МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ

Зав. кафедрой АСУ, профессор

 А.М. Кориков

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Методические рекомендации для выполнения лабораторной работы №3

Тема: «POSIX. Разделяемая память»

Учебно-методическое пособие

для студентов уровеня основной образовательной программы магистратура
направления подготовки 010400.68 «Прикладная математика и информатика»
профиля Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей

Разработчик

доцент кафедры АСУ

В.Г. Резник

2014

Резник В.Г.

Архитектура вычислительных комплексов. Лабораторная работа №3: POSIX. Разделяемая память. Учебно-методическое пособие. – Томск, ТУСУР, 2012. – 29 с.

Учебно-методическое пособие предназначено для выполнения лабораторной работы №3 по дисциплине «Архитектура вычислительных комплексов» для студентов уровеня основной образовательной программы магистратура направления подготовки 010400.68 «Прикладная математика и информатика» профиля «Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей».

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Рассматриваемое учебно-методическое пособие содержит методические рекомендации по написанию программного обеспечения (ПО), при выполнении лабораторных работ по дисциплине АВК - «Архитектура вычислительных комплексов».

Тема лабораторной работы: «POSIX. Разделяемая память».

Знание основ стандарта POSIX входит в набор общекультурных компетенций студента. Умение студента разрабатывать программы, удовлетворяющие стандарту POSIX, составляет основу профессиональных компетенций по этой дисциплине. В целом, лабораторная работа №3 продолжает формирование базовых знаний по разработке ПО на основе вычислительного комплекса УПК АСУ. При изложении материала использовалась информация сайта [1].

Последовательность и изложение материала данного учебного пособия предполагает, что студент:

- успешно выполнил задания по лабораторным работам №1 и №2 по темам: «Подготовка работы на УПК АСУ» и «POSIX. Сигналы процессов»;
- владеет теоретическими знаниями и практическим умением, которые полученны при изучении дисциплины «Современные операционные системы»;
- имеет практические навыки разработки прикладного ПО на языке программирования С.

Первый раздел пособия дает описание программных средств синхронизации процессов, основанных на понятиях стандарта POSIX. Здесь вводится понятие семафора и дается описание конкретной задачи синхронизации, известной как «Задача об обедающих философах».

Во втором разделе, изучаются программные средства, обеспечивающие доступ к разделяемой памяти (разделяемым сегментам памяти).

В третьем разделе, ставится задача по проведению лабораторной работы и дается описание этапов, обеспечивающих успешное выполнение учебного задания.

Четвертый раздел пособия содержит рекомендации по подведению итогов лабораторной работы №3.

1. Средства синхронизации POSIX

Основными (базовыми) средствами синхронизации процессов в языке С являются ***семафоры***.

Согласно определению стандарта POSIX-2001, **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 диапазоне от 0 до 32767.

Семафоры создаются функцией **semget()** и обрабатываются функцией **semop()** над **наборами** (массивами). Операции над этими наборами, с точки зрения приложений, являются атомарными.

В рамках ***групповых операций*** для любого семафора из набора можно сделать следующее:

1. ***увеличить значение,***
2. ***уменьшить значение,***
3. ***дождаться обнуления***.

Процессы, обладающие соответствующими правами, также могут выполнять различные управляющие действия над семафорами. Для этого служит функция **semctl()**.

Описание перечисленных функций имеет вид:

```
#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, ...);
```

Структура **semid_ds**, ассоциированная с идентификатором набора семафоров, должна содержать по крайней мере следующие поля.

```
/* Данные о правах доступа к набору семафоров */
struct ipc_perm sem_perm;

/* Число семафоров в наборе */
unsigned short  sem_nsems;
/* Время последней операции semop() */
time_t          sem_otime;
/* Время последнего изменения посредством semctl() */
time_t          sem_ctime;
```

Отдельные семафоры из набора представляются ***безымянной структурой***, состоящей по крайней мере из следующих полей.

```
unsigned short semval;
/* Значение семафора */
pid_t          sempid;
/* Идентификатор процесса, выполнившего
последнюю операцию над семафором */
unsigned short semncnt;
/* Число процессов, ожидающих увеличения
текущего значения семафора */
unsigned short semzcnt;
/* Число процессов, ожидающих обнуления
значения семафора */
```

Функция **semget()** аналогична **msgget()**; аргумент ***nsems*** задает число семафоров в наборе.

Структура **semid_ds** инициализируется так же, как **msqid_ds**.

Безымянные структуры, соответствующие отдельным семафорам, функцией **semget()** не инициализируются.

Операции, выполняемые посредством функции **semop()**, задаются массивом ***sops*** с числом элементов ***nsops***, состоящим из структур типа ***sembuf***, каждая из которых содержит по крайней мере следующие поля.

```
unsigned short sem_num;
/* Номер семафора в наборе (нумерация с нуля) */
short          sem_op;
/* Запрашиваемая операция над семафором */
short          sem_flg;
/* Флаги операции */
```

Операция над семафором определяется значением поля ***sem_op***:

- положительное значение предписывает увеличить значение семафора на указанную величину,
- отрицательное - уменьшить, нулевое - сравнить с нулем.
- Вторая операция не может быть успешно выполнена, если в результате значение семафора становится отрицательным,
- а третья - если значение семафора ненулевое.

Выполнение массива операций с точки зрения пользовательского процесса является неделимым действием. Это значит:

- **во-первых**, что если операции выполняются, то только все вместе;
- **во-вторых**, никакой другой процесс не может получить доступ к промежуточному состоянию набора семафоров, когда часть операций из массива уже выполнилась, а другая еще не успела.

Сама операционная система выполняет операции из массива по очереди, причем порядок не оговаривается. Если очередная операция не может быть выполнена, то эффект предыдущих аннулируется, а вызов функции **semop()** приостанавливается (***операция с блокировкой***) или немедленно завершается неудачей (***операция без блокировки***).

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

Следующий пример демонстрирует массив операций, который задает уменьшение (**с блокировкой**) семафора 1 при условии, что значение семафора 0 равно нулю.

```
sembuf [0].sem_num = 1;
sembuf [0].sem_flg = 0;
sembuf [0].sem_op = -2;

sembuf [1].sem_num = 0;
sembuf [1].sem_flg = IPC_NOWAIT;
sembuf [1].sem_op =  0;
```

Обращаясь к функции **semctl()**, процессы могут получать информацию о состоянии набора семафоров, изменить ряд его характеристик, удалить набор.
Аргументы ***semid*** (идентификатор набора семафоров) и ***semnum*** (номер семафора в наборе) определяют объект, над которым выполняется управляющее действие, задаваемое значением аргумента ***cmd***.
Если объектом является набор, значение **semnum** игнорируется.
Для некоторых действий задействован четвертый аргумент:

```
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
} arg;
```

Среди допустимых действий:

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

Имеются и аналогичные групповые действия:

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

Предусмотрены действия, позволяющие выяснить идентификатор процесса, выполнившего последнюю операцию над семафором (**GETPID**), а также число процессов, ожидающих ***увеличения*** / ***обнуления*** (**GETNCNT** / **GETZCNT**) значения семафора (информация о процессах выдается в качестве результата:

```
val = semctl (semid, semnum, GETVAL);

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

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);

lpid = semctl (semid, semnum, GETPID);

ncnt = semctl (semid, semnum, GETNCNT);

zcnt = semctl (semid, semnum, GETZCNT);
```

Наконец, для семафоров, как и для очередей сообщений, определены управляющие команды:

- **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);
```

В качестве примера использования семафоров рассмотрим известную задачу «***Задачу об обедающих философах***»:

- За круглым столом сидит несколько философов.
- В каждый момент времени каждый из них либо беседует, либо ест.
- Для еды одновременно требуется две вилки. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу надо дождаться, 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 беседе.
- Требуется разработать программную модель обеда философов.

Главное в этой задаче - корректная дисциплина захвата и освобождения вилок, **иначе**, каждый из философов одновременно с другими возьмется за вилку, лежащую слева от него, и будет ждать освобождения правой, обед не завершится никогда.

Предлагаемое решение состоит из двух программ.

Первая программа, приведенная ниже, реализует процесс-монитор, который:

- порождает набор семафоров (по одному семафору на каждую вилку);
- устанавливает начальные значения семафоров (занятой вилке будет соответствовать значение 0, свободной — 1);
- запускает несколько процессов, представляющих философов, указывая место за столом (в качестве одного из аргументов передается число от 1 до ***QPH***);
- ожидает, пока все процессы завершатся (все философы съедят свой обед);
- удаляет набор семафоров.

Предполагается (для нужд функции **ftok()**), что исходный текст программы находится в некотором файле ***phdin.c*** (точнее, что такой файл существует).

```
#include <unistd.h>
#include <stdio.h>
#include <sys/sem.h>
#include <sys/wait.h>

/* Программа-монитор обеда философов */

#define QPH 5

#define ARG_SIZE        20

int main (void) {
 int key;       /* Ключ набора семафоров */
 int semid;     /* Идентификатор набора семафоров */
 int no;        /* Номер философа и/или вилки */
 char ssemid [ARG_SIZE], sno [ARG_SIZE], sqph [ARG_SIZE];

                /* Создание и инициализация набора семафоров */
                /* (по семафору на вилку)                    */
 key = ftok ("phdin.c", 'C');
 if ((semid = semget (key, QPH, 0600 | IPC_CREAT)) < 0) {
   perror ("SEMGET");
   return (1);
 }
 for (no = 0;  no < QPH; no++) {
   if (semctl (semid, no, SETVAL, 1) < 0) {
     perror ("SETVAL");
     return (2);
   }
 }

 sprintf (ssemid, "%d", semid);
 sprintf (sqph, "%d", QPH);

                /* Все - к столу */
 for (no = 1;  no <= QPH; no++) {
   switch (fork ()) {
     case -1:
       perror ("FORK");
       return (3);
     case 0:
```

```
      sprintf (sno, "%d", no);
      execl ("./phil", "phil", ssemid, sqph, sno, (char *) 0);
      perror ("EXEC");
      return (4);
  }
 }

                /* Ожидание завершения обеда */
 for (no = 1; no <= QPH; no++) {
  (void) wait (NULL);
 }

                /* Удаление набора семафоров */
 if (semctl (semid, 0, IPC_RMID) < 0) {
  perror ("SEMCTL");
  return (5);
 }

 return 0;
}
```

Вторая программа описывает обед каждого философа:

- Философ какое-то время беседует (случайное значение **trnd**), затем пытается взять вилки слева и справа от себя;
- когда ему это удается, некоторое время ест (случайное значение **ernd**), после чего освобождает вилки.
- Так продолжается до тех пор, пока не будет съеден весь обед.

Предполагается, что выполнимый файл программы называется ***phil***.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>

/* Процесс обеда одного философа */

#define ernd (rand () % 3 + 1)
#define trnd (rand () % 5 + 1)
#define FO   15

int main (int argc, char *argv []) {
 int semid;    /* Идентификатор набора семафоров */
 int qph;      /* Число философов */
 int no;       /* Номер философа */
 int t;        /* Время очередного отрезка еды или беседы */
 int fo;       /* Время до конца обеда */
 struct sembuf sembuf [2];

 if (argc != 4) {
       fprintf  (stderr,  "Использование:  %s  идентификатор_набора_семафоров
число_философов номер_философа \n", argv [0]);
  return (1);
 }

 fo = FO;
 sscanf (argv [1], "%d", &semid);
 sscanf (argv [2], "%d", &qph);
```

```
 sscanf (argv [3], "%d", &no);

                                  /* Выбор вилок */
 sembuf [0].sem_num = no - 1;     /* Левая */
 sembuf [0].sem_flg = 0;
 sembuf [1].sem_num = no % qph;   /* Правая */
 sembuf [1].sem_flg = 0;

 while (fo > 0) {                 /* Обед */

                          /* Философ говорит */
   printf ("Философ %d беседует\n", no);
   t = trnd; sleep (t); fo -= t;
                          /* Пытается взять вилки */
   sembuf [0].sem_op = -1;
   sembuf [1].sem_op = -1;
   if (semop (semid, sembuf, 2) < 0) {
     perror ("SEMOP");
     return (1);
   }

                          /* Ест */
   printf ("Философ %d ест\n", no);
   t = ernd; sleep (t); fo -= t;
                          /* Отдает вилки */
   sembuf [0].sem_op = 1;
   sembuf [1].sem_op = 1;
   if (semop (semid, sembuf, 2) < 0) {
     perror ("SEMOP");
     return (2);
   }
 }

 printf ("Философ %d закончил обед\n", no);
 return 0;
}
```

Отметим, что возможность выполнения групповых операций над семафорами предельно упростила решение, сделав его прямолинейным, очевидным образом гарантирующим отсутствие тупиков.

Второй вариант решения этой задачи, предложенный С.В. Самборским. В нем реализованы четыре стратегии захвата вилок, которые сравниваются по результатам моделирования поведения философов в течение нескольких минут. Все стратегии гарантируют отсутствие тупиков, но только две из них, соответствующие опциям ***-a*** и ***-p***, заведомо не позволят ни одному философу умереть от голода из-за невозможности получить обе вилки сразу. (Это свойство "стратегий ***-a*** и ***-p*** " является следствием упорядоченности ресурсов.)

```
/* Обедающие философы. Запуск:
    mudrecProc [-a | -p | -I -V] [-t число_секунд] имя_философа ...
Опции:
   -t число_секунд - сколько секунд моделируется
   Стратегии захвата вилок:
   -a - сначала захватывается вилка с меньшим номером;
   -I - некорректная (но эффективная) интеллигентная стратегия: во время
        ожидания уже захваченная вилка кладется;
```

```
   -p - сначала захватывается нечетная вилка;
   -V - использован групповой захват семафоров.

Пример запуска:      mudrecProc -p -t 600  A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z
*/

static char rcsid[] __attribute__((unused)) = \
"$Id: mudrecProc.c,v 1.7 2003/11/11 13:14:07 sambor Exp $";

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include <limits.h>
#include <errno.h>

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

union semun {
 int val;
 struct semid_ds *buf;
 unsigned short *array;
} arg;

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)>(b)?(b):(a))

struct mudrec {
 long num;
 char *name;
 int left_fork, right_fork;
 int eat_time, wait_time, think_time, max_wait_time;
 int count;
};

int Stop;                    /* Семафор для синхронизации выхода */

/* Различные дескрипторы */
int protokol [2] = {-1, -1};
#define pFdIn (protokol [1])
#define pFdOut (protokol [0])

int semFork;    /* Вилки */

int from_fil;  /* Очередь для возврата результатов */

/* Разные алгоритмы захвата вилок */
static void get_forks_simple (struct mudrec *this);
static void get_forks_parity (struct mudrec *this);
static void get_forks_maybe_infinit_time (struct mudrec *this);
static void get_forks_use_groups (struct mudrec *this);

/* Используемый метод захвата вилок */
void (*get_forks) (struct mudrec *this) = get_forks_simple;

/* Возвращение вилок */
static void put_forks (struct mudrec *this);
```

```
/*
* Философы
*/
void filosof (struct mudrec this) {
 char buffer [LINE_MAX];
 int bytes;

 if (fork ()) return;

 srandom (getpid ()); /* Очень важно для процессов, иначе получим одно и то же!
*/
 random (); random (); random (); random (); random ();
 random (); random (); random (); random (); random ();

 /* Пока семафор Stop не поднят */
 while (!semctl (Stop, 0, GETVAL)) {
   /* Пора подкрепиться */
   {
     int wait_time, tm;

     sprintf (buffer, "%s: хочет есть\n", this.name);
     bytes = write (pFdIn, buffer, strlen (buffer));

     tm = time (0);

     (*get_forks) (&this);

     wait_time = time (0) - tm;    /* Сколько времени получали вилки */
     this.wait_time += wait_time;
     this.max_wait_time = max (wait_time, this.max_wait_time);

     sprintf (buffer, "%s: ждал вилок %d сек\n", this.name, wait_time);
     bytes = write (pFdIn, buffer, strlen (buffer));
   }

   /* Может, обед уже закончился? */
   if (semctl (Stop, 0, GETVAL)) {
     put_forks (&this);
     break;
   }

   /* Едим */
   {
     int eat_time = random () % 20 + 1;

     sleep (eat_time);

     this.eat_time += eat_time;
     this.count++;
     sprintf (buffer,"%s: ел %d сек\n", this.name, eat_time);
     bytes = write (pFdIn, buffer, strlen (buffer));
   }

   /* Отдаем вилки */
   put_forks (&this);

   if (semctl (Stop, 0, GETVAL)) break;

   /* Размышляем */
   {
     int think_time = random () % 10 + 1;
```

```
    sleep (think_time);

    this.think_time += think_time;
   }
 }

 sprintf (buffer,"%s: уходит\n", this.name);
 bytes = write (pFdIn, buffer, strlen (buffer));

 msgsnd (from_fil, &this, sizeof (this), 0);   /* Отослали статистику своего
обеда */

 _exit (0);    /* ВАЖНО (_): Нам не нужны преждевременные вызовы cleanup_ipc */
}

/* Кладем вилки одну за другой */
static void put_forks (struct mudrec *this) {
 struct sembuf tmp_buf;

 tmp_buf.sem_flg = 0;
 tmp_buf.sem_op = 1;
 tmp_buf.sem_num = this->left_fork - 1;
 semop (semFork, &tmp_buf, 1);

 tmp_buf.sem_flg = 0;
 tmp_buf.sem_op = 1;
 tmp_buf.sem_num = this->right_fork - 1;
 semop (semFork, &tmp_buf, 1);
}

/* Берем вилки по очереди в порядке номеров */
static void get_forks_simple (struct mudrec *this) {
 struct sembuf tmp_buf;

 int first = min (this->left_fork, this->right_fork);
 int last = max (this->left_fork, this->right_fork);

 tmp_buf.sem_flg = SEM_UNDO;
 tmp_buf.sem_op = -1;
 tmp_buf.sem_num = first - 1;
 semop (semFork, &tmp_buf, 1);

 tmp_buf.sem_flg = SEM_UNDO;
 tmp_buf.sem_op = -1;
 tmp_buf.sem_num = last - 1;
 semop (semFork, &tmp_buf, 1);
}

/* Берем сначала нечетную вилку (если обе нечетные - то с большим номером) */
static void get_forks_parity (struct mudrec *this) {
 struct sembuf tmp_buf;

 int left = this->left_fork, right = this->right_fork;
 int first = max ((left & 1) * 1000 + left, (right & 1) * 1000 + right) % 1000;
 int last = min ((left & 1) * 1000 + left, (right & 1) * 1000 + right) % 1000;

 tmp_buf.sem_flg = SEM_UNDO;
 tmp_buf.sem_op = -1;
 tmp_buf.sem_num = first - 1;
 semop (semFork, &tmp_buf, 1);
```

```
 tmp_buf.sem_flg = SEM_UNDO;
 tmp_buf.sem_op = -1;
 tmp_buf.sem_num = last - 1;
 semop (semFork, &tmp_buf, 1);
}

/* Берем вилки по очереди, в произвольном порядке.
* Но если вторая вилка не берется сразу, то кладем первую.
* То есть философ не расходует вилочное время впустую.
*/
static void get_forks_maybe_infinit_time (struct mudrec *this) {
 struct sembuf tmp_buf;

 int left = this->left_fork, right = this->right_fork;

 for (;;) {
   tmp_buf.sem_flg = SEM_UNDO;   /* Первую вилку берем с ожиданием */
   tmp_buf.sem_op = -1;
   tmp_buf.sem_num = left - 1;
   semop (semFork, &tmp_buf, 1);

   tmp_buf.sem_flg = SEM_UNDO | IPC_NOWAIT;  /* Вторую - без ожидания */
   tmp_buf.sem_op = -1;
   tmp_buf.sem_num = right - 1;

   if (0 == semop (semFork, &tmp_buf, 1)) return;   /* Успех */

   tmp_buf.sem_flg = 0;             /* Неуспех: возвращаем первую вилку */
   tmp_buf.sem_op = 1;
   tmp_buf.sem_num = left - 1;
   semop(semFork,&tmp_buf,1);

   tmp_buf.sem_flg = SEM_UNDO;      /* Отдав первую, ждем вторую */
   tmp_buf.sem_op = -1;
   tmp_buf.sem_num = right - 1;
   semop (semFork, &tmp_buf, 1);

   tmp_buf.sem_flg = SEM_UNDO | IPC_NOWAIT;   /* Берем первую вилку без ожидания
*/
   tmp_buf.sem_op = -1;
   tmp_buf.sem_num = left - 1;

   if (0 == semop (semFork, &tmp_buf, 1)) return;   /* Успех */

   tmp_buf.sem_flg =  0;             /* Неуспех: отдаем вторую вилку, */
   tmp_buf.sem_op = 1;               /* чтобы ждать первую            */
   tmp_buf.sem_num = right - 1;
   semop (semFork, &tmp_buf, 1);
 }
}

/* Хватаем обе вилки сразу, используя групповые операции */
static void get_forks_use_groups (struct mudrec *this) {
 struct sembuf tmp_buf [2];

 tmp_buf[0].sem_flg = SEM_UNDO;
 tmp_buf[0].sem_op = -1;
 tmp_buf[0].sem_num = this->left_fork - 1;
 tmp_buf[1].sem_flg = SEM_UNDO;
 tmp_buf[1].sem_op = -1;
 tmp_buf[1].sem_num = this->right_fork - 1;
 semop (semFork, tmp_buf, 2);
```

```
}

/*
* Мелкие служебные функции.
*/
static void stop (int dummy) {
 struct sembuf tmp_buf;

 tmp_buf.sem_flg = 0;
 tmp_buf.sem_op = 1;
 tmp_buf.sem_num = 0;
 semop (Stop, &tmp_buf, 1);
}

void cleanup_ipc (void) {
/*
* Уничтожение семафоров.
*/
 semctl (semFork, 1, IPC_RMID);
 semctl (Stop, 1, IPC_RMID);

 /* То же с очередью */
 msgctl (from_fil, IPC_RMID, NULL);
}

static void usage (char name []) {
  fprintf (stderr,"Использование: %s [-a | -p | -I| -V] [-t число_секунд]
имя_философа ...\n", name);
 exit (1);
}

/*
* Точка входа демонстрационной программы.
*/
int main (int argc, char *argv[]) {
 char buffer [LINE_MAX], *p;
 int i, n, c;
 int open_room_time = 300;
 union semun tmp_arg;
 int nMudr;
 struct sigaction sact;

 while ((c = getopt (argc, argv, "apIVt:")) != -1) {
   switch (c) {
     case 'a': get_forks = get_forks_simple; break;
     case 'p': get_forks = get_forks_parity; break;
     case 'I': get_forks = get_forks_maybe_infinit_time; break;
     case 'V': get_forks = get_forks_use_groups; break;
     case 't': open_room_time = strtol (optarg, &p, 0);
                if (optarg [0] == 0 || *p != 0) usage (argv [0]);
                break;
     default: usage (argv [0]);
   }
 }

 nMudr = argc - optind;
 if (nMudr < 2) usage (argv [0]);  /* Меньше двух философов неинтересно ... */

 /*
  * Создание канала для протокола обработки событий
  */
 pipe (protokol);
```

```
/*
 * Создадим семафоры для охраны вилок
 */
semFork = semget (ftok (argv [0], 2), nMudr, IPC_CREAT | 0777);
tmp_arg.val = 1;
for (i=1; i <= nMudr; i++)
  semctl (semFork, i - 1, SETVAL, tmp_arg);   /* Начальное значение 1 */

/* Прежде чем впускать философов, обеспечим окончание обеда */
Stop = semget (ftok (argv [0], 3), 1, IPC_CREAT | 0777);
tmp_arg.val = 0;
semctl (Stop, 0, SETVAL, tmp_arg);    /* Начальное значение 0 */

/* Очередь для возврата результатов */
from_fil = msgget (ftok (argv [0], 4), IPC_CREAT | 0777);

atexit (cleanup_ipc); /* Запланировали уничтожение семафоров */
               /* и других средств межпроцессного взаимодействия */

/*
 * Философы входят в столовую
 */
for (i = 0; i < nMudr; i++, optind++) {
  struct mudrec next;

  memset (&next, 0, sizeof (next));

  next.num = i + 1;           /* Номер */
  next.name = argv [optind];  /* Имя   */

  /* Указали, какими вилками пользоваться */
  next.left_fork = i + 1;
  next.right_fork = i + 2;
  if (i == nMudr - 1)
    next.right_fork = 1;  /* Последний пользуется вилкой первого */

  filosof (next);
}

/* Зададим реакцию на сигналы и установим будильник на конец обеда */
sact.sa_handler = stop;
(void) sigemptyset (&sact.sa_mask);
sact.sa_flags = 0;
(void) sigaction (SIGINT, &sact, (struct sigaction *) NULL);
(void) sigaction (SIGALRM, &sact, (struct sigaction *) NULL);

alarm (open_room_time);

/*
 * Выдача сообщений на стандартный вывод и выход после окончания обеда.
 */
close (pFdIn);  /* Сами должны закрыть, иначе из цикла не выйдем! */
for (;;) {
  n = read (pFdOut, buffer, LINE_MAX);
  if ((n == 0) || ((n == -1) && (errno != EINTR))) break;
  for (i = 0; i < n; i++) putchar (buffer [i]);
}
close (pFdOut);

/* Распечатали сводную информацию */
{
```

```
   int full_eating_time = 0;
   int full_waiting_time = 0;
   int full_thinking_time = 0;
   for (i = 1; i <= nMudr; i++) {
     struct mudrec this;
     /* Получили статистику обеда */
     msgrcv (from_fil, &this, sizeof (this), i, 0); /* За счет i получаем */
                                      /* строго по порядку  */
     full_eating_time += this.eat_time;
     full_waiting_time += this.wait_time;
     full_thinking_time += this.think_time;

     if (this.count > 0) {
        float count = this.count;
        float think_time = this.think_time / count;
        float eat_time = this.eat_time / count;
        float wait_time = this.wait_time / count;

          printf ("%s: ел %d раз     в среднем: думал=%.1f   ел=%.1f   ждал=%.1f
(максимум %d)\n",
                     this.name,  this.count,  think_time,  eat_time,  wait_time,
this.max_wait_time);
     }
     else
        printf("%s: не поел\n", this.name);
   }
   {
     float total_time = (full_eating_time + full_waiting_time
                                    + full_thinking_time) / (float)nMudr;

      printf ("    Среднее число одновременно едящих = %.3f\n    Среднее число
одновременно ждущих = %.3f\n",
        full_eating_time / total_time, full_waiting_time / total_time);
   }
 }

 /* Сообщим об окончании работы */
 printf ("Конец обеда\n");

 return 0;
}
```

Получит ли, в конце концов, философ вилки при групповых операциях (опция -V), зависит от реализации. Может случиться так, что хотя бы одна из них в каждый момент времени будет в руках у одного из соседей. То же верно и для "интеллигентной" стратегии (опция -I). Тем не менее, результаты моделирования показывают, что на практике две последние стратегии эффективнее в смысле минимизации времени ожидания вилок.

Отметим небольшие терминологические различия в двух приведенных вариантах решения задачи об обедающих философах. Во втором варианте явно выделены начальные и конечные моделируемые события - вход философов в столовую и выход из нее (в первом варианте они просто сидят за столом).

С методической точки зрения второй вариант интересен тем, что в нем использованы все рассмотренные нами средства межпроцессного

взаимодействия - каналы, сигналы, очереди сообщений и, конечно, семафоры. (Тонкость: флаг SEM_UNDO обеспечивает корректировку значения семафора при завершении процесса.)

Ниже приведена статистика поведения пяти философов для всех четырех стратегий при времени моделирования 100 секунд.

Результаты говорят в пользу групповых операций над семафорами:

```
-a:
A: ел 2 раза    в среднем: думал=3.5  ел=11.5  ждал=36.5 (максимум 73)
B: ел 3 раза    в среднем: думал=5.7  ел=7.7  ждал=20.0 (максимум 41)
C: ел 3 раза    в среднем: думал=5.7  ел=11.3  ждал=17.0 (максимум 33)
D: ел 3 раза    в среднем: думал=1.7  ел=16.7  ждал=15.7 (максимум 19)
E: ел 1 раз    в среднем: думал=10.0  ел=20.0  ждал=73.0 (максимум 41)
  Среднее число одновременно едящих = 1.471
  Среднее число одновременно ждущих = 2.980
-p:
A: ел 3 раза    в среднем: думал=3.7  ел=15.3  ждал=16.0 (максимум 34)
B: ел 4 раза    в среднем: думал=5.0  ел=13.8  ждал=8.2 (максимум 15)
C: ел 3 раза    в среднем: думал=6.7  ел=3.7  ждал=25.7 (максимум 27)
D: ел 4 раза    в среднем: думал=5.8  ел=8.5  ждал=13.8 (максимум 28)
E: ел 3 раза    в среднем: думал=5.3  ел=15.3  ждал=16.7 (максимум 29)
  Среднее число одновременно едящих = 1.761
  Среднее число одновременно ждущих = 2.413

-I:
A: ел 5 раз    в среднем: думал=4.2  ел=9.4  ждал=6.6 (максимум 15)
B: ел 3 раза    в среднем: думал=6.3  ел=10.3  ждал=17.0 (максимум 31)
C: ел 4 раза    в среднем: думал=6.8  ел=7.0  ждал=12.2 (максимум 45)
D: ел 3 раза    в среднем: думал=4.3  ел=16.0  ждал=13.0 (максимум 16)
E: ел 4 раза    в среднем: думал=5.8  ел=8.5  ждал=10.8 (максимум 22)
  Среднее число одновременно едящих = 1.858
  Среднее число одновременно ждущих = 2.125

-V:
A: ел 5 раз    в среднем: думал=5.6  ел=5.6  ждал=8.8 (максимум 17)
B: ел 3 раза    в среднем: думал=6.3  ел=10.3  ждал=16.7 (максимум 20)
C: ел 4 раза    в среднем: думал=4.8  ел=11.0  ждал=9.8 (максимум 18)
D: ел 4 раза    в среднем: думал=5.2  ел=12.0  ждал=8.8 (максимум 15)

E: ел 4 раза    в среднем: думал=5.2  ел=10.5  ждал=10.2 (максимум 20)

  Среднее число одновременно едящих = 1.892
  Среднее число одновременно ждущих = 2.049
```

2. Разделяемые сегменты памяти

В стандарте POSIX-2001 разделяемый объект памяти определяется как **объект, представляющий собой память**, которая может быть параллельно отображена в адресное пространство более чем одного процесса.

Таким образом, процессы могут иметь общие области виртуальной памяти и разделять содержащиеся в них данные.

Единицей разделяемой памяти являются сегменты.

Разделение памяти обеспечивает наиболее быстрый обмен данными между процессами.

Работа с разделяемой памятью начинается с того, что один из взаимодействующих процессов посредством функции **shmget()** создает разделяемый сегмент, специфицируя первоначальные права доступа к нему и его размер в байтах.

Чтобы получить доступ к разделяемому сегменту, его нужно присоединить (для этого служит функция **shmat()**), т. е. разместить сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их (быть может, синхронизируя свои действия с помощью семафоров). Когда разделяемый сегмент становится ненужным, его следует отсоединить с помощью функции **shmdt()**.

Предусмотрена возможность выполнения управляющих действий над разделяемыми сегментами (функция **shmctl()**).

Описание перечисленных функций имеет вид:

```
#include <sys/shm.h>
int shmget (key_t key, size_t size,
            int shmflg);
void *shmat (int shmid, const void *shmaddr,
             int shmflg);
int shmdt (const void *shmaddr);
int shmctl (int shmid, int cmd,
            struct shmid_ds *buf);
```

Структура **shmid_ds**, ассоциированная с идентификатором разделяемого сегмента памяти, должна содержать по крайней мере следующие поля.

```
struct ipc_perm shm_perm;
/* Данные о правах доступа к разделяемому
сегменту */
size_t          shm_segsz;
/* Размер сегмента в байтах */
pid_t           shm_lpid;
/* Идентификатор процесса, выполнившего
последнюю операцию над разделяемым сегментом */
pid_t           shm_cpid;
/* Идентификатор процесса, создавшего
разделяемый сегмент */
shmatt_t        shm_nattch;
/* Текущее число присоединений сегмента */
time_t          shm_atime;
/* Время последнего присоединения */
time_t          shm_dtime;
/* Время последнего отсоединения */
time_t          shm_ctime;
/* Время последнего изменения посредством
shmctl() */
```

Функция **shmget()** аналогична **msgget()** и **semget()**; аргумент **size** задает нижнюю границу размера сегмента в байтах; реализация, учитывающая, например, правила выравнивания, имеет право создать разделяемый сегмент большего размера.

Структура **shmid_ds** инициализируется в соответствии с общими для средств межпроцессного взаимодействия правилами.

Поле **shm_segsz** устанавливается равным значению аргумента **size**.

Число уникальных идентификаторов разделяемых сегментов памяти ограничено; попытка его превышения ведет к неудачному завершению **shmget()** (возвращается -1).

Вызов **shmget()** завершится неудачей и тогда, когда значение аргумента size меньше минимально допустимого либо больше максимально допустимого размера разделяемого сегмента.

Чтобы присоединить разделяемый сегмент, используется функция **shmat()**. Аргумент **shmid** задает идентификатор разделяемого сегмента; аргумент **shmaddr** - адрес, по которому сегмент должен быть присоединен, т. е. тот адрес в виртуальном пространстве процесса, который получит начало сегмента.

Поскольку свойства сегментов зависят от аппаратных особенностей управления памятью, не всякий адрес является приемлемым. Если установлен флаг SHM_RND, адрес присоединения округляется до величины, кратной константе SHMLBA.

Если **shmaddr** задан как пустой указатель, реализация выбирает адрес присоединения по своему усмотрению.

По умолчанию присоединяемый сегмент будет доступен и на чтение, и на запись (если процесс обладает необходимыми правами). Флаг SHM_RDONLY предписывает присоединить сегмент только для чтения.

При успешном завершении функции **shmat()** результат равен адресу, который получил присоединенный сегмент ; в случае неудачи возвращается -1. (Разумеется, для использования результата **shmat()** в качестве указателя его нужно преобразовать к требуемому типу.)

Отсоединение сегментов производится функцией **shmdt()**; аргумент **shmaddr** задает начальный адрес отсоединяемого сегмента.

Управление разделяемыми сегментами осуществляется при помощи функции **shmctl()**, аналогичной **msgctl()**. Как и для очередей сообщений, для разделяемых сегментов определены управляющие команды IPC_STAT (получить информацию о состоянии разделяемого сегмента), IPC_SET (переустановить характеристики), IPC_RMID (удалить разделяемый сегмент). Удалять сегмент нужно после того, как от него отсоединились все процессы.

Аппарат разделяемых сегментов предоставляет нескольким процессам возможность одновременного доступа к общей области памяти. Обеспечивая корректность доступа, процессы тем или иным способом должны синхронизировать свои действия. В качестве средства синхронизации удобно использовать семафор.

Далее показана реализация так называемого **критического интервала** - механизма, обеспечивающего взаимное исключение разделяющих общие данные процессов.

Для "создания" подобного механизма необходимо породить разделяемый сегмент памяти, присоединить его во всех процессах, которым предоставляется доступ к разделяемым данным, а также породить и проинициализировать простейший семафор. После этого монопольный доступ к разделяемой структуре обеспечивается применением P- и V-операций.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>

int main (void) {
 struct region {
   pid_t fpid;
 } *shm_ptr;
```

```
struct sembuf P = {0, -1, 0};
struct sembuf V = {0,  1, 0};

int shmid;
int semid;

shmid = shmget (IPC_PRIVATE, sizeof (struct region), 0777);
semid = semget (IPC_PRIVATE, 1, 0777);
(void) semctl (semid, 0, SETVAL, 1);

switch (fork ()) {
  case -1:
    perror ("FORK");
    return (1);
  case 0:
    if ((int) (shm_ptr = (struct region *) shmat (shmid, NULL, 0)) == (-1)) {
       perror ("CHILD-SHMAT");
       return (2);
    }

    if (semop (semid, &p, 1) != 0) {
       perror ("CHILD-SEMOP-P");
       return (3);
    }
    printf ("Процесс-потомок вошел в критический интервал\n");

    shm_ptr->fpid = getpid ();         /* Монопольный доступ */

    printf ("Процесс-потомок перед выходом из критического интервала\n");
    if (semop (semid, &V, 1) != 0) {
       perror ("CHILD-SEMOP-V");
       return (4);
    }

    (void) shmdt (shm_ptr);
    return 0;
}

if ((int) (shm_ptr = (struct region *) shmat (shmid, NULL, 0)) == (-1)) {
  perror ("PARENT-SHMAT");
  return (2);
}

if (semop (semid, &p, 1) != 0) {
  perror ("PARENT-SEMOP-P");
  return (3);
}
printf ("Родительский процесс вошел в критический интервал\n");

shm_ptr->fpid = getpid ();         /* Монопольный доступ */

printf ("Родительский процесс перед выходом из критического интервала\n");
if (semop (semid, &V, 1) != 0) {
  perror ("PARENT-SEMOP-V");
  return (4);
}

(void) wait (NULL);

printf ("Идентификатор родительского процесса: %d\n", getpid ());
printf ("Идентификатор процесса в разделяемой структуре: %d\n", shm_ptr->fpid);
```

```
 (void) shmdt (shm_ptr);

 (void) semctl (semid, 1, IPC_RMID);
 (void) shmctl (shmid, IPC_RMID, NULL);

 return 0;
}
```

Результат работы данной программы:

```
Родительский процесс вошел в критический интервал
Родительский процесс перед выходом из критического интервала
Процесс-потомок вошел в критический интервал
Процесс-потомок перед выходом из критического интервала
Идентификатор родительского процесса: 2161
Идентификатор процесса в разделяемой структуре: 2162
```

Рассмотрим пример использования разделяемых сегментов памяти в сочетании с обработкой сигнала SIGSEGV, который посылается процессу при некорректном обращении к памяти.
Идея в том, чтобы создавать разделяемые сегменты, "накрывающие" запрашиваемые адреса.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Реализация "виртуальной" памяти из одного сегмента. */
/* Используются разделяемые сегменты памяти            */
/* и обработка сигнала SIGSEGV                         */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <signal.h>

/* Константа, зависящая от реализации */
#define SHM_BASE_ADDR   0x40014000

static int shm_id = -1;
static void *shm_addr;

/* Реакция на сигнал SIGSEGV.                              */
/* Создаем и присоединяем на чтение разделяемый сегмент,   */
/* накрывающий переданный адрес.                           */
/* Если это не помогло, переприсоединяем сегмент на запись */
static void sigsegv_sigaction (int sig, siginfo_t *sig_info, void *addr) {
 struct shmid_ds shmid_ds;

 if (shm_id == -1) {
   /* Сегмента еще нет. Создадим */
   if ((shm_id = shmget (IPC_PRIVATE, SHMLBA, S_IRUSR)) == -1) {
     perror ("SHMGET");
     exit (1);
   }
   /* Присоединим сегмент на чтение */
     if ((int) (shm_addr = shmat (shm_id, sig_info->si_addr, SHM_RDONLY |
SHM_RND)) == (-1)) {
     perror ("SHMAT-RDONLY");
```

```
    exit (2);
   }
   return;
 } else {
   /* Сегмент уже есть, но обращение по адресу вызвало сигнал SIGSEGV. */
   /* Значит, это была попытка записи, и сегмент нужно                  */
   /* переприсоединить на запись, поменяв соответственно режим доступа */
   if (shmctl (shm_id, IPC_STAT, &shmid_ds) == -1) {
     perror ("SHMCTL-IPC_STAT");
     exit (3);
   }
   shmid_ds.shm_perm.mode |= S_IWUSR;
   if (shmctl (shm_id, IPC_SET, &shmid_ds) == -1) {
     perror ("SHMCTL-IPC_SET");
     exit (4);
   }
   (void) shmdt (shm_addr);
   if (shmat (shm_id, shm_addr, 0) != shm_addr) {
     perror ("SHMAT-RDWD");
     exit (5);
   }
 }
}

int main (void) {
 char *test_ptr;
 struct sigaction sact;

 /* Установим реакцию на сигнал SIGSEGV */
 (void) sigemptyset (&sact.sa_mask);
 sact.sa_flags = SA_SIGINFO;
 sact.sa_sigaction = sigsegv_sigaction;
 (void) sigaction (SIGSEGV, &sact, (struct sigaction *) NULL);

 /* Убедимся, что разделяемые сегменты инициализируются нулями */
 test_ptr = (char *) (SHM_BASE_ADDR + 3);
 printf ("Результат попытки чтения до записи: %x\n", *test_ptr);

 /* Попробуем записать */
 *test_ptr = 'A';
 printf ("Результат попытки чтения после записи: %x\n", *test_ptr);

 return (shmctl (shm_id, IPC_RMID, NULL));
}
```

Следует обратить внимание на использование флагов округления адреса присоединения разделяемого сегмента (SHM_RND) и присоединения только на чтение (SHM_RDONLY), а также обработчика сигналов, задаваемого полем **sa_sigaction** структуры типа **sigaction** (в сочетании с флагом SA_SIGINFO) и имеющего доступ к расширенной информации о сигнале и его причинах.

3. Разработка программы на языке С

Используя полученные в процессе обучения навыки программирования, теоретический и методический материал, изложенный в разделах 1 — 2 данного учебно-методического пособия, студент должен:

- написать, отладить и запустить программу на языке С, которая выделяет разделяемый сегмент памяти и проводит на нем некоторые операции;
- продемонстрировать преподавателю работу этой программы;
- подготовить и представить на проверку преподавателю письменный отчет о выполнении лабораторной работы, содержащий текст программы, результаты, демонстрирующие исследовательскую часть работы, и выводы о технологии и возможностях использования разделяемых сегментов памяти при разработке прикладных программ.

Методические указания по данной лабораторной работе, рекомендуют студенту последовательное выполнение следующих этапов:

1. Изучение раздела 1 учебно-методического пособия с целью повышения уровня своей общекультурной компетенции при разработке системного и прикладного программного обеспечения; студент должен осознать потребность синхронизации процессов при разработке системного и прикладного ПО.
2. Изучение раздела 2 обеспечивает студента теоретическими знаниями и инструментом для разработки программного обеспечения повышающего быстродействие алгоритмов обрабатывающих общие источники данных.
3. Завершив изучение методического материала разделов 1-2, студент должен запустить ОС УПК АСУ так, как изложено в методическом пособии по лабораторной работе №1.
4. Запустив инструментальную среду разработки EclipseC, студент должен создать новый проект, написать, отладить запустить на выполнение прикладную программу, согласно указанному выше заданию.
5. Результаты выполнения данной программы следует продемонстрировать преподавателю.
6. По результатам исследования студент подготавливает письменный отчет, при необходимости, иллюстрируя его результатами исследования.
7. Подготовив отчет, студент докладывает о выполнении задания преподавателю и следует его указаниям по сдаче выполненной работы.

4. Контроль выполнения лабораторной работы №3

Обязательным требованием по контролю знаний и умений студента является наличие письменного отчета по выполненной лабораторной работе №3.

Отчет оформляется как третий раздел общего отчета по дисциплине «Архитектура вычислительных комплексов» и находится в определенном преподавателем месте учебного комплекса кафедры АСУ.

Отчет по лабораторной работе №3 должен содержать, как минимум три подраздела: постановка задачи, описание работы, выводы.

Цель подготовки и сдачи отчета — формирование у студентов общекультурных компетенций по оформлению и представлению научных и исследовательских документов.

Порядок контроля навыков студента и сдача отчета проводятся в следующей последовательности.

1. Студент сообщает преподавателю о завершении выполнения задания и готовности студента к контролю навыков и сдаче отчета.
2. Преподаватель убеждается в наличии отчета и необходимых элементов его оформления, а затем делает замечания по устранению недостатков отчета или уточняет время и условия контроля навыков и приема результатов работы.
3. В процессе сдачи отчета, студент демонстрирует результаты работы и отвечает на вопросы преподавателя, при необходимости, устраняет ошибки или недоработки, отмеченные преподавателем.
4. Приняв отчет студента, преподаватель сообщает об этом факте устно, при необходимости оценивает результаты работы, а также определяет дальнейший процесс обучения студента.

ЛИТЕРАТУРА

1. Галатенко В.А. Прогаммирование в стандарте POSIX. Интернет ресурс: http://www.intuit.ru/department/se/pposix/8/.

Учебное издание

Резник Виталий Григорьевич

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Методические рекомендации для выполнения лабораторной работы №3 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки 010400.68 «Прикладная математика и информатика» профиля «Математическое и программное обеспечение вычислительных комплексов и компьютерных сетей».

Учебно-методическое пособие

Усл. печ. л. . Тираж ___. Заказ .

Томский государственный университет

систем управления и радиоэлектроники

634050, г. Томск, пр. Ленина, 40