UNIX

Лекция 4

ПРОЦЕССЫ ОС UNIX

- ***Процесс** - это задание в ходе его выполнения*. П - образ программы, включающий отображение в памяти исполняемого файла, полученного в ходе компиляции, сегментов стека, кода и данных, библиотек, а также ряд структур данных ядра, необходимых для управления процессом.
- **Выполнение** процесса заключается в точном следовании набору инструкций, который является замкнутым в том смысле, что он не передает управление набору инструкций другого процесса. Он считывает и записывает информацию в раздел данных и в стек, но ему недоступны данные и стеки других процессов.

- Программа - исполняемый файл, т.е. это нечто *потенциально* активное (множество файлов, необходимых для выполнения какой-либо задачи)
- Процесс – это последовательность операций программы или часть программы при ее *выполнении*.

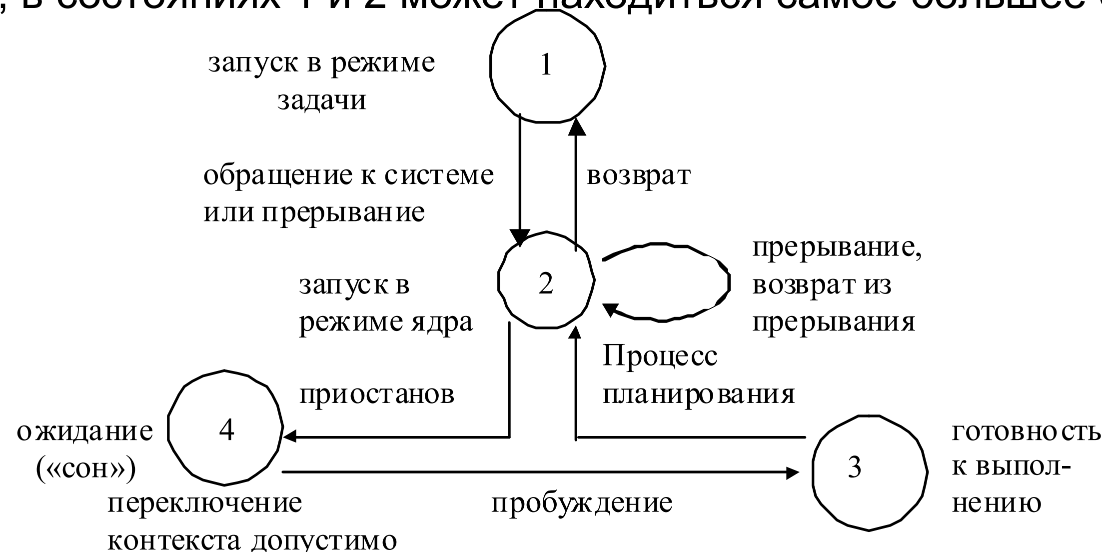
- UNIX - многозадачная система $\Rightarrow$ в ней параллельно выполняется множество процессов, их выполнение планируется ядром.
- Несколько процессов могут быть экземплярами одной программы.
- Процессы взаимодействуют с другими процессами и с вычислительными ресурсами только посредством обращений к ОС, которая распределяет системные ресурсы между активными процессами.

Состояния процесса

Время жизни процесса можно разделить на несколько состояний:

1. Процесс выполняется в режиме **задачи («обычные» инструкции)**.
2. Процесс выполняется в режиме **ядра (системные вызовы)**.
3. Процесс не выполняется, но готов к выполнению, находится в очереди готовых к исполнению процессов и ждет, когда планировщик выберет его. В этом состоянии может находиться много процессов, и алгоритм планирования устанавливает, какой из процессов будет выполняться следующим.
4. Процесс приостановлен («спит»). Процесс «впадает в сон», когда он не может больше продолжать выполнение (ждет завершения ввода-вывода или освобождения какого-либо занятого ресурса)

- Поскольку процессор в каждый момент времени выполняет только один процесс, в состояниях 1 и 2 может находиться самое большее один процесс.



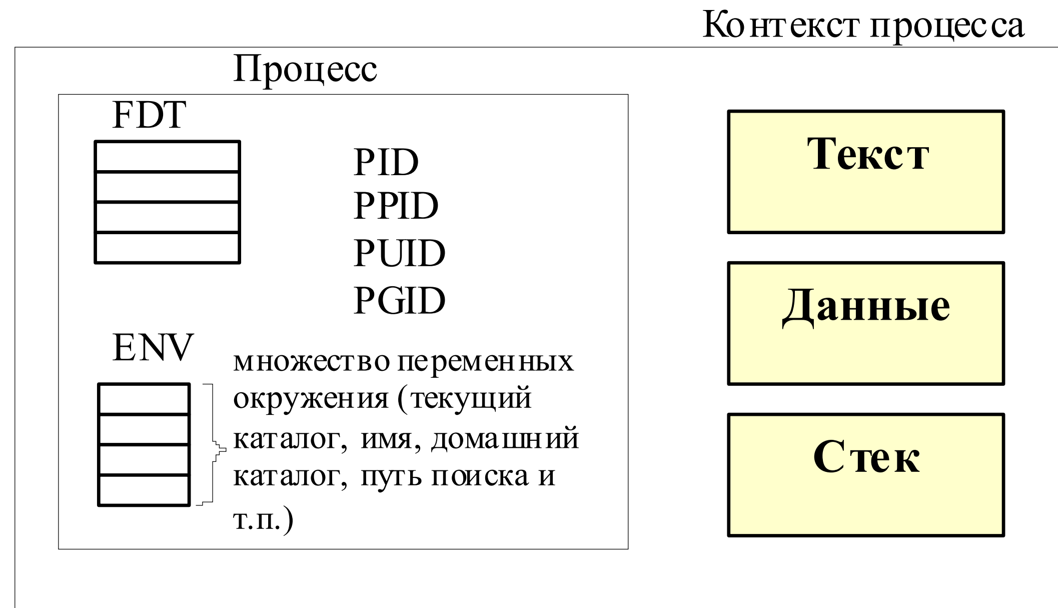
Контекст процесса

Контекст процесса - вся информация, необходимая для описания процесса:

- адресное пространство процесса в режиме задачи (код, данные и стек процесса, а также разделяемая память и данные динамических библиотек);
- управляющая информация (структуры *proc* и *user* - запись таблицы процессов и дополнительная информация соответственно);
- окружение процесса (системные переменные, например, домашний каталог, имя пользователя и др.);
- аппаратный контекст (значения используемых машинных регистров).

Прерывания

- Ядро обрабатывает прерывания в контексте прерванного процесса.
- Прерванный процесс мог при этом выполняться как в режиме задачи, так и в режиме ядра.
- Ядро сохраняет информацию, достаточную для того, чтобы можно было позже возобновить выполнение прерванного процесса, и обрабатывает прерывание в режиме ядра.
- Ядро не порождает и не планирует порождение какого-то особого процесса по обработке прерываний.



Сегменты

Сегмент текста - область памяти, в которой находятся коды, подлежащие выполнению (вообще говоря, сегмент – это область памяти, которой система управляет как единым целым);.

Сегмент данных - содержит глобальные и *статические* переменные

Сегмент стека - содержит динамический стек. Он обеспечивает вызов функций, и передачу параметров. В стеке хранятся аргументы функций, локальные переменные и адреса возврата всех функций, активных в процессе в каждый данный момент времени.

Статическая переменная – это переменная, объявленная внутри какой-то функции, а расположенная в глобальной области данных (в противном случае она располагается в стеке).

<u>*Например*</u>:

```
int a;
void f()
{     int i;
      static int s=10;
      ………
}
………
```

Здесь:

a – глобальная переменная;

i – локальная переменная;

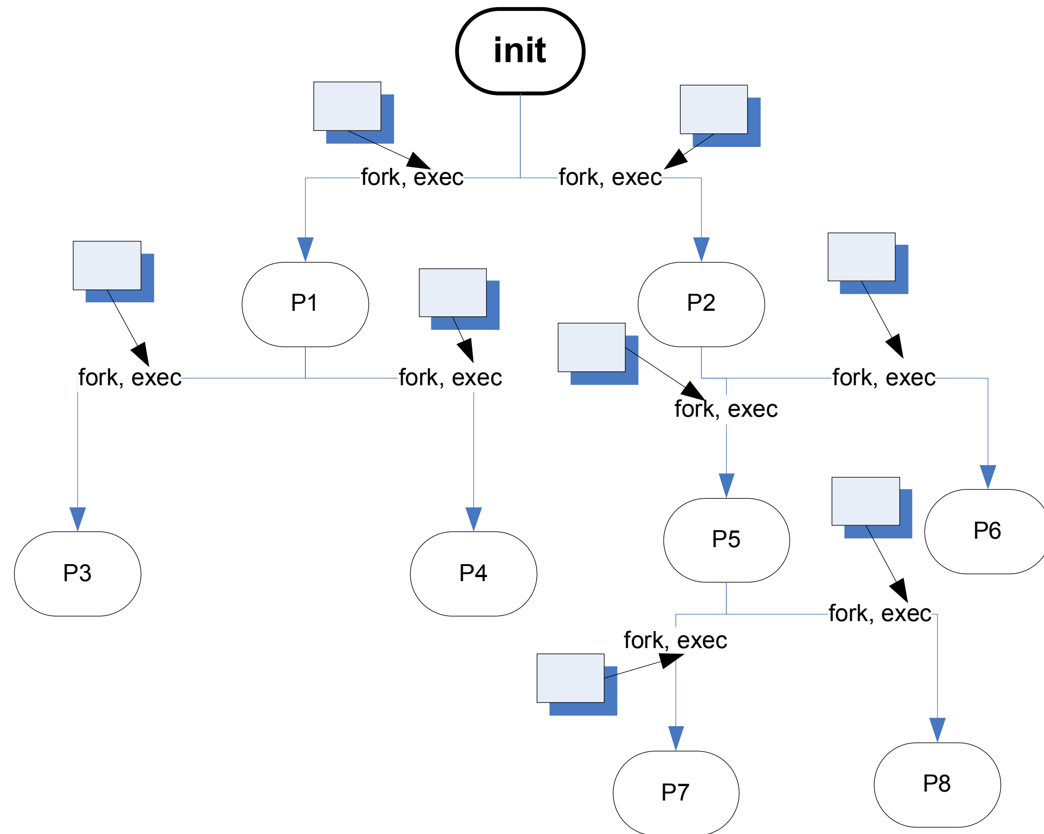
s – статическая переменная.

Статическая переменная инициализируется один раз. Она обладает всеми свойствами глобальной переменной, за исключением прав доступа.

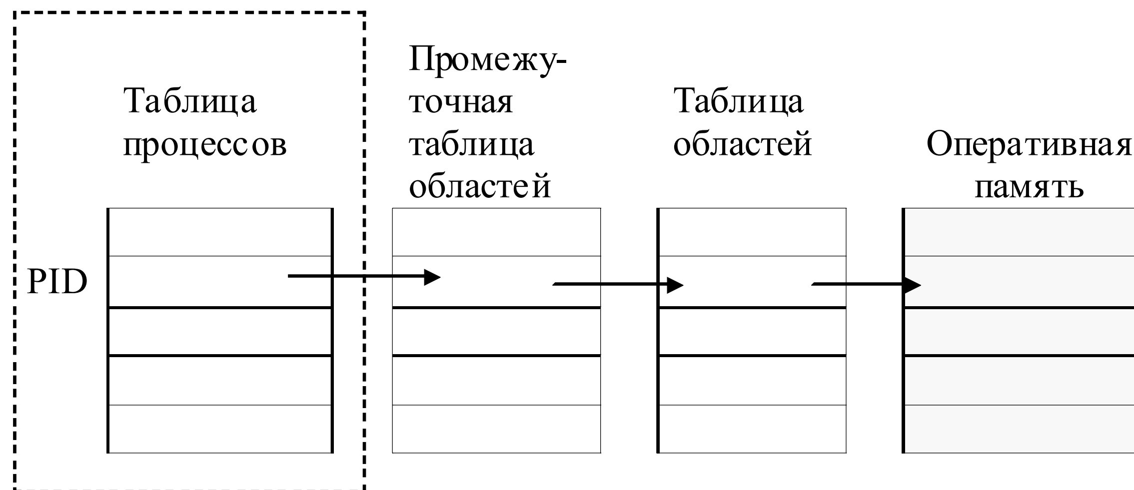
Выполнение процесса

Процессы-родители и процессы-потомки

- Процесс № 0
- Процесс № 1 ***init***
- ***fork()***
- ***exec()***



Идентификатор процесса и таблица процессов



- ТП. Принадлежит ядру. PID, PUID и т.п., а также ссылки на ПТО.
- ПТО. Информация о типе используемой памяти, является ли она общедоступной или приватной.
- ТО. Описание доступа к физической памяти

Системные вызовы для работы с процессами

fork. Создание нового процесса:

int fork(void)

Создается порожденный процесс, и функция возвращает идентификатор этого порожденного процесса родительскому процессу. Порожденный процесс получает от *fork* нулевой код возврата.

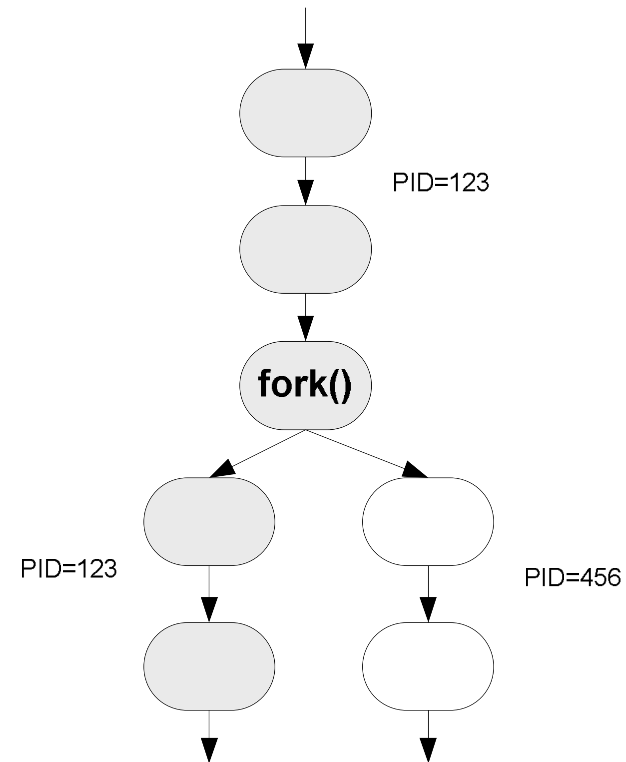
Действия ядра

1. Отводит место в таблице процессов под новый процесс.
2. Присваивает порождаемому процессу уникальный PID.
3. Делает копию контекста родительского процесса (или вместо копирования области в новый физический участок памяти увеличивает значение счетчика ссылок на область).
4. Увеличивает значения счетчика числа файлов, связанных с процессом, как в таблице файлов, так и в таблице индексов.
5. Возвращает родительскому процессу код идентификации порожденного процесса, а порожденному процессу - 0.

fork()

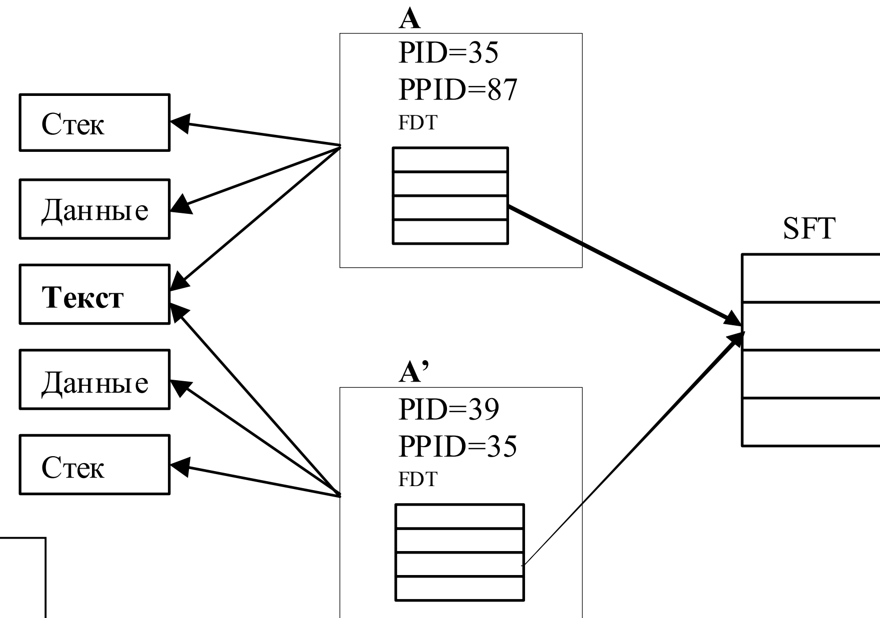
```
main()
{   write(1, "I am a parent",...); → ⊗
    fork();
    write(1, "...", ...);
}
```

```
⊗→      n=fork();
if (n==0)
        write(1, "I'm a child",...)
else
        write(1, "I'm a parent",...);
}
```



Примеры

```
main()
{       printf (“Start %d”, getpid());
        /*выводим информацию о PID*/
        n = fork();
        if (n == 0)
        {   printf(“CHILD %d”, getpid());
            exit(0);
        }
        else
        {  printf(“I AM PARENT %d”, getppid());
           printf(“MY CHILD IS %d”, n);
        }
}
```




```
f = open("file", O_RDONLY);
switch(fork())
{       case 0:  // Потомок
                read(f, &c, 1);
                write(1, &c, 1);
                break;
        case –1: // Ошибка
                perror("fork"); exit(1);
        default: // Родитель
                read(f, &c, 1);
                write(1, &c, 1);
}
```

fork увеличивает количество ссылок в SFT. Родитель и потомок ссылаются на одни и те же файлы в SFT.

exit. Завершение выполнения процесса

void exit(int status);

где status - значение, возвращаемое функцией родительскому процессу.

- Процессы могут вызывать функцию *exit* как в явном, так и в неявном виде (по окончании выполнения программы функция *exit* вызывается автоматически с параметром 0).
- Ядро может вызывать функцию exit по своей инициативе, если процесс не принял посланный ему сигнал. В этом случае значение параметра status равно номеру сигнала.
- Выполнение вызова *exit* приводит к «прекращению существования» процесса, освобождению ресурсов и ликвидации контекста.
- Функция *exit,* завершая выполнение процесса, *не освобождает* запись в таблице процессов.
- Запись в таблице процессов освобождается только при вызове функции ***waitpid (wait)****.*

wait. Ожидание завершения выполнения процесса-потомка

*int wait(int *stat)*

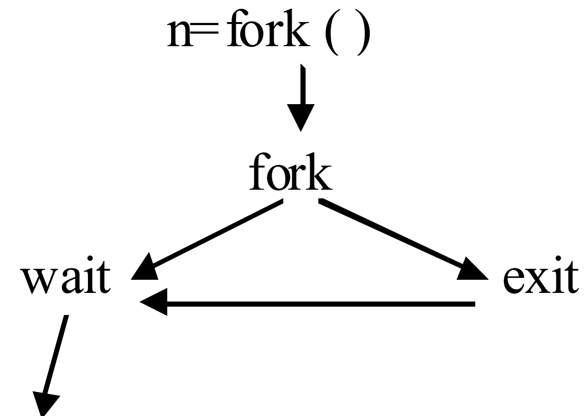
pid = wait(stat);

где *pid* – PID -завершившегося потомка, stat – адрес, по которму будет помещено возвращаемое функцией *exit* значение.

- *wait* приостанавливает выполнение родительского процесса, пока не завершится выполнение какого-нибудь его потомка.
- *wait* синхронизирует продолжение своего выполнения с моментом завершения потомка. Ядро ведет поиск потомков процесса, прекративших существование, и в случае их отсутствия возвращает ошибку.
- Если потомок, прекративший существование, обнаружен, то ядро передает его PID и значение, возвращаемое через параметр функции *exit*, процессу, вызвавшему функцию *wait*.
- Таким образом, через параметр функции *exit* (status) завершающийся процесс может передавать различные значения, содержащие информацию о причине завершения процесса.

Пример

```
n = fork (0);
if (n == 0)
{
………/*потомок*/
    exit (0);
}
wait (0);
printf (“Я дождался потомка %d”,n);
```



waitpid

Ожидание завершения выполнения определенного процесса-потомка

*int waitpid(int pid, int *stat, int flag)*

- flag - режим блокировки (обычно 0). Если flag ==1, то в случае отсутствия процесса, указанного pid, функция немедленно вернет управление вызывающей программе (проверка существования потомка_.

waitpid приостанавливает выполнение родительского процесса, пока не завершится выполнение конкретного потомка:

- Если pid = -1, то ждется завершение любого порожденного процесса;
- Если pid = 0, то ждется завершение любого порожденного процесса, принадлежащего той же группе, что и родитель;
- Если pid < 0 и при этом pid ≠ -1, то ждется завершение любого порожденного процесса, идентификатор группы которого равен модулю pid.

sleep

Приостанов работы процесса на определенное время:

void sleep(unsigned seconds)

- Сначала ядро повышает приоритет работы процесса так, чтобы заблокировать все прерывания, которые могли бы помешать работе с очередями приостановленных процессов, и запоминает старый приоритет, чтобы восстановить его, когда выполнение процесса будет возобновлено.
- Процесс получает пометку «приостановленного», адрес приостанова и приоритет запоминаются в таблице процессов, а процесс помещается в хеш-очередь приостановленных процессов.
- Параметр *seconds* устанавливает лишь **минимальный** интервал, в течение которого процесс будет приостановлен.

exec. Запуск программы

Системный вызов exec осуществляет несколько библиотечных функций - execl, execv, execle и др. :

*Например: int execv(char *path, char *argv[])*

где *path* - имя исполняемого файла, *argv* - указатель на массив параметров, которые передаются вызываемой программе. Аналогичен параметру *argv* командной строки функции *main*. *argv* должен содержать минимум два параметра: первый - имя программы, подлежащей выполнению (отображается в *argv[0]* функции *main* новой программы), второй - NULL (завершающий список аргументов).

- Содержимое пользовательского контекста после вызова функции становится недоступным, за исключением передаваемых функции параметров, которые переписываются ядром из старого адресного пространства в новое.
- *exec* возвращает 0 при успешном завершении и -1 при аварийном (тогда управление возвращается в вызывающую программу).

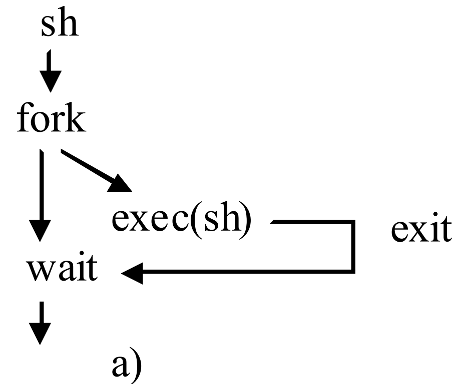
Примеры

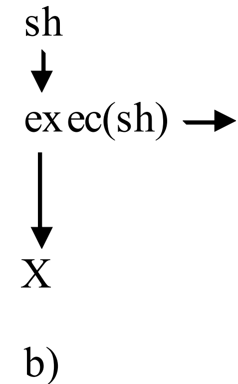
Пример. Работа командного интерпретатора shell:

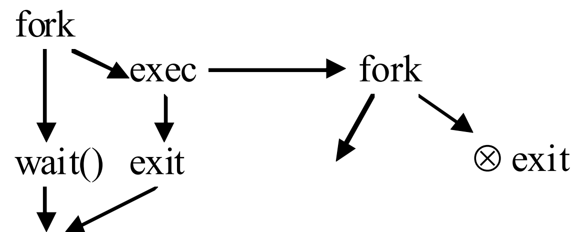
при выполнении команды он сначала порождает свою копию с помощью вызова *fork*, а затем запускает соответствующую указанной команде программу системным вызовом *exec*.

execl ("/bin/sh", "sh", "-c", "ls",0)

```
switch (n=fork())
{     case 0: //child.
              execl("/bin/sh... ",0);
              perror("…");
              break;
      case –1: perror("…");
}
/*можем завершить*/
pp=wait(&status);
      // или
pp=waitpid(n, &status, 0);
. . .
```







system

*int system(const char *cmd)*

Реализует тройку системных вызовов fork/exec/wait

```
int my_system(const char *cmd)
{	pid_t pid;
	int status;
	switch(pid=fork())
	{		case –1:	return 1;
			case 0:		execl("/bin/sh","sh","-c",cmd,0);
					perror("execl");
					exit(errno);
	}
	if(waitpid(pid,&status,0)==pid && WIFEXTED(status))  return WEXITSTATUS(status);
	return –1;
}
int main()
{	int rc=0;
	char buf[256];
	do
	{		printf("sh>");
			fflush(stdout);
			if(!gets(buf)) break;
			rc = my_system(buf);
	} while(!rc);
	return rc;
}
```