

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ



# Параллельные вычисления

Проектирование многопроцессных программ

Михаил Моисеев

Санкт-Петербург  
2011

# Проектирование многопроцессных программ

- Модель взаимодействующих процессов
  - с общей памятью
  - без общей памяти
- Стандарты и библиотеки
  - POSIX
  - MPI
  - PVM
  - ...

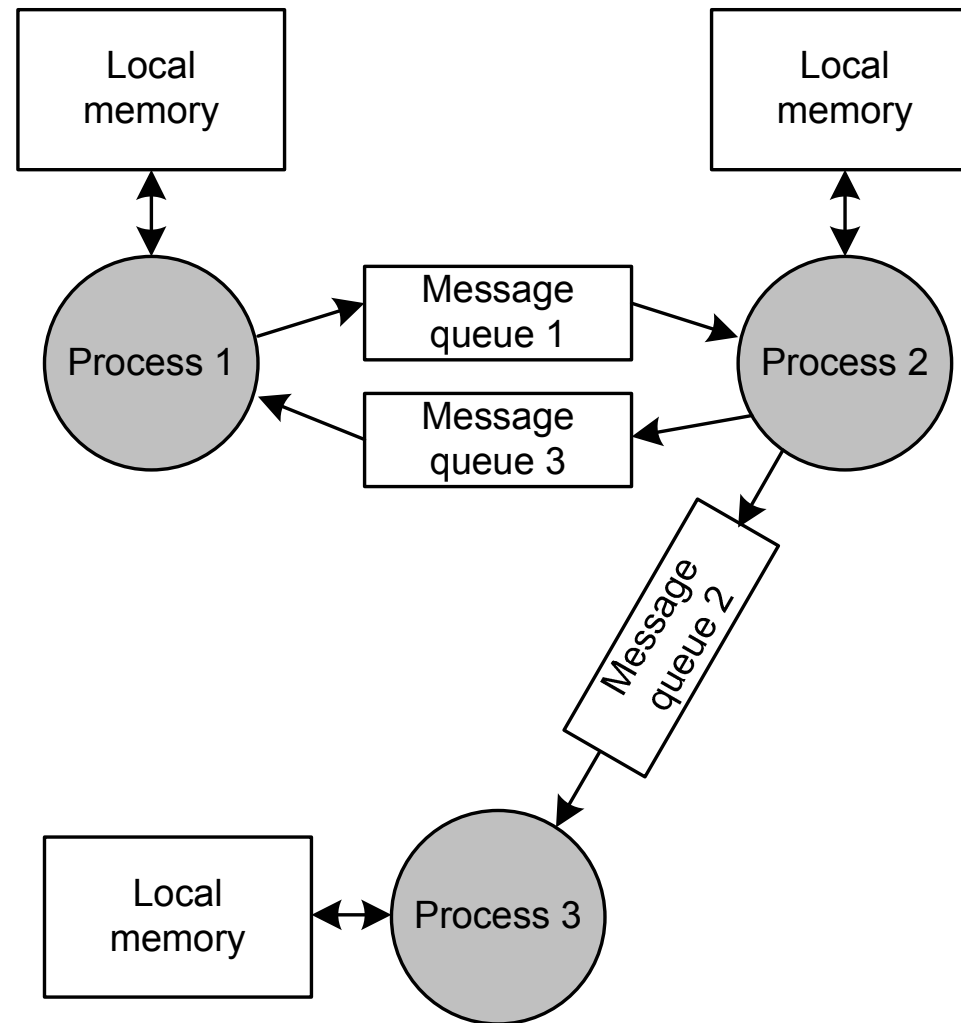
# Создание многопроцессных программ на основе MPI

- MPI (Message Passing Interface) – спецификация интерфейсов (API) для передачи сообщений между процессами
- Разрабатывается MPI Forum <http://www.mpi-forum.org>
  - Последняя версия MPI 2.2 (2009)
- Поддерживаются языки C, C++, Fortran
- Имеются реализации для ОС Linux, Mac OS, Windows

# Модель параллельных вычислений MPI

- В MPI рассматриваются автономные процессы, исполняющие произвольный код и обрабатывающие произвольные данные (MIMD)
- Каждый процесс выполняется в своем адресном пространстве, взаимодействие процессов организуется с помощью сообщений
  - Сообщение от одного процесса другому
  - Сообщение от одного процесса группе процессов
  - Ожидание сообщения одним процессом от группы процессов
- MPI обеспечивает запуск и управление процессами
  - Начальный запуск процессов
  - Динамический запуск процессов
- Поддержка разделяемой памяти процессов

# Модель параллельных вычислений MPI



# Использование MPI

- Инициализация/завершение

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize(void)
```

- Пример программы

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Init( &argc, &argv );
    ...
    MPI_Finalize( );
}
```

# Обмен сообщениями между процессами

## ■ Отправка сообщения с блокировкой процесса

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm)
```

- `Buf` – передаваемый буфер данных
- `Count` – размер передаваемых данных (в числе объектов указанного типа)
- `Datatype` – тип передаваемых данных (Integer, Real, Double, ...)
- `Dest` – идентификатор процесса, которому отправляется сообщение
- `Tag` – тип сообщения (позволяет различать сообщения)
- `Comm` – коммутатор, определяющий множество процессов

## ■ Процесс блокируется до момента когда сообщение получено или скопировано в промежуточный буфер

- `Buf` может быть перезаписан или освобожден

# Режимы передачи сообщений

- `Standard` – при передаче сообщение может копироваться или не копироваться в промежуточный буфер (определяется реализацией MPI)
- `Buffered` – передаваемое сообщение буферизируется, операция `Send` завершается не зависимо от приемной стороны

```
int MPI_Bsend(...)
```

- `Synchronous` – операция `Send` завершается только после начала выполнения `Receive` на приемной стороне

```
int MPI_Ssend(...)
```

- `Ready` – операция `Send` может быть запущена только при наличии соответствующей операции `Receive`, иначе вызов `Send` завершается с ошибкой

```
int MPI_Rsend(...)
```



# Обмен сообщениями между процессами

## ■ Получение сообщения с блокировкой процесса

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Buf – буфер для приема данных
- Count – размер приемного буфера (в числе объектов указанного типа)
- Datatype – тип передаваемых данных (Integer, Real, Double, ...)
- Source – идентификатор процесса (rank) или MPI\_ANY\_SOURCE
- Tag – тип принимаемого сообщения или MPI\_ANY\_TAG
- Status – хранит статус сообщения, а также данные об отправителе

## ■ Возможность получать сообщения от произвольного отправителя

## ■ Определение числа принятых данных

```
int MPI_Get_count(MPI_Status *, MPI_Datatype, int *count)
```

# Семантика передачи сообщений

- Сохранение порядка сообщений (**Order**) – последовательная отправка сообщений A и B гарантирует получение этих сообщений в том же порядке
  - При использовании конкретного получателя на приемной стороне
  - При однопоточной реализации
- Отсутствие гарантии справедливой обработки сообщений
  - При наличии постоянного потока сообщений одно конкретное сообщение на приемной стороне может быть никогда не получено
- Ограничение ресурсов
  - Необходимо наличие достаточных ресурсов для буферизации и обработки сообщений

# Адресация процессов

- Процесс, которому отправляется сообщение, адресуется с помощью аргументов `Dest` и `Comm`
- `Comm` – коммутатор, определяющий текущую группу процессов
- Коммутатор `MPI_COMM_WORLD` позволяет обращаться ко всем процессам, в которых выполнена инициализация MPI
- `Dest` – идентификатор (номер) процесса в данной группе процессов
- Группа процессов упорядочена – каждый процесс имеет уникальный номер в группе

# Группы процессов

- Группа процессов – упорядоченная коллекция процессов
  - Каждый процесс имеет идентификатор (**rank**) – целое число от 0 до  $n-1$
- Процесс может входить в несколько групп
- Группы процессов обеспечивают разделение пространства процессов для ограничения взаимодействия процессов
  - Выбора одного из процессов по его идентификатору
  - Организации коллективных операций
- Предопределенная пустая группа – `MPI_GROUP_EMPTY`

# Управление группами процессов

- Конструкторы групп процессов

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

- Деструкторы групп процессов

```
int MPI_Group_free(MPI_Group *group)
```

# Управление группами процессов

- Определение размера группы

```
int MPI_Group_size(MPI_Group group, int *size)
```

- Определение идентификатора процесса в группе

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, int  
*ranks1, MPI_Group group2, int *ranks2)
```

- Операции над группами процессов

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,  
MPI_Group *newgroup)
```

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,  
MPI_Group *newgroup)
```

# Коммуникаторы

- Операции с группами процессов выполняются с помощью коммуникаторов
- Коммуникаторы для операций обмена
  - Внутри группы процессов
  - Между двумя группами процессов
- Предопределенные коммуникаторы
  - `MPI_COMM_WORLD` – все процессы
  - `MPI_COMM_SELF` – сам процесс

# Управление коммутаторами

- Создание коммутатора связанного с указанной группой процессов

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm  
*newcomm)
```

- Выделение подгруппы в группы процессов, связанной с коммутатором

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm  
*newcomm)
```

- Другие операции

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```



# Пример программы 1

```
int main(int argc, char **argv)
{
    int me, count; int* buf;

    ...

    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    if(me == 0) {
        MPI_Send(buf, count, MPI_INTEGER, 1, SOME_TAG,
                MPI_COMM_WORLD);
    } else
        MPI_Recv(buf, count, MPI_INTEGER, 0, SOME_TAG,
                MPI_COMM_WORLD, &status);
}
```

## Пример программы 2 (Писатель)

```
int main(int argc, char **argv)
{
    int me, count; int* buf;

    ...

    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    int reader = 0;
    if(me == 0) {
        reader = 1;
    }

    MPI_Send(buf, count, MPI_INTEGER, reader, SOME_TAG,
MPI_COMM_WORLD);
}
```

## Пример программы 2 (Читатель)

```
int main(int argc, char **argv)
{
    int me, count; int* buf; MPI_Status status;

    ...

    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    int writer = 0;
    if (me == 0) {
        writer = 1;
    }

    MPI_Recv(buf, count, MPI_INTEGER, writer, SOME_TAG,
    MPI_COMM_WORLD, &status);
}
```

# Пример программы 3

```
int main(int argc, char **argv)
{
    int me, count; int* buf;
    MPI_Group GroupWorld, subgroup;
    MPI_Comm the_comm;
    int ranks[] = {2, 4, 6, 8};
    ...
    MPI_Comm_group(MPI_COMM_WORLD, &GroupWorld);
    MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup);
    MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
    ...
}
```

# Пример программы 4

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
int color = rank % 3;  
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &myComm);  
if (color == 0){  
    ...  
}  
else if (color == 1){  
    ...  
}  
else if (color == 2){  
    ...  
}
```

# Неблокирующие обмены сообщениями

## ■ Отправка сообщения без блокировки

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int  
dest, int tag, MPI_Comm comm, MPI_Request *request)
```

- Request – объект обеспечивающий контроль за операцией отправки
- ISsend, IBsend, IRsend

## ■ Получение сообщения без блокировки

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int  
source, int tag, MPI_Comm comm, MPI_Request *request)
```

- Request – объект обеспечивающий контроль за операцией приема

# Неблокирующие обмены сообщениями

- При выполнении не блокирующих операций необходимо контролировать состояние операции
  - После выполнения `ISend` процесс начинает передачу сообщения
  - После выполнения `IRecv` процесс готов к приему сообщения
- Контроль завершения обмена
  - Завершение отправки сообщения – процесс может освободить буфер
  - Завершение приема сообщения – процесс может читать полученное сообщение

# Неблокирующие обмены сообщениями

- Ожидание завершения не блокирующей операции

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Request – идентификатор операции обмена
- Status – результат операции

- Проверка текущего состояния операции

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

- Request – идентификатор операции обмена
- Flag – индикатор завершения операции (true)
- Status – результат операции



# Неблокирующие обмены сообщениями

- Ожидание завершения одной из операций

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)
```

- Ожидание завершения всех операций

```
int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
```

- Проверка группы операций

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status)
```

- Проверка всех операций

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)
```

# Неблокирующие обмены сообщениями

- Удаление обмена без ожидания завершения операции

```
int MPI_Request_free(MPI_Request *request)
```

- Вызовы `Test` и `Wait` завершают текущую операцию

- Проверка состояния без завершения операции

```
int MPI_Request_get_status(MPI_Request request, int *flag,  
MPI_Status *status)
```

- Проверка наличия сообщения

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,  
MPI_Status *status)
```

- Отмена передачи сообщения, после вызова `Cancel` необходимо завершить обмен с помощью вызова `Free`, `Wait` или `Test`

```
int MPI_Cancel(MPI_Request *request)
```

# Коллективные обмены

- Коллективные обмены включают все группы одной группы или несколько групп процессов
- Барьерная синхронизация, вызвавший процесс блокируется до тех пока все процессы группы не вызовут эту функцию

```
int MPI_Barrier(MPI_Comm comm)
```

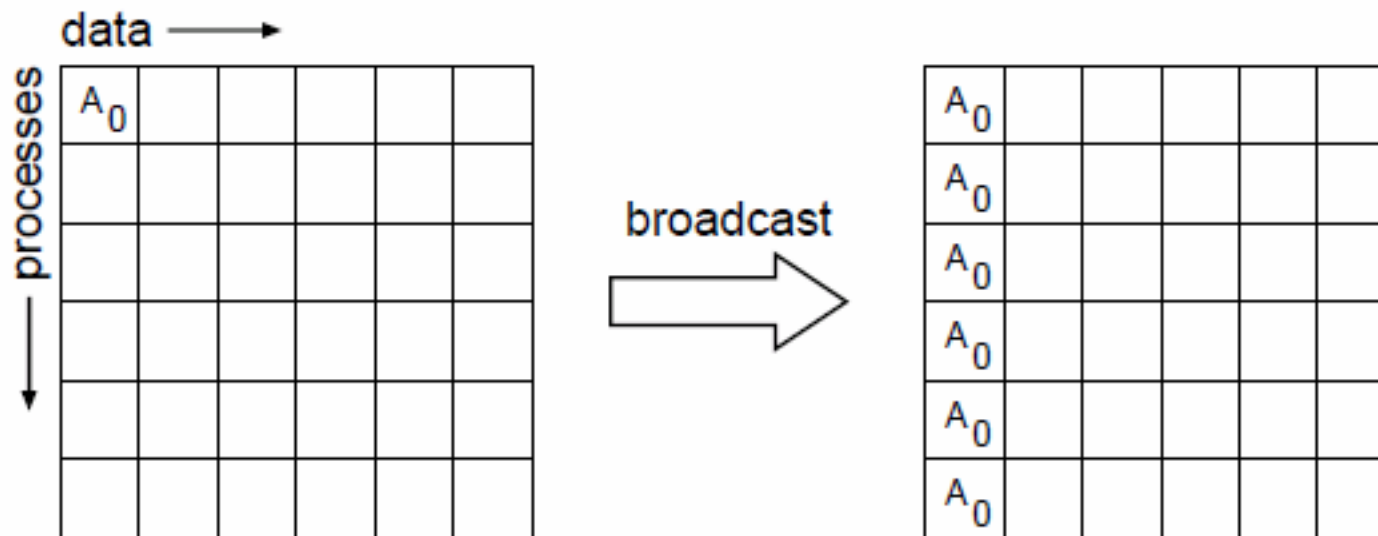
- Comm – коммутатор, определяющий группу процессов

# Коллективные обмены

- Широковещательная рассылка сообщения процессам группы

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,  
int root, MPI_Comm comm)
```

- Root – идентификатор (rank) рассылающего процесса



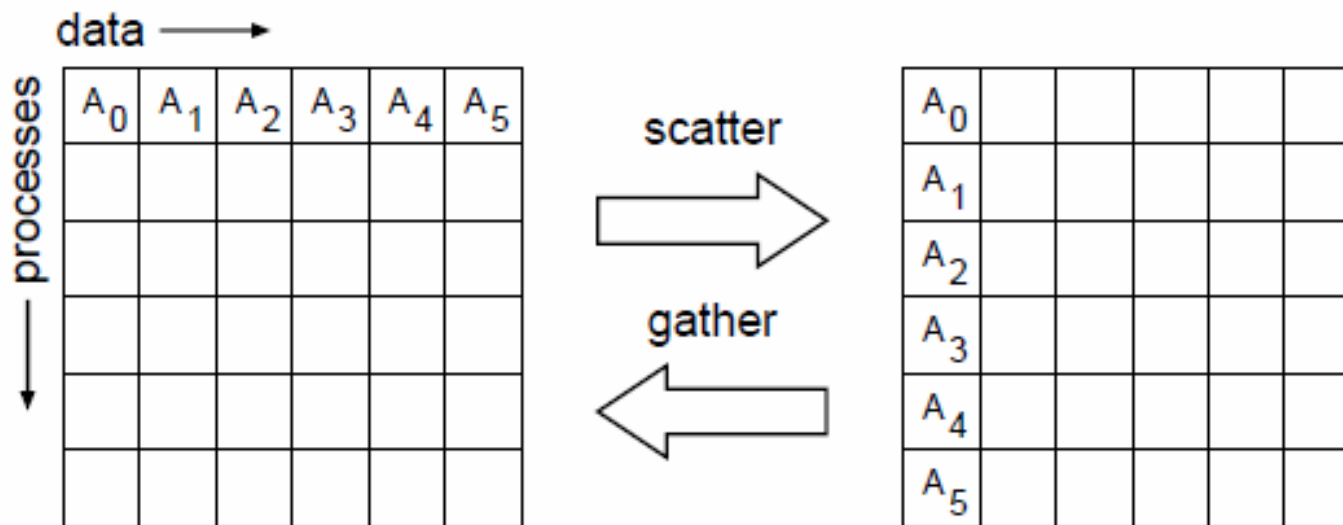
# Коллективные обмены

- Получение сообщения от каждого процесса группы

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype  
sendtype, void* recvbuf, int recvcount, MPI_Datatype  
recvtype, int root, MPI_Comm comm)
```

- Обратная операция

```
int MPI_Scatter(...)
```



# Пример программы

```
int main(int argc, char **argv)
{
    int me, count; int *data;
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if(me == 0){
        data = ...;
        MPI_Bcast(data, count, MPI_INTEGER, 0, MPI_COMM_WORLD);
    }
    MPI_Recv(...);
    ...
}
```

# Удаленный доступ к памяти

- Удаленный доступ к памяти (RMA) – возможность чтения/записи области памяти другого процесса
  - Операции выполняются одним процессом
  - Другой процесс «не знает» о внешнем доступе к его памяти
- Инициализация разделяемой памяти (вызывается для всех процессов группы)

```
int MPI_Win_create(void *base, MPI_Aint size, int  
disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

- Операции доступа к разделяемой памяти

```
int MPI_Put(void *addr, int count, ... MPI_Win win)
```

```
int MPI_Get(void *addr, int count, ... MPI_Win win)
```

# Удаленный доступ к памяти

- Обеспечение синхронизации лежит на пользователе
- Операции блокировки области памяти

```
int MPI_Win_lock(int lock_type, int rank, int assert,  
MPI_Win win)
```

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

- Пример

...

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win)
```

```
MPI_Put(..., rank, ..., win)
```

```
MPI_Win_unlock(rank, win)
```

...



# Управление процессами в MPI

- MPI позволяет динамически запускать и завершать процессы
- Запуск нового процесса

```
int MPI_Comm_spawn(char *command, char *argv[], int  
maxprocs, MPI_Info info, int root, MPI_Comm comm,  
MPI_Comm *intercomm, int array_of_errcodes[])
```

- `Command` – имя запускаемой программы
- `Maxprocs` – число запускаемых процессов
- `Intercomm` – коммутатор для передачи сообщений созданным процессам

# Управление процессами в MPI

- Запуск нескольких программ или программы с разными входными аргументами

```
int MPI_Comm_spawn_multiple(int count, char
    *array_of_commands[], char **array_of_argv[], int
    array_of_maxprocs[], MPI_Info array_of_info[], int root,
    MPI_Comm comm, MPI_Comm *intercomm, int
    array_of_errcodes[])
```

- Получение коммутатора родительского процесса

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

# Создание многопроцессных программ на основе POSIX

- POSIX включает объекты для создания многопроцессных программ (<http://pubs.opengroup.org/onlinepubs/009695399/mindex.html>)
- Способы организации межпроцессного взаимодействия
  - Файлы
  - Сокеты
  - Сигналы
  - Каналы/именованные каналы (pipe/fifo)
  - **Очереди сообщений (message queue)**
  - **Семафоры**
  - **Разделяемая память**

# Синхронизация процессов с помощью семафоров POSIX

- Семафор должен быть доступен из нескольких процессов, операции над семафором должны выполняться атомарно
- Идентификация семафоров с помощью ключей `key_t`
- Создание/получение созданного ключа

```
key_t ftok(const char *path, int id);
```

- `Path` – путь к некоторому файлу, который существует в системе во время работы программы
  - `Id` – ненулевое число
- Значение ключа является уникальным для разных `Path` и `Id`

# Операции над семафорами

- Создание/получение семафора (группы семафоров)

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

- Nsems – число семафоров
- Semflg – права доступа к семафору (IPC\_CREAT, IPC\_EXCL)

- Начальное значение семафора – нулевое

```
#include <sys/ipc.h>
#include <sys/sem.h>
key_t semkey = ftok("/tmp", 1);
if (semkey == (key_t) -1) ...; // Some error

int semid = semget(semkey, 1, IPC_CREAT);
if (semid == -1) ...; // Some error
```

# Операции над семафорами

## ■ Выполнение операций над семафором

```
int semop(int semid, struct sembuf *sops, size_t nsops)
```

- Sops – массив операций, которые нужно выполнить
- Nsops – число операций в Sops

## ■ Структура описывающая операции sembuf

- short sem\_num – номер семафора в группе
- short sem\_op – выполняемая операция
- short sem\_flg – флаги операции

## ■ Возможные операции

- sem\_op > 0 – процесс увеличивает значение семафора на sem\_op
- sem\_op = 0 – процесс ожидает пока семафор не обнулится
- sem\_op < 0 – процесс ожидает пока значение семафора не станет достаточным, затем вычитает модуль sem\_op из значения семафора

# Пример программы

```
int semid;
struct sembuf sb[2];
int nsops = 2;
...
sb[0].sem_num = 0;
sb[0].sem_op = -1;
sb[0].sem_flg = IPC_NOWAIT;
sb[1].sem_num = 1;
sb[1].sem_op = 1;
sb[1].sem_flg = 0;

int result = semop(semid, sb, nsops);
```

# Работа с разделяемой памятью

- Разделяемая память используется для повышения производительности
- Операции с разделяемой памятью должны защищаться с помощью семафоров
- Идентификация областей разделяемой памяти осуществляется с помощью ключей
- Создание/получение доступа к области разделяемой памяти

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

- `key` – идентификатор области памяти
- `size` – размер области в байтах
- `shmflg` – флаги доступа



# Работа с разделяемой памятью

- Присоединение области разделяемой памяти

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

- Shmid – дескриптор полученный в shmget
- Shmaddr – позволяет определить в каком месте памяти процесса будет расположена область разделяемой памяти (0 – на выбор ОС)

- Вызов `shmat` возвращает адрес присоединенной области памяти в виртуальном адресном пространстве процесса

- Удаление присоединенной области разделяемой памяти

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

# Пример программы (писатель)

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
int main() {
    key_t mkey = ftok("/tmp", 1);
    key_t skey = ftok("/tmp", 2);
    int shmid = shmget(mkey, sizeof(int), IPC_CREAT);
    void* shmptr = shmat(shmid, 0, 0);
    int semid = semget(skey, 1, IPC_CREAT);
    *shmptr = 1; // Запись в разделяемую память
    struct sembuf sb[1];
    sb[0].sem_num = 0; sb[0].sem_op = 1; sb[0].sem_flg = 0;
    semop(semid, sb, 1);
}
```

# Пример программы (читатель)

```
int main() {
    key_t mkey = ftok("/tmp", 1);
    key_t skey = ftok("/tmp", 2);
    int shmid = shmget(mkey, sizeof(int), IPC_CREAT);
    void* shmptr = shmat(shmid, 0, 0);
    int semid = semget(skey, 1, IPC_CREAT);
    struct sembuf sb[1];
    sb[0].sem_num = 0; sb[0].sem_op = -1; sb[0].sem_flg = 0;
    // Блокировка процесса до тех пор пока писатель не закончит запись
    int result = semop(semid, sb, 1);
    int i = *shmptr; // Чтение значения из разделяемой памяти
    return i;
}
```

# Пример программы

```
int main() {
    int PERM = S_IRUSR | S_IWUSR;
    key_t skey = ftok("/tmp", 1);
    int semid = semget(skey, 1, PERM | IPC_CREAT | IPC_EXCL);
    if (semid < 0) {
        // Получаем уже созданный семафор
        semid = semget(skey, 1, PERM);
    } else {
        // Если семафор создан нами, то увеличиваем его значение
        struct sembuf sb[1];
        sb[0].sem_num = 0; sb[0].sem_op = 1; sb[0].sem_flg = 0;
        semop(semid, sb, 1);
    }
    ...
}
```

# Передача сообщений между процессами

- Между процессами можно создавать очереди сообщений
- Сообщение включает
  - Тип сообщения
  - Размер данных
  - Данные
- В одной очереди могут передаваться сообщения от разных процессов, сообщения могут приниматься разными процессами
- Идентификация очереди сообщений осуществляется с помощью ключа

# Передача сообщений между процессами

- Создание/получение доступа к очереди сообщений

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

- Отправка сообщения

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

- Msgp – структура типа `msg`
- Msgsz – число байт в поле `text`

```
struct msg {  
    long    mtype;    /* Message type. */  
    char    mtext[]; /* Message text. */  
}
```

# Передача сообщений между процессами

- Прием сообщения

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

- Msgtyp – тип принимаемых сообщений

- Msgtyp = 0 – прием первого сообщения
- Msgtyp > 0 – прием первого сообщения типа Msgtyp
- Msgtyp < 0 – прием первого сообщения с наименьшим типом, который меньше или равен модулю Msgtyp

# Передача сообщений между процессами

- Если `msgflg` содержит `IPC_NOWAIT`, то вызов `Send` или `Receive` будет не блокирующийся

- Информацию о завершении операции можно получить с помощью

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

- `msqid_ds.msg_qnum` – увеличивается на 1
- `msqid_ds.msg_ls_pid` – устанавливается ID процесса



# Пример программы

```
#include <sys/msg.h>

...

int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
msg.type = 1;
strcpy(msg.text, "This is message 1");

...

result = msgsnd(msqid, (void *) &msg, sizeof(msg.text),
IPC_NOWAIT);
```

# Пример программы

```
#include <sys/msg.h>

...

int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
long msgtyp = 1;

...

result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
msgtyp, MSG_NOERROR | IPC_NOWAIT);
```