MINIX 3.X: Локальный связующий сервис

Pablo Andrés Pessolani
Departamento de Sistemas de InformaciÛn
Facultad Regional Santa Fe - Universidad TecnolÛgica Nacional
Santa Fe - Argentina

Аннотация

MINIX 3.X является операционной системой с открытым кодом, спроектированной на максимальную надёжность, гибкость и защищённость. Ядро здесь малое, и пользовательские процессы, специализированные серверы и драйверы устройств выполняются как пользовательские процессы, в своих изолированных адресных пространствах. MINIX использует передачу сообщений как коммуникационные примитивы между пользовательскими процессами, серверами и драйверами устройств.

Системные вызовы используют `sendrec()` как примитив межпроцессного взаимодействия (InterProcess Communication — IPC) для того, чтобы послать сообщение с запросом к серверу процессов (Process Manager Server - PM) или серверу файловой системы (File System Server - FS), а затем пославший процесс ожидает ответов. Сообщения запросов адресуются на процессы получатели через фиксированные номера концевых точек (endpoint number) для каждого сервера.

MINIX 3.X мог бы использовать локальный связующий сервис (Local Bind Service) для получения номера концевой точки желаемого сервиса, допускающий, что пользовательские процессы будут обслуживаться процессами, отличными от FS или MM, без внесения изменений в код программ. Такие программы должны быть просто перекомпилированы с библиотекой, которая прозрачно разрешает номера конечных точек сервисов через связующий сервер (Bindery Server - BS). Как только связующий сервер допускает, что различные сервера станут регистрировать одни и те же сервисы, но с отличающимися номерами версий, различные процессы смогли бы использовать одни и те же сервися, поддерживаемые различными серверами. Статья описывает проблему, приближения решения и представляет изменения программного кода ядра как подтверждение концепции.

1. Введение

MINIX [1] это полнообъёмная, многозадачная операционная система с разделением времени, спроектированная в основах Эндрю Таненбаумом. Главной целью проекта есть его широкое использование в компьютерных университетских курсах. Следуя её лицензионным условиям, исходный код системы становится широко доступным для университетских курсов и исследований. Её главными отличительными чертами являются:

- Базирование на микроядре: обеспечивает управления процесспми и диспетчеризация, базовое управление памятью, межпроцессные взаимодействия, обслуживание прерываний и операции ввода-вывода (I/O) низкого уровня;
- Многоуровневость системы: допускает модульное проектирование и ясность реализации новых возможностей;
- Клиент-серверная модель: все системные сервисы и драйвера устройств релизованы как серверные процессы со своим собственным окружением исполнения;
- Взаимодействие процессов (InterProcess Communications — IPC) на основе передачи сообщений: использовано для синхронизации процессов и разделения данных;
- Сокрытие прерываний: прерывания преобразуются в передачи сообщений.

MINIX 3.X это новая операционная система с открытым кодом [2], спроектированная на максимальную надёжность, гибкость и защищённость. Она в некоторой мере базируется на предыдущих версиях MINIX, но радикально отличается от них во многих ключевых понятиях. MINIX 1 и 2 были предназначены для учебных целей; MINIX 3 добавляет новую цель, заключающуюся в использовании её как серьёзную операционную систему для встраиваемых и ограниченных ресурсами компьютеров, и для приложений, требующих высокую надёжность.

Ядро MINIX 3.X весьма невелико (около 5000 строк исходного кода) и только оно является кодом, который выполняется на привилегированном уровне ядра. Пользовательские процессы, системные сервисы, включая драйверы устройств, изолированы друг от друга, и выполняются с пониженными привилегиями (рисунок 1). Эти возможности и ряд других аспектов значительно расширяют надёжность системы.

Рисунок 1. Внутренняя структура MINIX 3.X (из [4]). Рисунок 1 ничего не добавляет для понимания текста, поэтому он опущен - см. рисунок в оригинальном тексте.

Системные вызовы реализованы используя пересылку сообщений, как это демонстрируется в следующем фрагменте кода:

```
#include <lib.h>
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;                                /* destination server i.e. PM or FS */
int syscallnr;
register message *msgptr;
{
  int status;
  msgptr->m_type = syscallnr;           /* System Call number */
  status = _sendrec(who, msgptr);       /* Message Transfer: Send the Request
                                           and wait the Reply */
  if (status != 0) {
                    msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
                    errno = -msgptr->m_type;
                    return(-1);
  }
  return(msgptr->m_type);
}
```

Системный вызов, например подобный `getpid()`, реализуется посылкой `GETPID` запроса к MM (это алиас PM), и ожидания ответа от сервера, как это показано в следующем фрагменте кода:

```
#include <lib.h>
#define getpid       _getpid
#include <unistd.h>
PUBLIC pid_t getpid()
{
  message m;
  return(_syscall(MM, GETPID, &m));
}
```

MM есть константой, определённой как:

```
#define MM                PM_PROC_NR
#define PM_PROC_NR        0              /* process manager */
```

Таким образом, процесс получатель жёстко закодирован в коде системного вызова. Применение таких фиксированных адресатов получателей накладывает ограничение, что POSIX и другие системные вызовы будут обслуживаться только PM или FS серверами. Эта проблема обозначена Эндрю Таненбаумом:

Одна вещь, как я думаю, могла бы быть полезной. Текущее значение FS_PROC_NR определено как твёрдо заданная константа (1). Напротив, могла бы быть по-процессные входы в таблице процессов, так, что когда процесс посылает сообщение к 1, это говорило бы ядру искать реальный номер в таблице процессов. Это означало бы, что каждый процесс имел бы «собственный» файловый сервер. То же и относительно PM_PROC_NR.

Для специфических процессов, некоторое число «файловых серверов» могли бы быть пользовательского уровня шлюзовыми процессами, которые имели бы фиксированные TCP соединения к удалённым серверам. Команда, которую посылает пользователь, могла бы затем транслироваться шлюзам локально, а от них — пересылаться на удалённые машины и выполняться там. Это позволио бы использование удалённых файловых систем. На удалённых машинах были бы другие шлюзовые процессы, которые делали бы работу, маршрутизировали и возвращали ответ. Это позволило бы машинам, имеющим сетевое подключение, но не имеющим диска, нормально функционировать, используя удалённый диск. Это могло бы быть оболочкой для «наладонных» (hand-held) мобильных устройств.

Другое ограничение выдвигается на передний план, когда системный программист нуждается в добавлении нового системного вызова. Он/она должен изменить при этом следующие системные файлы:

- `src/include/minix/callnr.h`
- `src/servers/pm/table.c`
- `src/servers/fs/table.c`

И дополнительно: перекомпилировать ядро, FS и PM, а система должна быть перезапущена для вступления изменений в силу.

Разработка таких интересных возможностей как удалённое выполнение процессов, поддержка множественных файловых систем, множественных окружений исполнения, прозрачные прокси и шлюзовые сервера, монитор контроля защищённости, профилирование системных вызовов, и другие — упрощалась бы, если бы передача сообщений системных вызовов могла бы управляться с динамическим присвоением сервера.

Эта статья далее описывает использование связующего сервера (Bindery Server - BS) для разрешения конечных точек процессов, обслуживающих запросы системных вызовов. Этим предложением достигается:

- Гибкость: системные программисты будут добавлять новые системные вызовы с лёгкостью;
- Надёжность: это позволит заменять упавший сервер другим;
- Динамичность: сервер может стартовать/рестартовать в любое время, но процессы смогут воспользоваться его сервисом позже без перезагрузки;

Изменения исходного кода для MINIX 3.1.2a представляются как подтверждение концепции. Должно быть ясно, что они не для произвольно определенной версии MINIX 3.X.

Остаток статьи организован следующим образом. Секция 2 является обзором реализации системных вызовов MINIX 3.X. Секция 3 описывает механизм косвенных сообщений. Секция 4 относится к предложению по реализации `relay()` IPC примитива. Детально тестированный обзор представленный в секции 5, объясняет использования возможностей. Финальная, секция 6 продставляет обсуждения и дальнейшие предложения.

2. Взгляд на реализацию системных вызовов MINIX 3.X

Все процессы в MINIX 3.X могут сообщаться, используя следующие IPC примитивы:

`send()`: послать сообщение процессу;
`receive()`: принять сообщение от специфицированного процесса, или от какого-либо процесса;
`sendrec()`: послать запрос сообщением процессу, и затем принять ответ от него;
`notify()`: послать сообщение процессу, но отправитель не блокируется, если получатель не ожидает сообщения.

Эти примитивы реализованы как прерывания процессора (CPU), которые переводят процессор из режима пользователя в режим ядра.

Как это упоминалось в предыдущей секции, MINIX использует передачу сообщений для реализации системных вызовов. Получателем сообщений запросов являются PM или FS сервера. Ядро MINIX содержит таблицу процессов, чтобы поддерживать атрибуты каждого из процессов. FS и PM имеют их собственные версии таблицы процессов с полями атрибутов, которые им необходимы. Таблица процессов имеет `(NR_TASKS+NR_PROCS)` точек входа (смотри рисунок 2).

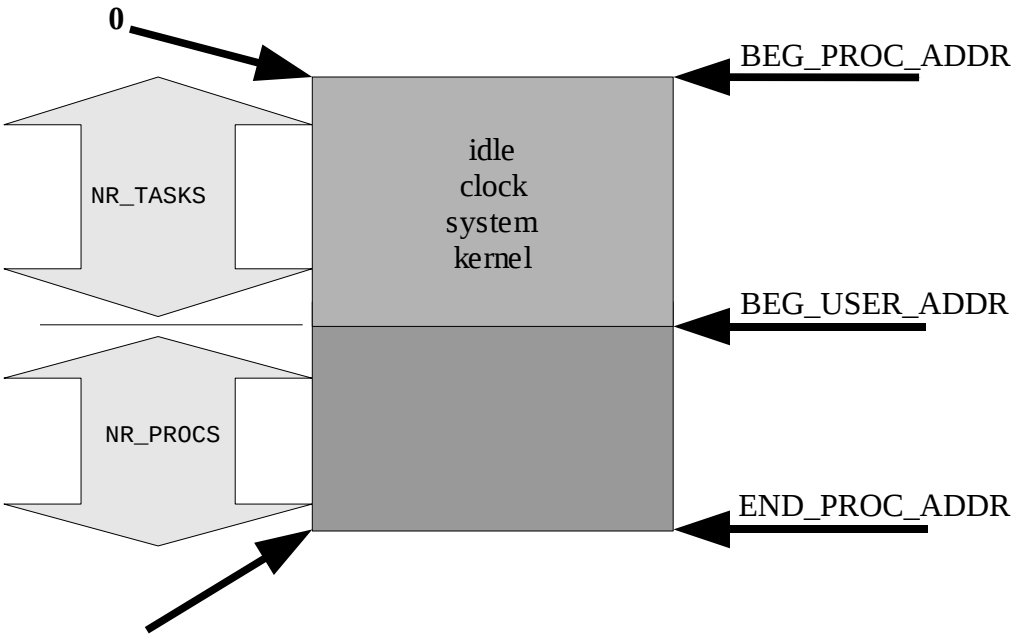

Рисунок 2. Таблица процессов ядра.

`NR_TASKS` входов резервированы для специальных задач (tasks):

- Idle задача
- Clock задача
- System задача
- Фиктивная Kernel задача

`NR_PROCS` входов доступны для серверов, драйверов устройств и пользовательских процессов.

Структура данных proc ядра, которая описывает процесс, имеет два важных поля:

```
proc_nr_t    p_nr;             /* number of this process (for fast access) */
int          p_endpoint;       /* endpoint number, generation-aware */
```

Поле `p_nr` field есть индексом точки входа в таблицу процессов ядра минус `NR_TASKS`,

таким образом, первый процесс (`proc[0]`) в этой таблице имеет `p_nr = (-NR_TASKS)`.

Поле `p_endpoint` есть комбинацией `p_nr` поля и поколения (generation) слота таблицы процессов:

```
#define _ENDPOINT(g, p) ((g) * _ENDPOINT_GENERATION_SIZE + (p))
```

Каждый слот имеет номер поколения, что является счётчиком его использования. Каждый раз, когда процесс использует слот, счётчик поколения инкрементируется. Этот трюк избегает проявления того, что новый процесс, использующий тот же слот, что некоторый завершившийся процесс, сможет принимать сообщения направляемые мёртвому процессу.

Ядро реализует IPC примитивы в функции с путанным названием: `sys_call()`.

```
/*===========================================================================*
 *                                sys_call                                   *
 *===========================================================================*/
PUBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
int call_nr;                  /* system call number and flags */
int src_dst_e;                /* src to receive from or dst to send to */
message *m_ptr;               /* pointer to message in the caller's space */
long bit_map;                 /* notification event set or flags */
```

Параметр `call_nr` в действительности является кодом IPC примитива (`SEND, RECEIVE, SENDREC, NOTIFY, ECHO`). Параметр `src_dst_e` является номером конечной точкой процесса источника или назначения. Остальные параметны комментированы в программном коде.

3. Локальный связующий сервис для системных вызовов

Локальный связующий сервис относится к разрешению номеров конечных точек для системных вызовов через связующий сервер (Bindery Server - BS), аналогично как RPC связывание [5] делается для сетевых сервисов. Локальный связующий сервер (BS) может стартовать после старта системы. На своей инициализации он устанавливает свою базу данных системных вызовов (называемую Global EndPoint Table — GEPT), с концевыми точками PM и FS, затем он ожидает серверной регистрации, или клиентской регистрации. Следующая последовательность описывает шаги динамического связывания для системных вызовов:

- Как только новый сервер (SS: Service Server) будет готов предлагать свои сервисы, он должен зарегистрировать их на BS (сообщение 1 на Рисунке 3);
- Если пользовательский процесс делает системный вызов, он ищет в своей локальной таблице концевых точек (Local EndPoint Table - LEPT) концевую точку для SS в том конкретном системном вызове;
- Первый раз, когда процесс запрашивает этот сервис, он не найдёт его в LEPT, поэтому процесс должен запросить BS для разрешения номера концевой точки желаемого SS (сообщение 2 на Рисунке 3);
- BS просматривает GEPT для запрашиваемого сервиса. Если сервис и его версия совпадают (с запрошенными), BS отвечает процессу номером концевой точки SS (сообщение 3 на Рисунке 3);
- Клиентский процесс сохраняет полученный номер концевой точки в его LEPT, и затем запрашивает системный вызов к SS (сообщение 4 на Рисунке 3);

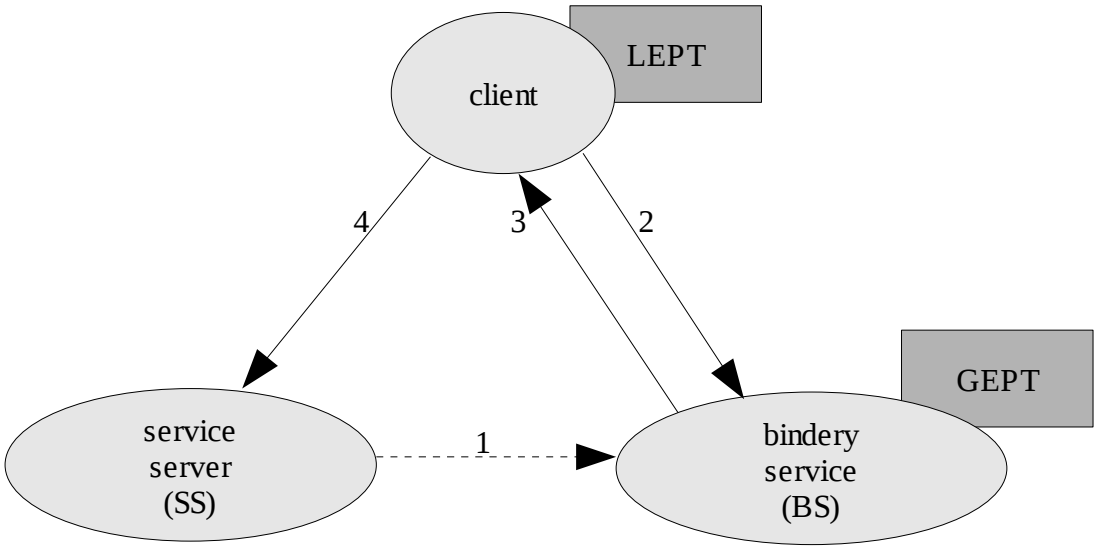

Рисунок 3. Динамическое связывание системных вызовов.

Сообщения 2 и 3 наблюдаются только первый раз для каждого системного вызова, использованного процессом, поскольку LEPT работает как кэш концевых точек. Для достижения динамического связывания потребуются внести изменения в функции библиотеки системных вызовов.

3.1. Структуры данных

Каждый процесс имеет кэш концевых точек сервисов (LEPT). Он имеет следующую C структуру данных:

```
#ifdef LCLBIND
typedef int endpoint_t;                 /* endpoint number              */
#endif /* LCLBIND */
```

Базой данных концевых точек сервисов в BS есть GETP. Она имеет следующую C структуру данных:

```
typedef int endpoint_t;                 /* endpoint number         */
/* Definition of the Global EndPoint Table (GEPT) entry structure */
typedef struct {
   char        svc_name[M3_STRING];     /* name of the service */
   endpoint_t  svr_ep[SERVICE_VERSION]; /* Server Endpoint     */
} gepte_t;
```

Где `SERVICE_VERSION` — число возможных версий для каждого сервиса.

BS имеет вспомогательную таблицу, именуемую `std_sct` (Standard System Call Table) со следующей C структурой:

```
/* Definition of the System Call Table */
typedef struct {
   char        sc_name[M3_STRING];      /* name of the service          */
   endpoint_t  sc_ep;                   /* Server Endpoint              */
   int         sc_nbr;                  /* system call number           */
} sctable_t;
```

`std_sct` статически заполняется именами POSIX сервисов, с концевыми точками серверов (MM/PM или FS), и номерами системных вызовов:

```
#define POSIXCALLS           65
sctable_t std_sct[POSIXCALLS] = {
 {"_exit",   MM, EXIT},
 {"access",  FS, ACCESS},
 {"alarm",   MM, ALARM},
 {"chdir",   FS, CHDIR},
 {"chdir",   FS, FCHDIR},
. . .
 {"getegid", MM, GETGID},
 {"geteuid", MM, GETUID},
 {"getgid",  MM, GETGID},
. . .
```

Новое поле с именем `p_version` было добавлено к структуре данных `proc` дескриптора процессов, для хранения номера версии системного вызова, который каждый процесс будет использовать:

```
struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame    */
. . .
. . .
  int p_endpoint;               /* endpoint number, generation-aware          */
#ifdef LCLBIND
  int              p_version;   /* system call version number                 */
#endif
#if DEBUG_SCHED_CHECK
  int p_ready, p_found;
#endif
};
```

3.2. Инициализация структур данных

BS инициализирует GEPT в две ступени:

1. Устанавливает все элементы таблицы в специальное значение NONE для концевых точек сервисов, указывая тем, что эти позиции пустые;
2. Устанавливает позиции, соответствующие POSIX сервисам, на концевые точки FS и PM, в соответствии со стандартными MINIX системными вызовами, извлекаемыми из таблицы `std_sct`. Номер версии устанавливается в 0 для стандартных сервисов.

```
/*===========================================================================*
 *                              initgept                                     *
 *===========================================================================*/
PRIVATE void initgept()
{
   int i, j;
   for (i = 0 ; i < GEPTSIZE; i++) {
      strncpy(gept[i].svc_name,"no_service",M3_STRING-1);
      for (j = 0 ; j < SERVICE_VERSION; j++)
         gept[i].svr_ep[j] = NONE;
   }
   for (i = 0 ; i < POSIXCALLS; i++) {
      j = std_sct[i].sc_nbr;
      gept[j].svr_ep[0] = std_sct[i].sc_ep;
      strncpy(gept[j].svc_name, std_sct[i].sc_name,M3_STRING-1);
   }
}
```

GEPT инициализирован концевыми точками PM и FS для версии 0, и специализированным значением NONE для концевых точек других версий, как показано в Таблице 1.

Таблица1: Global EndPoint Table - GEPT

Service Name	Version 0	Version 1	Version 2	Version 3	...
_exit	PM endpoint	NONE	NONE	NONE	...
access	FS endpoint	NONE	NONE	NONE	...
alarm	PM endpoint	NONE	NONE	NONE	...
...	...	...	...	...	...

Все программы, которые будут использовать связанные сервисы, должны быть собраны с модифицированной версией C библиотеки `libc.a`. C стартовая процедура периода исполнения `crtso` была модифицирована так, тобы вызывать инициализирующую функцию LEPT с именем `initlept()` ранее вызова функции `main()`.

```
push       ecx          ! push envp
push       edx          ! push argv
push       eax          ! push argc
call _initlept          ! >>> INIT LOCAL ENDPOINT TABLE <<<<
. . .
call       _main        ! main(argc, argv, envp)
push       eax          ! push exit status
call       _exit
```

Каждый процесс, собранный с модифицированной библиотекой `libc.a`, будет иметь LEPT, хранящую номера концевых точек для каждого сервиса, и глобальную переменную с именем `bind_ep`, котора будет загружена значением номера концевой точки BIND.
Функция `initlept()` инициализирует `GEPT` и `bind_ep` в следующем порядке:

1. Устанавливает все элементы таблицы в специальное значение NONE для концевых точек сервисов, указывая тем, что эти позиции пустые;
2. Запросить у PM номер концевой точки для BS, используя `getprocnr()` системный вызов, и загрузить это значение в глобальную переменную `bind_ep`.

```
endpoint_t          lept[LEPTSIZE];
int                 bind_ep;
/*===========================================================================*
 *                           initlept                                        *
 *Gets the BS endpoint number and store it in bind_ep process global variable*
 *===========================================================================*/
void initlept(void)
{
   int i,retcode;
   message m;
   char bindname[]="bind";
   for (i = 0 ; i < LEPTSIZE; i++)
      lept[i] = NONE;
   /***************** BS ENDPOINT ***************/
   m.m1_p1 = bindname;
   m.m1_i1 = -1;                        /* search by name */
   m.m1_i2 = 5;
   if (_syscall(MM, GETPROCNR, &m) < 0)
      bind_ep=NONE;
   else
      bind_ep=m.m1_i1;
}
```

Поле `p_version` инициализируется при загрузке, в `main()` функции ядра:

```
  /* Clear the process table. Anounce each slot as empty and set up mappings
   * for proc_addr() and proc_nr() macros. Do the same for the table with
   * privilege structures for the system processes.
   */
  for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
     rp->p_rts_flags = SLOT_FREE;                  /* initialize free slot */
     rp->p_nr = i;                                 /* proc number from ptr */
     rp->p_endpoint = _ENDPOINT(0, rp->p_nr);      /* generation no. 0 */
     (pproc_addr + NR_TASKS)[i] = rp;              /* proc ptr from number */
#ifdef LCLBIND
     rp->p_version = 0;
#endif
  }
```

Когда процесс выполняет `fork()`, `p_version` должна наследоваться дочерним процессом.

```
/*===========================================================================*
 *                                do_fork                                    *
 *===========================================================================*/
PUBLIC int do_fork(m_ptr)
register message *m_ptr; /* pointer to request message */
{
. . .
#ifdef LCLBIND
       rpc->p_version = rpp->p_version; /* from parent to child */
#endif /* LCLBIND */
. . .
  return(OK);
}
```

3.3. Системный вызов `xfork()`

Вариант системного вызова `fork()` с именем `xfork()` был создан для того, чтобы устанавливать поле `p_version` дочернего процесса. Системный вызов `xfork()` имеет следующий формат:

```
pid_t xfork(int version)
```

где `version` - это значение, загружаемое в поле `p_version` структуры данных `proc` создаваемого дочернего процесса.

Системный вызов `xfork()` вызывает:

1. `sys_fork()`: для создания нового (дочернего) процесса и получения его PID;
2. `sys_setpver()`: новый вызов ядра, который устанавливает поле `p_version` процесса в значение, указанное параметром `version`.

3.4. Изменения задачи SYSTASK

Два новых вызова было добавлено к SYSTASK (`src/kernel/system.c`).

1. Установить версию для системных вызовов процесса (SETPVER): установить номер версии для системных вызовов, значение которого процесс будет использовать для каждого выполняемого системного вызова. Функция `do_setpver()` из SYSTASK используется новым системным вызовом `xfork()`для установки версии системных вызовов в дескрипторе процесса;
2. Возвратить версию для системных вызовов процесса (GETPVER): возвращает значение номера версии системных вызовов, который процесс использует при выполнении каждого системного вызова. Функция `do_getpver()` из SYSTASK использована BS для получения версии системных вызовов из дескриптора процесса, и затем поиска по GEPT этой версии.

3.5. Изменения в библиотеке системных вызовов

Для использования локального динамического связывания, все программы должны компилироваться (связываться) с модифицированной версией библиотеки `libc.a`. Модифицированный исходный код функции `getuid()` представлен как образец:

1. Проверить, что BS выполняется. Если он не выполняется, делать стандартный системный вызов к PM;
2. Проверить, что требуемого сервиса номер концевой точки находится в LEPT. Если он не в LEPY, делать SVCBIND (Service Bind) запрос к BS, затем сохранить отвеченную концевую точку в LEPT для использования в будущем;
3. Запросить разрешённую концевую точку у GETPID сервиса.

```
extern endpoint_t lept[LEPTSIZE];
extern int bind_ep;
PUBLIC pid_t getpid()
{
   message m;
#ifdef LCLBIND
   int server_ep, retcode;
   if (bind_ep == NONE)                              /* bind server not running  */
      return( (pid_t)_syscall(MM, GETPID, &m));      /* standard sys call to MM  */
   if ( (server_ep = lept[GETPID]) == NONE) {        /* is service resolved yet? */
      m.m3_i1 = GETPID;
      retcode = _syscall(bind_ep,SVCBIND,&m);        /* request the binder       */
      server_ep = m.m3_i1;
      if (retcode !=0 )                              /* some error               */
         return( (pid_t)_syscall(MM, GETPID, &m));/* standard sys call to MM  */
         lept[GETPID] = server_ep;                   /* save the resolved service*/
   }
   return( (pid_t)_syscall(server_ep,GETPID,&m)); /* request the service      */
#else
   return(_syscall(MM, GETPID, &m));
#endif
```

3.6. Сервисы связующего сервера

BS предлагает четыре сервиса.

1. Инсталляция сервиса: используется серверами (SS) при регистрации их сервисов. Они запрашивают BS установить специфические версии системных вызовов, которые будут обслуживаться ими.
2. Деинсталляция сервиса: используется серверами (SS) снять регистрацию своих сервисов. Они запрашивают сбросить специфические версии системных вызовов, обслуживаемые ими.
3. Связывание сервиса: используется программами, собранными с модифицированной версией библиотеки `libc.a`, использующих динамическое связывание. Процесс запрашивает номер концевой точки сервиса у BS, который ищет концевую точку и версию по запрашивающего процесса дескриптору, возвращая концевую точку SS процессу.
4. Дать глобальную таблицу концевых точек (Global Endpoint Table — GEPT): этот сервис может быть использован любым процессом, который нуждается в копии BS GEPT. Он используется также модифицированным информационным сервером (Information Server — IS) для выввода дампа GEPT на экран консоли по нажатию комбинации клавиш `<Shift><F9>`.

4. Тестирование

Для проверки концепции и её реализации, простой `fakepm` сервер был создан, который

всего лишь предлагает один системный вызов `getpid` в версии 1. Сервер `fakepm` всегда возвращает результат PID равный 333 для каждого процесса, обращающегося к нему. Как и другие сервера пользовательского пространства, `fakepm` запускается командой `service`, и его процесса номер концевой точки присваивается в динамический способ ядру. Как только он стартует, он регистрирует `getpid` сервис версии 1, с помощью BS, после чего ожидает клиентских запросов.

```
# service up /sbin/fakepm
fakepm: bind server with bind_nr=98 and bind_ep=35637
bind: SVCSET from 71086 service:20 version=1 name=getpid
bind: do_svcset: name=getpid nbr=20 ver=1 ep=71086
```

Чтобы проверить номера концевых точек BS и сервера fakepm, нажимаем <F1>:

```
-nr-----gen---endpoint--name--- -prior-quant- -user---sys----size-rts flags-
(-4)      0         -4 idle      15/15 01/08  49624    182    80K ------
[-3]      0         -3 clock     00/00 08/08      0      0    80K --R--- ANY
[-2]      0         -2 system    00/00 08/08    197      0    80K ------
[-1]      0         -1 kernel    00/00 08/08      0      0    80K M-----
  0       0          0 pm        03/03 32/32     21      0   124K --R--- ANY
  1       0          1 fs        04/04 32/32     32      0  4956K --R--- ANY
  2       0          2 rs        03/03 04/04      2      0    48K --R--- ANY
  3       0          3 mem       02/02 04/04      2      0   304K --R--- ANY
  4       0          4 log       02/02 04/04     27      0    76K --R--- ANY
  5       0          5 tty       01/01 04/04     34      0    84K --R--- ANY
  6       0          6 ds        03/03 04/04      0      0    24K --R--- ANY
  7       0          7 init      07/07 08/08      0      1    16K --R--- pm
  8       2      71086 fakepm    03/03 04/04      0      0    28K --R--- ANY
 10       1      35549 pci       03/03 04/04     12      0    48K --R--- ANY
...
 98       1      35637 bind      03/03 04/04      0      0    28K --R--- ANY
```

GETPID сервиса регистрация может быть подтверждена нажатием <Shift><F9>:

```
BIND Global End Point Table dump
---service-- number version0 version1 version2 version3 version4
no_service        0    NONE     NONE     NONE     NONE     NONE
_exit             1       0     NONE     NONE     NONE     NONE
fork              2       0     NONE     NONE     NONE     NONE
.....
getpid           20       0    71086     NONE     NONE     NONE
mount            21       1     NONE     NONE     NONE     NONE
```

Шаги выполняемые клиентской тестовой программой (`bindtest`) и связанными с ней процессами, есть:

- Стартовая процедура периода исполнения `crtso()` выполняет функцию `initlept()`, которая инициализирует LEPT, и глобальную переменную процесса `binder_ep` со значением концевой точки BS.
- Она же (`crtso()`) вызывает `xfork()` с аргументом version равным 1, создавая дочерний процесс с номером версии 1.
- Родительский процесс выводит PID дочернего процесса на экран, после чего дожидается завершения этого порождённого процесса.
- Порождённый процесс делает системный вызов `getpid()`. Поскольку LEPR позиция для `getpid` сервиса равна NONE, процесс делает SVCBIND (Service Bind) запрос к BS.
- Если сервер `fakepm` выполняется, BS содержит его концевую точку в версии номер 1 в GEPT для сервиса `getpid`.

- BS запрашивает SYSTASK о номере версии процесса `bindtest`.
- SYSTASK возвращает значение 1 к BS.
- BS возвращает для дочернего процесса `bindtest` значение концевой точки `fakepm`.
- Дочерний процесс вызывает `getpid()`, используя номер концевой точки сервера `fakepm`.
- Сервер `fakepm` возвращает значение 333 как PID.
- Дочерний процесс выводит этот PID, который отличается от PID, который его родитель получил как результат `xfork()`.

Консольный вывод	Пояснения
initializing LEPT BS endpoint = 35637	Отладочный вывод стартовой C процедуры времени исполнения crtso(), выполнение функции initlept(), которая нициализирует LEPT и глобальную переменную процесса binder_ep значением концевой точки BS (35637).
SYSTASK: do_setpver proc_nr=45 version=1 PM: do_xfork proc_nr=45 p_version=1	Отладочный вывод SYSTASK функции do_setpver(), и PM функции do_fork() когда bindtest вызывает xfork() для создания дочернего процесса с номером версии равным 1.
getpid: bind_ep=35637 getpid: start getpid:lept[GETPID] = NONE	Отладочный вывод getpid() библиотечной функции когда bindtest дочерний процесс делает getpid() системный вызов, и поскольку LEPT вход для getpid() сервиса равен NONE, он делает SVCBIND (Service Bind) запрос к BS.
BS: SVCBIND from 71123 service:20 BS: do_svcbind(): svcnbr =20 SYSTASK: do_getpver() proc= 71123 version=1 BS: do_svcbind(): p_version = 1	BS принимает запрос SVCBIND для сервиса 20 (getpid) от bindtest дочернего процесса (номер концевой точки 71123). BS делает запрос (taskcall) Get Process Version (getpver()) к SYSTASK, который возвращает p_version = 1.
getpid:SVCBIND retcode= 0 getpid:server_ep for GETPID = 71086 getpid:GETPID request to enpoint 71086	Отладочный вывод библиотечной функции getpid() которая получила номер концевой точки fakepm (71086) для GETPID версии 1.
fakepm: GETPID request received from 71123	Отладочный вывод fakepm, когда он принимает GETPID запрос от bindtest дочернего процесса (71123).
bindtest parent: my child is 132	Родительский процесс bindtest выводит на экран действительный PID своего дочернего процесса, который был ему возвращён системным вызовом xfork(), после чего он только ожидает завершения потомка.
bindtest child: getpid()= 333	Дочерний процесс bindtest выводит подменённый PID, возвращённый ему сервером fakepm.

5. Обсуждение и дальнейшие работы

Современная архитектура MINIX 3.X, базирующаяся на микроядре и драйверах устройств в пользовательском пространстве, делает её надёжной. Передача сообщений является парадигмой реализации системных вызовов, вызовов задачи и вызовов ядра. Изъяном реализации MINIX3 является тот факт, что системные вызовы обслуживаются FS или PM. Если необходимо добавить новый системный вызов, то некоторые константы ядра, и FS и PM таблицы системных вызовов должны быть изменены, эти сервера должны быть перекомпилированы, а система перезагружена. Главным преимуществом локального связывающего сервиса есть то, что новые сервисы могут быть имплементированы без перекомпиляции ядра/серверов, и без рестарта системы, повышая производительность процесса разработки. Связывающий сервис представляет новые разработческие перспективы. Эта статья рассматривает механизмы и реализацию связывающий сервис, применяемые на уровне пользовательских процессов. Примером такого использования могло бы быть использование стандартного файлового сервера (FS) для одних процессов, и других серверов, которые поддерживают EXT2/3/4, FAT16/32, VFAT, NTFS и другие — для других процессов. Другое приложение могло бы быть реализацией удалённой файловой системы через File System Proxy/Gateway Server только подменой номера версии в дескрипторе процесса. Новые IPC примитивы, подобные RELAY, предлагаемому в [6], могли бы нуждаться в поддержке со стороны таких возможностей. С использованием локального связывания MINIX 3.X мог бы иметь множественные процессы управления памятью (отличного от PM), которые могли бы управлять различные ареалы памяти различными механизмами, в одном случае используя Buddy алгоритм, в другом First Fit, в третьем Best Fit.

Предлагаемые изменения являются только доказательством концепции, но некоторые расширения уже могли бы быть добавлены. Процесс мог бы иметь модифицируемую LEPT с номерами концевых точек серверных процессов для каждого системного вызова и номера версии для него. Таким образом, системный вызов мог бы иметь различную версию для каждого сервиса для одного и того же процесса. Локальный связывающие сервисы не ограничен системными вызовами, они могут быть использованы и с вызовами задач, и с вызовами ядра, открывая широкие возможности для разработчиков.

Поскольку MINIX 3.X представляется как серьёзная система, используемая на компьютерах с ограниченными ресурсами и для приложений требующих высокой надёжности, её простота и универсальность являются атрибутами, которые делают её хорошим выбором для целей обучения и исполняемым испытательным стендом для развития ОС, и расширений что могут быть с лёгкостью добавлены до неё.

Источники информации

[1] Tanenbaum Andrew S., Woodhull Albert S., «Operating Systems Design and Implementation, Third Edition», Prentice-Hall, 2006.
[2] MINIX3 Home Page, http://www.minix3.org/
[3] Jorrit N. Herder, «Towards A True Microkernel Operating System», master degree thesis, 2005.
[4] Jorrit N. Herder, Herbert Bos, Ben Gras, Philiph Omburg ,Andrew S. Tanenbaum, «Modular system programming in MINIX 3», ;Login: April 2006.
[5] RFC1833 - Binding Protocols for ONC RPC Version 2. http://www.faqs.org/rfcs/rfc1833.html
[6] Pessolani, Pablo A., «MINIX 3.X Message Indirection», 2009

Перевод: О.Цилюрик , 10.12.2009
Оригинал находится по адресу:
http://sites.google.com/site/minix3projects/MINIX3-BIND-REPORT.pdf
«MINIX 3.X: Local Bindery Service»