Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

# Методические указания для выполнения лабораторной работы № 7 по курсу «Операционные системы и системное программирование»

# «Взаимодействие между процессами: семафоры, очереди сообщений»

Полоцк, 2020

**Приостановка и продолжение работы заданий**

Предложим еще один метод, с помощью которого процесс можно перевести в фоновый режим. Процесс запускается обычным образом (на переднем плане), затем приостанавливается командой stop, а потом запускается повторно в фоновом режиме.

Запустим сначала процесс командой yes на переднем плане, как это делалось раньше

```
/home/larry# yes > /dev/null
```

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш Ctrl-C, задание можно приостановить (suspend, буквально – «подвесить»). «Подвешенное» задание не будет уничтожено; его выполнение будет временно остановлено до тех пор, пока оно не будут возобновлено. Для приостановки задания надо нажать соответствующую комбинацию клавиш, обычно это Ctrl-Z

```
/home/larry# yes > /dev/null
{ctrl-Z}
[1]+  Stopped                 yes >/dev/null
/home/larry#
```

Приостановленный процесс попросту не выполняется. На него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно запустить выполняться с той же точки, как будто бы оно и не было приостановлено.

Для возобновления выполнения задания на переднем плане можно использовать команду fg (от слова «foreground» – передний план)

```
/home/larry# fg yes
>/dev/null
```

Командная оболочка еще раз выведет на экран название команды, так что пользователь будет знать, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание еще раз нажатием клавиш Ctrl-Z, но в этот раз запустим его в фоновый режим командой bg (от слова «background» – фон). Это приведет к тому, что данный процесс будет работать так, как если бы при его запуске использовалась команда с символом «&» в конце (как это делалось в предыдущем разделе)

```
/home/larry# bg
[1]+ yes &>/dev/null  &
```

/home/larry#

При этом приглашение командной оболочки возвращается. Сейчас команда jobs должна показывать, что процесс yes действительно в данный момент работает; этот процесс можно уничтожить командой kill, как это делалось раньше.

Для того, чтобы приостановить задание, работающее в фоновом режиме, нельзя пользоваться комбинацией клавиш Ctrl-Z. Прежде чем приостанавливать задание, его нужно перевести на передний план командой fg и лишь потом приостановить. Таким образом, команду fg можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме.

Между заданиями в фоновом режиме и приостановленными заданиями есть большая разница. Приостановленное задание не работает и на него не тратятся вычислительные мощности процессора. Это задание не выполняет никаких действий. Приостановленное задание занимает некоторый объем оперативной памяти компьютера, хотя оно может быть перенесено в «своп». Напротив, задание в фоновом режиме выполняется, использует память и совершает некоторые действия, которые, возможно, требуются пользователю, но он в это время может работать с другими программами.

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами. Например, если ввести команду

/home/larry# yes &

(стандартный вывод не был перенаправлен на устройство /dev/null), то на экран будет выводится бесконечный поток символов y. Этот поток невозможно будет остановить, поскольку комбинация клавиш Ctrl-C не воздействует на задания в фоновом режиме. Чтобы остановить эту выдачу, надо использовать команду fg, а затем уничтожить задание комбинацией клавиш Ctrl-C.

Сделаем еще одно замечание. Обычно командой fg и командой bg воздействуют на те задания, которые были приостановлены последними (эти задания будут помечены символом «+» рядом с номером задания, если ввести команду jobs). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды fg или команды bg их идентификационный номер (job ID). Например, команда

/home/larry# fg %2

помещает задание номер 2 на передний план, а команда

/home/larry# bg %3

помещает задание номер 3 в фоновый режим. Использовать PID в качестве аргументов команд fg и bg нельзя. Более того, для перевода задания на передний план можно просто указать его номер. Так, команда

/home/larry# %2 будет

эквивалентна команде

/home/larry# fg %2

Важно помнить, что функция управления заданием принадлежит оболочке. Команды fg, bg и jobs являются внутренними командами оболочки.

**Механизмы межпроцессного взаимодействия в ОС Unix**

При решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемых ОС, может потребоваться использование следующих системных вызовов

- создание, завершение процесса, получение информации о процессе: *fork*, *exit*, *getpid*, *getppid* и т. д.;
- синхронизация процессов: *signal*, *kill*, *sleep*, *alarm*, *wait*, *paus)*, *semop*, *semctl*, *semcreate* и т. д.;
- создание информационного канала, разделяемой памяти, очереди сообщений и работа с ними: *pipe*, *mkfifo*, *read*, *write*, *msgget*, *shmget*, *msgctl*, *shmctl* и т. д.

Механизм межпроцессного взаимодействия (*Inter-Process Communication Facilities – IPC*) включает

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам – *семафоры* (*semaphores*);
- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу – очереди сообщений (*message queries*);
- средства, обеспечивающие возможность наличия общей для процессов памяти – сегменты разделяемой памяти (*shared memory segments*);
- средства, обеспечивающие возможность «общения» процессов, как родственных, так и нет, через пайпы или каналы (*pipes*).

Наиболее общим понятием *IPC* является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *keyt*, состав которого зависит от реализации и определяется в файле

<*sys/types.h*>. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту.

**Семафоры**

Для работы с семафорами поддерживаются три системных вызова

• *semget* – для создания и получения доступа к набору семафоров;

• *semop* – для манипулирования значениями семафоров (системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);

• *semctl* – для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

Системный вызов *semget* имеет следующий синтаксис

```
int semid = semget (key_t key, int count, int flag)
```

параметрами которого является ключ или уникальное имя сегмента (*key*), набора семафоров и дополнительные флаги (*flag*), определенные в <*sys/ipc.h*>, число семафоров в наборе семафоров (*count*), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров *semid*. Живучесть такого семафора определяется живучестью ядра, т. е. объект семафор будет уничтожен тогда и только тогда, когда произойдет перезагрузка ядра либо его принудительно удалят. После вызова *semget* индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова *semget* приведены ниже в таблице:

Таблица 4.1

*Флаги системного вызова* semge

| Флаг | Описание |
|---|---|
| *IPC_CREAT* | Вызов semget создает новый семафор для данного ключа. Если флаг *IPC_CREAT* не задан, а набор |
| | семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров. |

| | |
|---|---|
| *IPC_EXLC* | Флаг *IPC_EXLC* вместе с флагом *IPC_CREAT* предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, *semget* возвратит -1, а системная переменная errno будет содержать значение *EEXIST*. |

Младшие 9 бит флага задают права доступа к набору семафоров (табл. 4.2).

Таблица 4.2

*Константы режима доступа при создании нового объекта IPC*

| Константа | Описание |
|---|---|
| *S_IRUSR* | Владелец – чтение |
| *S_IWUSR* | Владелец – запись |
| *S_IRGRP* | Группа – чтение |
| *S_IWGRP* | Группа – запись |
| *S_IROTH* | Прочие – чтение |
| *S_IWOTH* | Прочие – запись |

Таким образом, флаг создания семафора можно указать так int flag = S_IRUSR | S_IWUSR | S_IRGRP | IPC_CREAT;

Системный вызов *semctl* имеет формат int semctl (int semid, int sem_num, int command, union semun arg)

где *semid* – это идентификатор набора семафоров, *sem_num* – номер семафора в группе; *command* – код операции; *arg* – указатель на структуру, содержимое которой интерпретируется по разному, в зависимости от операции.

Объединение имеет вид union

```
semun
{
int val; /* устанавливает значение семафора только для SETVAL */  struct
semid_ds *buf;
        /* используется командами IPC_STAT и IPC_SET */ unsigned short
*array; /* используется командами SETALL и GETALL */
};
```

Объединение *semun* всегда должен быть переопределен в глобальной секции программы. Структура *semid_ds* выглядит следующим образом

```
struct semid_ds {
    struct ipc_perm sem_perm; /* разрешения на операции */      struct sem
*sem_base; /* указатель на массив семафоров в наборе */      ushort
sem_nsems;   /* количество семафоров */      time_t sem_otime; /* время
последнего вызова semop() */      time_t sem_ctime; /* время создания
последнего IPC_SET */
};
```

Вызов *semctl* позволяет:

• уничтожить набор семафоров или индивидуальный семафор в указанной группе (*IPCRMID*);

• вернуть значение отдельного семафора (*GETVAL*) или всех семафоров (*GETALL*);

• установить значение отдельного семафора (*SETVAL*) или всех семафоров (*SETALL*);

• вернуть число семафоров в наборе семафоров (*GETPID*).

Основным системным вызовом для манипулирования семафором является int semop (int semid, struct sembuf *op_array, int count)

где *semid* – это ранее полученный дескриптор группы семафоров; *op_array* – массив структур *sembuf* struct sembuf {

```
    short sem_num; /* номер семафора: 0,1,2..n */      short
sem_op;        /* операция с семафором */
    short sem_flg;     /* флаги операции: 0, IPC_NOWAIT, SEM_UNDO */
};
```

определенных в файле *<sys/sem.h>* и содержащих описания операций над семафорами группы, а *count* – количество элементов массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора.

Если указанные в массиве *op_array* номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля «операция». Возможны три случая

1. Отрицательное значение sem_op.
2. Положительное значение sem_op.
3. Нулевое значение sem_op.

Если значение поля операции *sem_op* отрицательно, и его абсолютное значение меньше или равно значению семафора *semval*, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора. Если же значение поля операции *sem_op* по

абсолютной величине больше семафора *semval*, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

Если значение поля операции *sem_op* положительно, то оно прибавляется к значению семафора *semval*, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии *Unix*).

Если значение поля операции *sem_op* равно нулю и значение семафора *semval* также равно нулю, выбирается следующий элемент массива *op_array*. Если же значение семафора *semval* отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания. При использовании флага *IPCNOWAIT* ядро ОС *Unix* не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса при отсутствии флага *IPCNOWAIT*.

**Очереди сообщений**

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами

- *msgget* для образования новой очереди сообщений или получения дескриптора существующей очереди;
- *rnsgsnd* для постановки сообщения в указанную очередь сообщений;
- *rnsgrcv* для выборки сообщения из очереди сообщений;
- *rnsgctl* для выполнения ряда управляющих действий.

Прототипы перечисленных системных вызовов описаны в файлах

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

По системному вызову *msgget* в ответ на ключ (*key*), определяющий уникальное имя очереди, и набор флагов (полностью аналогичны флагам в системном вызове *semget*). Вызовом *msgget* ядро, либо создает новую очередь сообщений в ядре и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений ядра, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди

```
int msgqid = msgget(key_t key, int flag)
```

Таким образом, очередь сообщения обладает живучестью ядра. Для помещения сообщения в очередь служит системный вызов *msgsnd*

int rnsgsnd (int msgqid, void *rnsg, size_t size, int flaq)

где *msg* – это указатель на структуру длиной *size*, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение, причем размер пользовательских данных вычисляется следующим образом: *size = sizeof(msg) – sizeof(long)*.

Структура *msg* всегда имеет вид

```
struct  rnsq {
lonq rntype; /*   тип   сообщения   */ char
mtext[SOMEVALUE]; /*текст сообщения  */
};
```

Поле типа *long* всегда должно быть первым в структуре, далее могут следовать в любом порядке пользовательские данные, в этом случае ядро не накладывает ограничение на тип данных, а только на их длину (зависящую от реализации системы). Параметр *flag* определяет действия ядра для вызвавшего потока при чтении очереди или выходе за пределы допустимых размеров внутренней буферной памяти. Если *flag* = 0, то при отсутствии сообщения в очереди поток блокируется.

Если *flag = IPCNOWAIT*, то поток не блокируется и при отсутствии сообщения возвращается ошибка *ENOMSG*.

Условиями успешной постановки сообщения в очередь являются

- наличие прав процесса по записи в данную очередь сообщений;
- непревышение длиной сообщения заданного системой верхнего предела;
- положительное значение типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг *IPCNOWAIT* при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений, или очередь не будет удалена, или вызвавшей поток не будет прерван перехватываемым сигналом.

Для приема сообщения используется системный вызов *msgrcv*

int rnsgrcv (int msgqid, void *msg, size_t size, long rnsg_type, int flag)

Аргумент *rnsg_type* задает тип сообщения, которое нужно считать из очереди

- если значение равно 0, то возвращается первое сообщение в очереди, т. е. самое старое сообщение;

• если тип больше 0, то возвращается первое сообщение, тип которого равен указанному числу;

• если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше, либо равно модулю указанного числа.

Значение *size* в данном случае указывает ядру, что возвращаемые данные не должны превышать размера указанного в *size*.

Системный вызов *msgctl* позволяет управлять очередями сообщений

int msgctl (int msgqid, int command, struct msgid_ds *msg_stat) и

используется:

• для опроса состояния описателя очереди сообщений (*command = IPCSTAT*) и помещения его в структуру *msgstat*;

• изменения его состояния (*command = IPCSET*), например изменения прав доступа к очереди;

• для уничтожения указанной очереди сообщений (*command = IPCRMID*).

•

**Примеры практической реализации**

**Семафоры.**

Программа *semsyn*, исходный код которой приведен ниже, создает семафор и два процесса, синхронизирующихся с помощью созданного семафора. В программе дочерний процесс является главным, он блокирует и разблокирует семафор, родительский процесс ждет освобождения семафора.

```
#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <fcntl.h> #include
<time.h>
#include <iostream.h>
#define MAXLINE 128
#define SVSEM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define SKEY 1234L // идентификатор семафора union
semun {
```

```
    int val;
    struct semid_ds *buf;     ushort
*array;
}; int var;
int main(int argc, char **argv) { char
filename[] = "./rezult.txt";
pid_t pid; // идентификатор дочернего процесса
time_t ctime; // переменная времени int oflag, c,
semid; struct tm *ctm; union semun arg; struct
semid_ds seminfo; struct sembuf psmb; unsigned
short *prt = NULL;
    var = 0;
    oflag = SVSEM_MODE | IPC_CREAT; // флаг семафора
printf("Parent: Creating semaphore...\n");
    semid = semget(SKEY, 1, oflag); // создание семафора       arg.buf
= &seminfo;
    printf("Parent: Getting info about semaphore (not required, for
example)...\n");       semctl(semid, 0, IPC_STAT, arg); //получение инф. о
семафоре       g.buf->sem_ctime;       ctm = localtime(&ctime);
    printf("%s %d %s %d %s %d %s","Parent: Creating time - ",       ctm-
>tm_hour,":",ctm->tm_min,":",ctm->tm_sec,"\n");
    arg.val = 5;
    printf("%s %d %s","Parent: Setting value \"",arg.val, "\" to
semaphores...\n");       semctl(semid, 0, SETVAL, arg); // установка значения
семафора       printf("Parent: Creating child process...\n"); if ((pid = fork())
== 0) { // child process ;
        printf("    Child: Child process was created...\n");
struct sembuf csmb;          unsigned short semval;          union
semun carg;
        int oflag = SVSEM_MODE | IPC_EXCL;          printf("     Child:
Opening semaphore...\n");          int smd = semget(SKEY, 1, oflag);
// открытие семафора          csmb.sem_num = 0;
csmb.sem_flg = 0;          csmb.sem_op = -1;
        printf("    Child: Locking semaphore...\n");
semop(smd,&csmb,1); // блокировка семафора          printf("
  Child: Do something...\n");
        // работа процесса в защищенном режиме
```

```
        // работа процесса в защищенном режиме закончена
    printf("   Child: Done something...\n");          carg.buf = NULL;
    carg.array = &semval;
        semctl(smd,0,GETALL,carg); // получение значения семафора
    semval = *carg.array;
        printf("%s %d %s","     Child: Semaphore value = ",semval,"\n");
    csmb.sem_num = csmb.sem_flg = 0;          csmb.sem_op = -semval;
        printf("    Child: Unlocking semaphore...\n");          semop(smd,&csmb,1);
        printf("    Child: Terminating child process...\n");          exit(0);
      }

    printf("Parent: Waiting for unlocking semaphore...\n");     psmb.sem_num =
    psmb.sem_flg = psmb.sem_op = 0; semop(semid,&psmb,1);
    printf("Parent:   Semaphore   is   unlocked...\n");
    printf("Parent:   Waiting   for   SIGCHILD...\n");
    waitpid(pid,NULL,0);
    printf("Parent: Deleting semaphore...\n"); semctl(semid, 0,
    IPC_RMID);
    exit(0);
    }
```

Запуск приведенной выше программы происходит следующим образом semsyn

```
Parent: Creating semaphore...
Parent: Getting info about semaphore (not required, for example)...
Parent: Creating time -  13 : 14 : 6
Parent: Setting value " 5 " to semaphore...
Parent: Creating child process...
    Child: Child process was created...
    Child: Opening semaphore...
    Child: Locking semaphore...
    Child: Do something...
Parent: Waiting for unlocking semaphore...
    Child: Done something...
    Child: Semaphore value =  4
    Child: Unlocking semaphore...
Parent: Semaphore is unlocked...
Parent: Waiting for SIGCHILD...
```

Child: Terminating child process...

Parent: Deleting semaphore...

Во время работы программы создается семафор с живучестью ядра

ipcs –s

------ Semaphore Arrays --------

key semid owner perms nsems

0x000004d2 425986 root 644 1

Очереди сообщений

Программа msgcreate создает очередь сообщений. Параметр командной строки e позволяет указать флаг IPC_EXCL. Полное имя файла, являющееся обязательным аргументом командной строки, передается функции ftok. Получаемый ключ преобразуется в идентификатор функцией msgget.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <error.h>
#include <unistd.h>
#include <fcntl.h>
#define SVMSG_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
int c, oflag, mqid;
oflag = SVMSG_MODE | IPC_CREAT;
while ( (c = getopt(argc, argv, "e")) != -1) {
switch (c) {            case 'e':
        oflag |= IPC_EXCL;
break;
    }
}
if (optind != argc - 1)
    {
```

```
            printf("usage: msgcreate [ -e ] <path_to_file>");
return 0;
    }
mqid = msgget(ftok(argv[optind], 0), oflag); return
0;
}
```

**Задания**

1. Ознакомиться с теоретическим материалом.

2. Обеспечить синхронизацию процессов и передачу данных между ними на примере двух приложений «клиент» и «сервер», создав два процесса (два исполняемых файла) – процесс «клиент» (первый исполняемый файл) и процесс «сервер» (второй исполняемый файл). С помощью механизмов межпроцессного взаимодействия обеспечить передачу информации от «клиента» к «серверу» и наоборот. В качестве типа передаваемой информации можно использовать: данные, вводимые с клавиатуры; данные, считываемые из файла; данные, генерируемые случайным образом и т. п.

3.Обмен данными между процессами «клиент»-«сервер» осуществить с использованием программных каналов (именованных либо неименованных), использованием механизмов синхронизации процессов (например с помощью семафоров), очередь сообщений;