ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение высшего
профессионального образования

«Тихоокеанский государственный университет»

ПРОЦЕССЫ И ПОТОКИ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Методические указания к лабораторной работе
по дисциплине «Операционные системы»
для студентов всех форм обучения по направлениям
«Информатика и вычислительная техника»:
654600 – подготовка дипломированных специалистов;
522800 – подготовка бакалавров

Автор: Сорокин Н.Ю.

Хабаровск
Издательство ТОГУ
2006

УДК 681.3 (075)

Процессы и потоки в операционной системе Linux: методические указания к лабораторной работе по дисциплине «Операционные системы» для студентов всех форм обучения по направлениям «Информатика и вычислительная техника»: 654600 – подготовка дипломированных специалистов; 522800 – подготовка бакалавров / сост. Н.Ю. Сорокин. – Хабаровск: Изд-во ТОГУ, 2006. – 19 с.

Методические указания составлены на кафедре «Вычислительная техника». В них содержатся материалы, необходимые для самостоятельной подготовки студентов к выполнению лабораторной работы. В описание лабораторной работы включены цель работы, необходимое описание стандартных функций для работы с процессами и потоками в ОС Linux, порядок выполнения и правила оформления результатов. Методические указания разработаны в соответствии с рабочей программой и требованиями государственных образовательных стандартов высшего профессионального образования.

Печатается в соответствии с решением кафедры «Вычислительная техника».

Цель работы: изучение принципов создания, работы и завершения процессов и потоков в операционной системе Linux; изучение функций библиотеки Pthreads.

1. Общие сведения

Основной задачей любой многозадачной операционной системы является управление процессами. Операционная система должна разделять между ними ресурсы, предоставлять возможность совместно использовать информацию, обмениваться ею, защищать ресурсы и т.д. В мультизадачной операционной системе несколько различных процессов могут выполняться, чередуясь друг с другом. С этим типом параллелизма связана масса проблем распределения ресурсов, с которыми сталкивается как создающий приложение программист, так и операционная система.

Во многих современных операционных системах положение с управлением процессами усложняется введением понятия потока. В многопоточной системе принадлежность ресурсов остается атрибутом процесса, в то время как сам процесс представляет собою множество параллельно выполняющихся потоков [1].

1.1 Процессы

Процесс в операционной системе Linux представляет собой единицу работы вычислительной системы, которой операционная система выделяет ресурсы. Каждый процесс в системе имеет свой уникальный идентификатор процесса (PID), представляемый целым числом. Существует так же идентификатор родительского процесса (PPID). Процесс может порождать другой процесс.

Порождение нового процесса в операционной системе Linux реализовано копированием записи таблицы процессов, таким образом, что дочерний процесс (процесс-потомок) в момент своего порождения представляет собой точную копию родительского процесса (процесса-предка). Родительский процесс и дочерний процесс далее выполняются параллельно, но родительский процесс может и ожидать завершения дочернего процесса.

Создание процесса

Новый процесс порождается системным вызовом fork(), который создает дочерний процесс - копию родительского. Прототип данной функции находится в <unistd.h>. В дочернем процессе выполняется та же программа, что и в родительском, и когда дочерний процесс начинает выполняться, он выполняется с точки возврата из системного вызова fork(). Системный вызов fork() возвращает родительскому процессу PID дочернего процесса, а дочернему процессу - 0. По коду возврата вызова fork() дочерний процесс может "осознать" себя как дочерний.

Свой PID процесс может получить при помощи системного вызова getpid(), а PID родительского процесса - при помощи системного вызова getppid(). Если требуется, чтобы в дочернем процессе выполнялась программа, отличная от программы родительского процесса, процесс может сменить выполняемую в нем программу при помощи одного из системных вызовов семейства exec. Все вызовы этого семейства загружают для выполнения в процессе программу из заданного в вызове файла и отличаются друг от друга способом передачи параметров этой программе. Таким образом, наиболее распространенный контекст применения системного вызова fork() выглядит примерно так:

```
/* порождение дочернего процесса и сохранение его PID */
if (!(ch_pid=fork()))
/* загрузка другой программы в дочернем процессе */
exec(программа);
else /* продолжение родительского процесса */
```

Пример 1. Создание процесса

```
1:  #include <stdio.h>
2:  #include <unistd.h>
3:
4:  int main(void)
5:  {
6:  int pid;
7:
8:  printf("Hello World!\n");
9:  printf("I am the parent process and my PID is:
10: %d.\n",getpid());
11: printf("Here I am before use of forking\n");
12: pid = fork();
13: printf("Here I am just after forking\n");
14:
15: if (pid == 0)
16: printf("I am the child process and PID is:
17: %d.\n",getpid());
18: else
19: printf("I am the parent process and PID is:
20: %d.\n",getpid());
21:
22: return 0;
23: }
```

Примерный[1] результат работы программы:

```
Hello World!
I am the parent process and my PID is: 23951.
Here I am before use of forking
Here I am just after forking
Here I am just after forking
I am the child process and my PID is: 23952.
I am the parent process and my PID is: 23951.
```

Завершение процесса

Нормальное завершение процесса происходит при достижении конца функции main или при выполнении системного вызова exit(). При этом процесс устанавливает некоторый код своего завершения, который может быть прочитан процессом-предком. Существует общепринятое соглашение, по которому 0 означает нормальное завершение, а любое другое значение – ошибку. Множество стандартных библиотечных вызовов используют ошибки, определенные в <sys/stat.h>.

Принудительное завершение процесса может быть выполнено при помощи системного вызова kill(), посылающего процессу сигнал. Гарантированно принудительно завершить процесс, имеющий PID = p, можно системным вызовом kill(p,SIGKILL).

Процесс-предок может ожидать завершения процесса-потомка (или процессов-потомков) при помощи системных вызовов wait() или waitpid(). Прототипы этих функций находятся в <sys/wait.h>. Если процесс-потомок еще не завершился, процесс-предок переводится таким системным вызовом в состояние ожидания до завершения процесса-потомка (процесс может и не ожидать завершения потомка, а только проверить, завершился ли он). Эти системные вызовы позволяют также процессу предку узнать код завершения потомка [4].

Пример 2. Ожидание завершения процесса

```
1:   #include <stdio.h>
2:   #include <sys/wait.h>
3:   int main(void)
4:   {
5:   int pid;
6:   int status;
```

[1] При выполнении данных программ на различных по типу и по скорости процессора ЭВМ, а также при использовании различных версий ядра операционной системы Linux результаты выполнения могут различаться с приведенными в данных методических указаниях

```
7:
8:  printf("Hello World!\n");
9:  pid = fork( );
10:
11: if (pid == -1)
12: {
13: perror("bad fork");
14: exit(1);
15: }
16:
17: if (pid == 0)
18: printf("\t I am the child process.\n");
19: else
20: {
21: wait(&status); /* parent waits for child to finish */
22: printf("I am the parent process.\n");
23: }
24:
25: exit(0);
26: }
```

Примерный результат работы программы:

```
Hello World!
I am the child process.
I am the parent process.
```

1.2 Потоки

В операционных системах концепция процесса может быть охарактеризована двумя параметрами [1]:

- владение ресурсами,
- планирование/выполнение.

В большинстве устаревших операционных систем эти две характеристики являются сущностью процесса. Однако, современные операционные системы позволяют различать две приведенные выше характеристики: единицу диспетчеризации обычно называют *потоком*, а единицу владения ресурсами – *процессом*.

Главная цель использования потоков – реализовать выгоды от потенциальной производительности программы. Если сравнить стоимости создания и управления процесса и потока, то создание потока требует гораздо меньших накладных расходов. Все потоки внутри процесса разделяют одно и то же адресное простран-

ство. Межпотоковое сообщение более эффективно в большинстве случаев и проще, чем межпроцессное сообщение.

Потоковые приложения позволяют получить прирост производительности в следующих случаях:

- Одновременное выполнение вычислительной работы и с устройствами ввода-вывода. Например, программа может иметь участок, где она выполняет длительную операцию ввода-вывода. В то время как один поток ждет завершения операции ввода-вывода, остальные потоки могут интенсивно использовать процессор.
- Асинхронная обработка событий. Задачи, которые обрабатывают события с неизвестной частотой и длительностью, могут выполняться параллельно. Например, ftp-сервер может в одном потоке передавать данные по запросу, а в другом – ожидать новые запросы.

Многопоточные приложения работают как на однопроцессорных системах, так и на многопроцессорных без перекомпиляции, используя во втором случае все преимущества многопроцессорной обработки.

Поток – это легковесный процесс, который имеет свой собственный стек и выполняет заданную часть кода. В отличие от обычных процессов потоки обычно разделяют свою память с другими потоками. В составе процесса может быть запущено несколько потоков, которые выполняются параллельно (или квазипараллельно - в режиме разделения времени процессора). Можно считать (и в некоторых операционных системах это действительно так), что в любом процессе имеется по крайней мере один поток – главный. Это тот поток, в котором выполняется функция main. Главный поток может порождать другие потоки. В программе процесса поток имеет вид процедуры/функции, которая вызывается специальным системным вызовом и после вызова выполняется параллельно с запустившим ее потоком.

В Таблице 1 приведена информация о наследовании атрибутов при создании процессов (используя функцию fork()) и потоков (используя функцию pthread_create(…)) в операционной системе Linux. Полная версия таблицы приведена в книге [4].

Таблица 1. Наследование атрибутов процессов и потоков

Атрибут	**fork()**	**pthread_create(…)**
Виртуальная память		
Сегмент кода	копия	общий
Пост. сегмент данных	не исп.	общий
Изменяемый сегмент денных	копия	общий
Стек	копия	раздельный

Командная строка	копия	общая
Переменные окружения	копия	общие
Файлы		
Файловые дескрипторы	копия	общие
Блокировки файлов	раздельные	раздельные
Сигналы		
Обработчики сигналов	копия	общие
Маски сигналов	раздельные	раздельные
Квант времени	раздельный	общий

Функции управления потоками

В стандарте Posix функции, предназначенные для работы с потоками, и функции синхронизации потоков объединены в одну библиотеку Pthreads. В Таблице 2 перечислен базовый набор функций работы с потоками.

Таблица 2. Функции для работы с потоками библиотеки Pthreads

Функция	`int` **`pthread_create`**`( pthread_t * threadhandle, pthread_attr_t *attribute, void *(*start_routine)(void *), void *arg );`
Описание	Запрос к библиотеке на создание нового потока. Возвращаемое значение равно нулю в случае успеха. В случае неуспешного выполнения возвращаемое значение отрицательное. phread_t – это абстрактный тип данных для идентификации потока.
Функция	`void` **`pthread_exit`** `(void *retval);`
Описание	Эта функция используется для инициации завершения потока. Возвращаемое значение передается в виде указателя. Значение указателя может быть любым. Как вариант можно передавать адрес структуры данных, если возвращаемое значение очень большое.
Функция	`int` **`pthread_join`**`(pthread_t threadhandle, void *returnvalue);`
Описание	Ожидание завершения другого потока, заданного параметром threadhandle. Возвращает ноль в случае успешного завершения или отрицательное в противном случае. Возвращаемое значение – указатель, возвращенный по ссылке. Если не нужны возвращаемые значения, то можно передать NULL в качестве второго аргумента.
Функция	`int` **`pthread_cancel`**`(pthread_t thread);`
Описание	Функция pthread_cancel посылает запрос завершения потоку thread. Если такого потока нет, то pthread_cancel завершается с

	ошибкой. В противном случае она возвращает ноль. Запрос завершения информирует поток thread, чтобы он завершился как можно быстрее. Однако выдача этого запроса вовсе не гарантирует того, что поток получит или обработает этот запрос. Например, если запрос завершения появляется во время очень важной операции, поток продолжит свою работу (в случае, если то, что делает поток не может быть прервано в той точке, где происходит запрос). Программист может передать статус завершения, хранимый как указатель на void, для любого потока, который может присоединить данный поток.
Функция	`pthread_t` **`pthread_self`**`(void);`
Описание	Возвращает уникальный идентификатор вызывающего потока ThreadID.

Для включения поддержки потоков в программе необходимо:

1. Включить заголовочный файл <pthread.h>.
2. Объявить переменную типа pthread_t: pthread_t the_thread, которую необходимо использовать при вызове функции pthread_create.
3. При компиляции использовать опцию компоновщика pthread:

```
gcc threads.c –o threads –lpthread
```

Сначала потоки создаются изнутри процесса. После того, как созданы эти потоки, они становятся равноправными и могут создавать другие потоки. Начальный поток в процессе – это поток, выполняющий функцию main.

Существует несколько способов завершения потоков в ОС Linux:

- Поток возвращается из своей функции (главной функции для потока). По умолчанию библиотека Pthreads забирает все ресурсы, использованные этим потоком; это подобно завершению процесса, когда управление достигает конца функции main.
- Поток вызывает функцию pthread_exit.
- Поток завершается через запрос завершения из другого потока, использующего для этого функцию pthread_cancel.
- Поток получает сигнал, который его завершает.
- Весь процесс завершает свое выполнение по системному вызову exec или exit.

После завершения основного процесса, все порожденные им потоки завершаются системой, если только эти потоки не были созданы с атрибутом PTHREAD_CREATE_DETACHED. По умолчанию, все потоки создаются с атри-

бутом PTHREAD_CREATE_JOINABLE - это означает, что данный поток доступен другим потокам для ожидания завершения.

Примеры использования библиотеки потоков

Пример 3. Создание и уничтожение потока

В данном примере показывается создание потока, используя функции библиотеки Pthread.

```
1:  #include <stdio.h>
2:  #include <pthread.h>
3:
4:  void *kidfunc(void *p)
5:  { printf ("Kid ID is ---> %d\n", getpid( )); }
6:
7:  main ( )
8:  {
9:  pthread_t kid;
10:
11: pthread_create (&kid, NULL, kidfunc, NULL);
12: printf ("Parent ID is ---> %d\n", getpid( ));
13: pthread_join (kid, NULL);
14: printf ("No more kid!\n");
15: }
```

Примерный результат работы программы:

```
Parent ID is ---> 2577
Kid ID is ---> 2577
No more kid!
```

Пример 4. Разделение глобальных данных между потоками

Пример демонстрирует возможность использования (чтение, модификация) глобальных переменных в многопоточных приложениях.

```
1:  #include <stdio.h>
2:  #include <pthread.h>
3:  #include <unistd.h>
4:
```

```
5:  int glob_data = 25;
6:
7:  void *kidfunc(void *p)
8:  {
9:  printf ("Kid here. Global data was %d.\n", glob_data);
10: glob_data = 12;
11: printf ("Kid Again. Global data was now %d.\n",
12: glob_data);
13: }
14:
15: int main ( )
16: {
17: pthread_t kid;
18:
19: pthread_create (&kid, NULL, kidfunc, NULL);
20: printf ("Parent here. Global data = %d.\n", glob_data);
21:
22: glob_data = 33;
23: pthread_join (kid, NULL);
24: printf ("End of program. Global data = %d.\n",
25: glob_data);
26:
27: exit(0);
28: }
```

Примерный результат работы программы:

```
Kid here. Global data was 25.
Kid Again. Global data was now 12.
Parent here. Global data = 12.
End of program. Global data = 33.
```

Пример 5. Разница между процессом и потоком

Данный пример является демонстрацией работы порожденных процессов и потоков с глобальными и локальными переменными, используя различные адресные пространства.

```
1:  #include <stdio.h>
2:  #include <pthread.h>
3:  #include <unistd.h>
```

```
4:
5:  int this_is_global;
6:
7:  void thread_func(void *dummy)
8:  {
9:  int local_thread;
10: printf("Thread %d, pid %d, addresses: &global: 0x%X,
11: &local: 0x%X\n", pthread_self(), getpid(),
12: &this_is_global, &local_thread);
13: this_is_global++;
14: printf("In Thread %d, incremented this_is_global=%d\n",
15: pthread_self(), this_is_global);
16: pthread_exit(0);
17: }
18:
19: int main( )
20: {
21: int local_main;
22: int pid, status;
23: pthread_t thread1, thread2;
24:
25: printf("First, we create two threads...\n");
26:
27: this_is_global=1000;
28: printf("Set this_is_global=%d\n",this_is_global);
29: pthread_create(&thread1, NULL, (void*)&thread_func,
30: NULL);
31: pthread_create(&thread2, NULL, (void*)&thread_func,
32: NULL);
33: pthread_join(thread1, NULL);
34: pthread_join(thread2, NULL);
35: printf("After threads, this_is_global=%d\n\n",
36: this_is_global);
37:
38: printf("Now that the threads are done, let's call
39: fork...\n");
40: local_main=17; this_is_global=17;
41: printf("Before fork(), local_main=%d,
42: this_is_global=%d\n",local_main, this_is_global);
43:
44: pid=fork();
```

```
45:
46: if (pid == 0)
47: {
48: printf("In child, pid %d: &global: 0x%X, &local:
49: 0x%X\n", getpid(), &this_is_global, &local_main);
50: local_main=13; this_is_global=23;

51: printf("Child set local main=%d, this_is_global=%d\n",
52: local_main, this_is_global);
53: exit(0);
54: }
55: else
56: {
57: printf("In parent, pid %d: &global: 0x%X, &local:
58: 0x%X\n", getpid(), &this_is_global, &local_main);
59: wait(&status);
60:
61: printf("In parent, local_main=%d, this_is_global=%d\n",
62: local_main, this_is_global);
63: }
64: exit(0);
65: }
```

Примерный результат работы программы:

```
First, we create two threads...
Set this_is_global=1000
Thread 16386, pid 2900, addresses: &global: 0x8049C80,
&local: 0x4099FAD0
In Thread 16386, incremented this_is_global=1001
Thread 32771, pid 2900, addresses: &global: 0x8049C80,
&local: 0x4119FAD0
In Thread 32771, incremented this_is_global=1002
After threads, this_is_global=1002

Now that the threads are done, let's call fork...
Before fork(), local_main=17, this_is_global=17
In parent, pid 2898: &global: 0x8049C80, &local: 0xBFFFF634
In child, pid 2901: &global: 0x8049C80, &local: 0xBFFFF634
Child set local main=13, this_is_global=23
In parent, local_main=17, this_is_global=17
```

Пример 6. Использование глобальной переменной по записи

```
1:  #include <stdio.h>
2:  #include <pthread.h>
3:  #include <stdlib.h>
4:  #include <unistd.h>
5:
6:  int tot_items = 0;
7:
8:  struct kidrec {
9:  int data ;
10: pthread_t id ; };
11:
12: #define NKIDS 50
13:
14: void *kidfunc(void *p)
15: {
16: int *ip = (int *)p;
17: int tmp, n;
18:
19: tmp = tot_items;
20: for (n = 0;n<50000;n++) tot_items = tmp + *ip;
21: }
22:
23: int main ( )
24: {
25: struct kidrec kids[NKIDS];
26: int m;
27:
28: for (m=0; m<NKIDS; ++m)
29: {
30: kids[m].data = m+1;
31: pthread_create (&kids[m].id, NULL, kidfunc,
32: &kids[m].data);
33: }
34:
35: for (m=0; m<NKIDS; ++m) pthread_join(kids[m].id, NULL);
36: printf ("End of Program. Grand Total = %d\n",
37: tot_items);
38:
```

```
39:  exit(0);
40:  }
```

Примерный результат работы программы:

```
End of Program. Grand Total = 1275
```

2. Задания на лабораторную работу

Задания на лабораторную работу состоят из двух частей, и выполнение данных заданий рассчитано на несколько часов лабораторных работ. Первая часть заданий рассчитана на выполнение в течение двух часов, вторая часть в течение четырех часов.

Часть 1. Процессы

1.1. Изучите команды **ps** и **kill**. Запустите любую свою программу, используя в конце & (например: ./a.out &). Дайте письменно объяснение результата работы команды **ps**. Опишите также, как завершить процесс a.out ?

1.2. Изучите команды ОС Linux, предназначенные для управления процессами и мониторинга состояния операционной системы: **jobs**, **fg**, **bg**, **top**. Объясните письменно назначение и использование каждой из команд, а также их параметров. Подробно опишите действия, необходимые для переключения процесса в фоновый режим выполнения и возврата из этого режима.

1.3. Изучите функции семейства exec, а именно execv(), execl(), execvp()и execlp(). Объясните письменно разницу между этими функциями.

1.4. Напишите программу на языке C/C++, удовлетворяющую всем перечисленным условиям:

1) Программа должна создать четыре процесса и ожидать окончания их выполнения.
2) Порядок создания процессов задается с командной строки (передается в виде параметров в функцию main), например: ./a.out 1 3 2 4
3) Каждый процесс и основная программа должны выводить на экран сообщения о начале и завершении своей работы в следующем формате: PID PPID “сообщение” “время сообщения”
4) Для получения текущего времени используйте функцию ctime() из <**time.h**>.

5) Процессы должны использовать функции, перечисленные по порядку: execv(), execl(), execvp()и execlp() с любыми командами ОС Linux внутри. Вы можете также использовать любые свои программы.
6) Программа должна анализировать и сообщать о причинах завершения процессов, используя переменную errno. Все значения ошибок определены в заголовочном файле <**sys/errno.h**>

Часть 2. Потоки

2.1 Запустите несколько раз программу, приведенную в примере 6. Объясните письменно:
1) Результат работы программы.
2) Почему при запуске программы несколько раз и при увеличении числа потоков (NKIDS) получаются различные результаты?

2.2 Напишите программу на языке C/C++, в которой:
1) Запускается четное количество потоков параллельно.
2) Каждый нечетный поток (например, первый) создает файл с именем из своего PID, записывает в него произвольное число символов (от нескольких символов до сотен миллионов символов) и закрывает этот файл.
3) Каждый четный поток (например, второй) открывает файл, созданный предыдущим потоком с нечетным номером (в нашем случае первым), читает его, считает количество символов в файле и закрывает его; при этом четный поток не должен иметь никакой информации о количестве записываемых в файл символов и о том, закончена ли запись в файл нечетным потоком.
4) Каждый из потоков выводит следующую информацию: ThreadID, PID, PPID, время, имя файла, количество записанных или считанных символов.
5) Количество пар создаваемых потоков передается аргументом в программу с командной строки. Программа должна ждать завершения работы всех потоков, анализировать и сообщать о причинах завершения потоков. Для передачи имен файлом между потоками можно использовать символьный массив в основной программе.

2.3 Ответьте письменно на вопрос: какие атрибуты потоков реализованы в библиотеке Pthreads (см. <pthread.h> и файлы справки).

3. Порядок выполнения работы

1. Изучить основы работы с процессами и потоками в операционной системе Linux.
2. Подготовить отчет по лабораторной работе, состоящий из двух частей – заданий для процессов и для потоков. В отчете дать ответы на вопросы заданий.
3. Написать программу по заданию части 1. Продемонстрировать работу программы. Объяснить работу программы.
4. Написать программу по заданию части 2. Продемонстрировать работу программы. Объяснить работу программы.
5. Ответить на вопросы преподавателя по тематике лабораторной работы.

4. Контрольные вопросы

1. Что такое процесс?
2. Работа с процессами в операционной системе Linux: команды kill, ps, top.
3. Функция ожидания завершения работы процесса, возвращаемые параметры.
4. Что такое поток?
5. Основные функции работы с потоками библиотеки Pthreads.
6. Примеры использования функций создания, завершения потока.
7. Адресное пространство процесса и потока.
8. Передача параметров при создании потока.
9. Что такое атрибуты потока?
10. Что такое атрибуты планирования потока?
11. Что такое отсоединенные потоки?

5. Содержание отчета

1. Титульный лист.
2. Цель и задачи работы.
3. Ответы на вопросы.
4. Описание программ.
5. Текст программ.
6. Результаты работы программ.
7. Выводы.
8. Список литературы.

Литература

1. В. Столлингс Операционные системы - М: Вильямс, 2004. – 845 с.
2. М.Дансмур Г.Дейвис Операционная система UNIX и программирование на языке Си - М.: Радио и связь, 1989. – 254 с.
3. Г.Р. Эндрюс Основы многопоточного, параллельного и распределенного программирования - М: Вильямс, 2003. – 505 с.
4. K. Wall, M. Watson, M. Whitis Linux programming unreleased – SAMS, 1999. – 818 с.
5. Файлы справки операционной системы Linux.

ПРОЦЕССЫ И ПОТОКИ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Методические указания к лабораторной работе
по дисциплине «Операционные системы»
для студентов всех форм обучения по направлениям
«Информатика и вычислительная техника»:
654600 – подготовка дипломированных специалистов;
522800 – подготовка бакалавров

Сорокин Николай Юрьевич

Главный редактор Л.А. Суевалова

Редактор Е.Н. Ярулина

Компьютерная верстка Н.Ю. Сорокин

Лицензия на издательскую деятельность ЛР №020526 от 23.04.97

Подписано в печать _____. Формат 60x84 1/16

Бумага писчая. Офсетная печать. Усл. печ. л. ____

Уч.-изд. Л. ____. Тираж 50 экз. Заказ ____. С ____

Издательство Тихоокеанского государственного университета.

680035, Хабаровск, ул. Тихоокеанская , 136

Отдел оперативной полиграфии издательства

Тихоокеанского государственного университета.

680035, Хабаровск, ул. Тихоокеанская, 136