\_ctime – время последнего  
изменения атрибутов  
разделяемой памяти

shm\_nattch – число процессов,  
использующих разделяемую  
память



# Получение идентификатора

```
int shmget(key_t key, size_t size,  
int shmflg);
```

shmflg – perm | flags

IPC\_CREAT

IPC\_EXCL

# Присоединение памяти

```
void *shmat(int shmid,  
            const void *shmaddr, int shmflg);
```

SHM\_RND

SHM\_RDONLY



# Отключение памяти

```
int shmdt(const void *shmaddr);
```

# Управление памятью

```
int shmctl(int shmid, int cmd,  
            struct shmid_ds *buf);
```

IPC\_STAT

IPC\_SET

IPC\_RMID



# fork() exec()

fork() – все сегменты наследуются,  
счетчик ссылок увеличивается

exec() – происходит отключение  
всех сегментов разделяемой  
памяти

# Использование разделяемой памяти (1)

- Сервер получает доступ к разделяемой памяти, используя семафор
- Сервер производит запись данных в разделяемую память



# Использование разделяемой памяти (2)

- После завершения записи сервер освобождает разделяемую память с помощью семафора
- Клиент получает доступ к разделяемой памяти, запирая ресурс с помощью семафора

# Использование разделяемой памяти (3)

- Клиент производит чтение данных из разделяемой памяти
- После завершения чтения клиент освобождает разделяемую память с помощью семафора



# Отображаемые файлы

Отображение участков файла (всего файла) в виртуальное адресное пространство процесса. Позволяет осуществлять быстрый произвольный доступ к файлу. Применяется системой при отображении динамических разделяемых библиотек.

# Отображение файла (1)

```
void *mmap(void *addr,  
           size_t length, int prot, int flags,  
           int fd, off_t offset);
```

Заккрытие файлового дескриптора  
не приводит к снятию  
отображения.



# Отображение файла (2)

prot

PROT\_NONE

PROT\_READ

PROT\_WRITE

PROT\_EXEC

# Отображение файла (3)

flags

MAP\_SHARED

MAP\_PRIVATE

MAP\_FIXED

MAP\_NORESERVE



# Снятие отображения

```
int munmap(void *addr,  
           size_t length);
```