РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе на тему:

"Мониторинг виртуальной памяти в ОС Linux"

# Введение

Зачастую бывает необходимо отследить за работой того или иного процесса с памятью, например, чтобы обнаружить утечки памяти, узнать, в какие моменты и сколько памяти процесс выделяет. Для решения данной задачи существует ряд средств, а именно:

* *Файловая система /proc* – позволяет прочесть различную информацию о всей системе в целом и о каждом из процессов, в том числе информацию об использовании процессом памяти и об отображениях памяти данного процесса. (пример:

# cat /proc/`pgrep test`/status

Name: test1

…

VmPeak: 1556 kB

VmSize: 1544 kB

VmLck: 0 kB

VmHWM: 308 kB

VmRSS: 308 kB

VmData: 148 kB

VmStk: 88 kB

VmExe: 4 kB

VmLib: 1276 kB

VmPTE: 12 kB

# cat /proc/`pgrep test`/maps

08048000–08049000 r-xp 00000000 08:01 17432879 /home/twee/work/mstu/coding/memmon/test/test1

08049000–0804a000 rw-p 00000000 08:01 17432879 /home/twee/work/mstu/coding/memmon/test/test1

0804a000–0806b000 rw-p 0804a000 00:00 0 [heap]

b7e4b000‑b7e4c000 rw-p b7e4b000 00:00 0

b7e4c000‑b7f75000 r-xp 00000000 03:05 1604119 /lib/tls/libc‑2.3.6.so

b7f75000‑b7f76000 r–p 00128000 03:05 1604119 /lib/tls/libc‑2.3.6.so

b7f76000‑b7f79000 rw-p 00129000 03:05 1604119 /lib/tls/libc‑2.3.6.so

b7f79000‑b7f7c000 rw-p b7f79000 00:00 0

b7f9d000‑b7fb3000 r-xp 00000000 03:05 752968 /lib/ld‑2.3.6.so

b7fb3000‑b7fb5000 rw-p 00015000 03:05 752968 /lib/ld‑2.3.6.so

bfc2a000‑bfc40000 rw-p bfc2a000 00:00 0 [stack]

ffffe000‑fffff000 r-xp 00000000 00:00 0 [vdso]

). На приведенной распечатке видно, сколько памяти использует процесс test и под какие именно нужды, а так же его карту памяти. Однако таким образом нельзя отследить динамику работы процесса с памятью.

* *strace* – утилита, позволяющая трассировать все системные вызовы, выполняемые данным процессом (в частности, выделение памяти вызовами *brk/mmap*). Она использует стандартный отладочный механизм ядра под названием **ptrace –** подключается к исследуемому процессу как отладчик (вызовов *ptrace(),* указывая при этом флаг **PTRACE\_SYSCALL**, что заставляет систему уведомлять трассирующий процесс о всех системных вызовах трассируемого). Пример его работы:

execve (»./test3», [«test3»], [/\* 61 vars \*/]) = 0

…

fsync(0) = -1 EINVAL (Invalid argument)

mmap2 (NULL, 2101248, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xb7bf4000

fsync(1) = -1 EINVAL (Invalid argument)

fsync(2) = -1 EINVAL (Invalid argument)

munmap (0xb7bf4000, 2101248) = 0

exit\_group(0)

На приведенной трассировке видно, как процесс выделяет и освобождает 2101248 байт памяти. К сожалению, это средство не позволяет следить за всеми процессами в системе в целом, а так же за выделениями физических страниц процессу.

Таким образом, возникает необходимость в средстве, позволяющем отслеживать не только выделения виртуальной памяти процессу, но и выделения отдельных страниц физической памяти в результате страничных сбоев.

# Аналитический раздел

## Постановка задачи

Разработать драйвер под Linux, отслеживающий выделение и освобождение процессами виртуальной памяти и выделение физических страниц при страничных отказах.

Драйвер должен поддерживать динамическую загрузку и выгрузку без перезапуска системы и задание списка процессов, за которыми необходимо выполнять мониторинг.

Взаимодействие с пользовательской программой осуществляется посредством файлов, создаваемых в файловой системе */proc*.

Ядро отвечает за создание и уничтожение процессов и за их связь с внешним миром (ввод и вывод). Взаимодействие процессов друг с другом (посредством сигналов, программных каналов или других средств межпроцессного взаимодействия) является основой общей системной функциональности, и тоже осуществляется при помощи ядра. Планировщик, распределяющий время центрального процессора между процессами, является частью подсистемы управления процессами. В общих словах, подсистема управления процессами реализует абстракцию множества процессов, работающих одновременно на одном или нескольких процессорах.

### **Управление памятью**

Память компьютера – один из главных ресурсов, и производительность системы критически зависит от политики распределения памяти. Ядро создает виртуальное адресное пространство для каждого процесса, используя при этом ограниченное количество физической памяти и, при необходимости, вторичную память, такую, как жесткий диск. По мере необходимости страницы могут быть выгружены в файл подкачки, либо файл, из которого они были отображены в память (в случае, если они не были модифицированы с момента загрузки из файла, они просто удаляются из памяти). По умолчанию ядро не позволяет выделить одному процессу больше памяти, чем суммарный объем доступной оперативной и swap‑памяти. Однако есть такая возможность, как overcommit («перевыделение»), которая позволяет выделить гораздо больше памяти, при условии, что реально использоватьcя будет лишь небольшая ее часть (допустим, при работе с разреженным массивом). Overcommit включается командой

# echo 1 > /proc/sys/vm/overcommit\_memory

а отключается

# echo 0 > /proc/sys/vm/overcommit\_memory

Цифра 1 означает выбранный режим управления перевыделением (0 означает его отсутствие, 1 – допустимо перевыделение неограниченных объемов памяти, 2 – некоторый эвристический алгоритм определения максимально допустимого объема перевыделения).

### **Файловая система**

Система Linux основана на концепции файловой системы. Практически любой объект системы может быть рассмотрен как файл. Ядро строит единую иерархическую файловую систему (единое дерево директорий) на основе не обладающего иерархической структурой оборудования. В результате абстракция файла активно используется всей системой.

### **Подсистема ввода-вывода**

Практически любая операция в системе есть, по сути, операция с устройством. За исключением процессора, памяти и очень небольшого числа других элементов, порядок выполнения работы с устройством, а следовательно и исполняемый код, выполняющий эту работу, зависит главным образом от конкретного устройства. Такой код называется драйвером устройства. Ядро должно включать в себя драйверы для всех периферийных устройств, входящих в систему, от жесткого диска до клавиатуры.

### **Сетевая подсистема**

Ядро должно управлять работой с сетью, поскольку большинство сетевых операций не зависит от процесса. Приходящие сетевые пакеты являются асинхронными событиями, т.е. во время работы одного процесса может прийти пакет, адресованный другому процессу. Пакеты должны быть приняты, распознаны и распределены до того, как о них узнает процесс. Система отвечает за передачу данных через программные и сетевые интерфейсы, а так же управлять выполнением программ в соответствии с работой сети. К тому же, все задачи маршрутизации и выделения сетевых адресов выполняет ядро.

### **Системные вызовы**

Системные вызовы, такие как *open()*, *fork()*, *read()*, etc являются связующим интерфейсом между ядром и пользовательскими приложениями. В Linux 2.6 существует около 330 различных вызовов (многие из них избыточны или сохранены по причинами совместимости). Их вызов происходит через прерывание 0x80 или инструкцию sysenter (на современных процессорах). При этом в регистр **EAX** помещается номер системного вызова, а в остальные 6 регистров (кроме **ESP**) – аргументы (т.е. любой системный вызов может принимать до шести 32‑битных аргументов) в порядке **EBX, ECX, EDX, ESI, EDI, EBP**. Точка входа всех системных вызовов расположена в файле *arch/i386/kernel/entry.S*, который вызывает обработчик конкретного вызова по таблице вызовов *sys\_call\_table,* передавая ей регистры через стек.

### **Загружаемые модули**

Одна из важных особенностей ядра Linux – это способность расширять собственную функциональность непосредственно в период выполнения.

Каждый фрагмент исполняемого кода, который может быть добавлен в ядро во время его работы, называется модулем ядра. Каждый модуль создается из объектного кода, не связанного в полноценный исполняемый файл. Модуль может быть загружен в ядро с помощью программы insmod (вызывающей функции *create\_module() / init\_module()*), и выгружен с помощью rmmod (вызывающего *delete\_module()*). В данной работе реализует именно такой динамически загружаемый модуль.

### **Типы устройств**

В Linux различают три основных типа устройств. Каждый драйвер обычно соответствует одному из этих типов. Выделяют:

* Символьные драйверы
* Блочные драйверы
* Сетевые драйверы

Символьное устройство может рассматриваться как поток байт (так же как и файл); символьный драйвер должен реализовывать такое поведение. Такой драйвер имеет, как правило, функции открытия, закрытия, чтения и записи. Текстовая консоль и последовательный порт – примеры символьных устройств. Они могут легко быть представлены абстракцией потоков. Работа с символьными устройствами осуществляется через специальные файлы устройств, находящиеся в директории /dev. Единственное значимое отличие обычного файла от такого устройства – это произвольный доступ, тогда как к большинству символьных устройств можно обращаться лишь последовательно.

Как и к символьным, доступ к блочным устройствам можно получить через файлы в директории /dev. Блочное устройство – это устройство (например, диск), способное содержать в себе файловую систему. В системе Unix блочное устройство может лишь передавать один или более целых блоков данных, обычно по 512 байт. Интерфейс взаимодействия блочных драйверов с ядром значительно отличается от интерфейса символьных драйверов.

Любая сетевая транзакция выполняется через интерфейс сокетов. Сетевой интерфейс отвечает за передачу и прием пакетов под управлением сетевой подсистемы ядра вне зависимости от того, к каким именно транзакциям они относятся. Многие сетевые соединения (особенно использующие протокол TCP) ориентированы на потоки данных. Но сетевые устройства обычно работают с пакетами, а не с потоками. Таким образом, сетевые устройства содержат в себе черты как символьных, так и блочных.

# Конструкторский раздел

## Модульная структура драйвера

Драйвер memmon состоит из следующих модулей:

*mmon.c* – основной модуль, отвечающий за инициализацию и выгрузку драйвера

*mm-fault.c* – обработчик страничных ошибок

*syscalls.c* – высокоуровневая часть перехвата системных вызовов

*syscalls-entry.S* – низкоуровневая часть перехвата системных вызовов

*watch-pids.c* – список процессов, за которыми осуществляется мониторинг, добавление и удаление из него

*events.c* – кольцевой буфер событий

## Инициализация и выгрузка драйвера

Инициализацию драйвера выполняет функция *int \_\_init init(void).* Она вызывается при загрузке драйвера в контексте процесса, вызвашего *init\_module()* (системный вызов загрузки драйвера) и выполняет следующие действия:

1. Инициализирует битовую карту отслеживаемых процессов и кольцевой буфер событий
2. Устанавливает обработчики системных вызовов и страничной ошибки
3. Создает директорию */proc/memmon* и файлы в ней

Создание файлов происходит в последнюю очередь для того, чтобы пользовательские приложения не могли обратиться к драйверу до завершения инициализации.

Выгрузку выполняет функция *void \_\_exit exit(void)*, вызываемая в контексте процесса, сделавшего *delete\_module()*. Она выполняет действия, обратные к *init():*

1. Удаляет директорию */proc/memmon*
2. Снимает перехват системных вызовов и страничного отказа
3. Освобождает память

## Взаимодействие с пользовательскими приложениями

Для взаимодействия с пользовательскими приложениями драйвер использует файловую систему *procfs –* псевдо-файловую систему, предоставляющую различную информацию о системе. Любой драйвер может добавлять в ее иерархию свои файлы и папки для передачи пользовательским программам различной информации. Ядро предоставляет ряд функций для работы с *procfs,* из который данный драйвер использует следующие:

*proc\_mkdir() –* создает папку в */proc*

*create\_proc\_entry() –* создает файл в указанной папке */proc* или самой */proc*

*remove\_proc\_entry() –* удаляет папку или файл в */proc*

На уровне ядра Linux любой открытый файл представлен структурой *file,* хранящей указатель *f\_fops* на структуру *file\_operations,* содержащую адреса обработчиков различных запросов к файлу. Допустим, когда приложение открывает файл (делая системный вызов *open()*), он вызывает обобщенный уровень файловых систем VFS (функцию *vfs\_read()*), которая в свою очередь вызывает обработчик *f\_ops->open.* Для обычных файловых систем обработчики запросов к файлу находятся в драйвере файловой системы, которой принадлежит данный файл. Однако */proc* не представляет из себя какой-либо реальной ФС на реальном хранилище данных, и каждый драйвер, добавляющий туда файлы, должен для них предоставлять свои обработчики (точки входа), которые и будут вызываться при работе с этими файлами.

Файлы, создаваемые в /proc, представляются структурой proc\_entry, содержащую поле proc\_fops, куда и заносится указатель на структуру file\_operations для данного файла.

Данный драйвер создает в папке /proc/memmon 2 файла – watch-pids – для добавления / удаления процессов в список отслеживаемых и events – файл, содержащий собственно лог событий.

## Перехват системных вызовов

Одно из основных действий, выполняемых драйвером – перехват системных вызовов.

Это довольно небезопасный прием, так как если 2 драйвера перехватят один и тот же вызов, и будут выгружены не в том порядке, в каком загрузились, последний восстановит неверный адрес в таблице вызовов, вследствие чего произойдет сбой при следующей попытке сделать данный вызов. В связи с этим, начиная с версии 2.5, ядро более не экспортирует таблицу системных вызовов. Тем не менее, эта проблема устраняется небольшим исправлением ядра (patch), который добавляет в произвольный файл ядра следующие строки, экспортирующие из ядра данную таблицу.

extern void \*sys\_call\_table[];

EXPORT\_SYMBOL\_GPL (sys\_call\_table);

Для перехвата системных вызовов имеются 3 таблицы указателей – оригинальных (системных) обработчиков, наших пре-обработчиков (вызываемых ДО оригинального, и принимающих аргументы) и пост-обработчиков (вызываемых ПОСЛЕ и принимающих возвращаемое значение):

void \*old\_sys\_call [NR\_syscalls];

void \*sys\_call\_trap [NR\_syscalls];

void \*sys\_call\_exit [NR\_syscalls];

Кроме того, есть общая таблица структур, каждый элемент которой описывает один из перехватываемых системных вызовов:

struct syscall\_handler

{

/\* Syscall nr \*/

int nr;

/\* Pre-call & post-call handler \*/

void \*hand1, \*hand2;

};

Функция *capture\_syscalls()*, вызываемая при инициализации драйвера, копирует эти адреса в предыдущие 2 таблицы и записывает в *sys\_call\_table* по нужным номерам адрес универсального перехватчика – *syscalls\_entry*, находящегося в файле *syscalls-entry.*

Необходимость в ассемблерном коде обусловлена механизмом обработки системных вызовов в Linux. На рис. 3 показан стек на момент вызова обработчика системного вызова (замененного или стандартного). Проблема заключается в том, что некоторые стандартные обработчики требуют, чтобы стек имел именно такой вид, и если вызывать их из нового обработчика, они правильно работать не будут. В связи с этим syscalls\_entry сначала вызывает пре-обработчик системного вызова, затем заменяет в стеке адрес возврата на адрес следующей инструкции и делает переход на стандартный обработчик, который получает кадр стека в изначальном виде. Затем, при возврате, мы попадаем на следующую инструкцию, которая вызывает пост-обработчик и делает перход на изначальный адрес возврата, на код в *arch/i386/kernel/entry.S* (точки входа всех системных вызовов в Linux). Этот адрес сохраняется внизу стека ядра, там же где хранится указатель на текущую задачу и некоторая другая служебная информация ядра. Данные действия продемонстрированы на рис. 3.3–3.5.

Данный драйвер перехватывает следующие системные вызовы:

*m[un] map() –* выделение / освобождение региона памяти

*mremap() –* перемещение региона памяти

*brk() –* расширение / сужение сегмента данных программы

*m[un] lock[all]() –* блокировка набора страниц в рабочем множестве процесса

*fsync() –* используется в качестве маркера в журнале событий

Для каждого из вызовов в журнал печатается имя вызова, PID вызвавшего процесса и список (с расшифровкой, там, где это имеет смысл) аргументов.

## Кольцевой буфер событий

Для хранения журнала событий, таких как выделение блоков виртуальной памяти и страниц, использует кольцевой буфер, защищенный спин-блокировкой. Размер данного буфера может задаваться при загрузке драйвера в качестве его параметра, значение по умолчанию – 32 кб, минимальное – 8 кб. Память для буфера выделяется функцией *kzalloc()* – аналогом фукнций *malloc/calloc()* из стандартной библиотеки С. Передаваемый ей параметр **GFP\_KERNEL** означает, что память выделяется для ядра (т.е. не может быть позже выгружена с диска), но не в атомарном контексте (т.е. текущий процесс может быть отложен до освобождения необходимой памяти).

Каждая запись в буфере представляет из себя следующую структуру:

enum memmon\_event\_type – тип события

{

NOTUSED = 0

MMAP2,

MUNMAP,

MREMAP,

MLOCK,

MUNLOCK,

MLOCKALL,

MUNLOCKALL,

BRK,

FSYNC,

ANON\_PF, – страничная ошибка на анонимной странице

SWAP\_PF, – на странице из файла подкачки

FILE\_PF, – из разделяемого файла

SYSCALLRET – возврат из системного вызова

};

struct memmon\_event

{

enum memmon\_event\_type type; – тип события

pid\_t pid; – PID вызвавшего процесса

union – специфичны для события данные

{

struct

{

void \_\_user \*start;

size\_t len;

} munmap;

struct

{

void \_\_user \*start;

size\_t len;

unsigned long prot, flags;

unsigned long fd, off;

} mmap2;

……

};

}

С буфером событий связаны следующие переменные:

static struct memmon\_event \*events; – собственно буфер

static int ev\_start; – индекс самой старой записи в буфере

static int ev\_end; – индекс последней записи

static int ev\_ovf = 0; – было ли уже переполнение буфера

DECLARE\_WAIT\_QUEUE\_HEAD (ev\_waitq); – очередь ожидания (для блокирующего чтения)

spinlock\_t ev\_lock = SPIN\_LOCK\_UNLOCKED; – спин-блокировка для защиты от гонок при обращении к буферу

Пользовательские приложения запрашивают содержимое буфера событий, читая файл */proc/memmon/events*. Если при открытии файла не был установлен флаг **O\_NONBLOCK**, операция чтения по нему блокирующая – т.е., если новых данных в буфере нет, *read()* переводит процесс в состояние ожидания (**interruptible sleep**) вызовом функции *wait\_event\_interruptible()* до получения сигнала либо появления новых данных в буфере.

Помимо *open()* и *release(),* вызываемых при открытии (создании новой структуры *file*) и ее уничтожении, в *file\_operations* данного файла определены всего 2 точки входа – *read()* и *poll().* Обработчик *poll()* вызываемается, когда какой-то процесс делает вызов *select()* по данному файлу – ожидает, пока на нем будут доступные для чтения данные. Кроме того, в флагах структуры *file* вызовом *nonseekable\_open()* сбрасывается бит, позволяющий делать вызов *llseek()* по файлу (т. к. данная операция лишена смысла для кольцевого буфера).

Для реализации функции *read()* используется абстракция под названием *seq\_file*, предназначенная для буферизации считываемых данных. Она требует задания 4 функций – *seq\_start()*, вызываемой при начале чтения из файла, *seq\_next()*, вызываемой перед копированием в буфер пользователя записи об очередном событии, *seq\_show()*, собственно осуществляющющей это копирование, и *seq\_stop()*, вызываемой при завершении передачи данных в пользовательский буфер (когда скопировано затребованное количество данных или не осталось событий в буфере).

Рис. показывает связь между этими функциями и структурами.

При добавлении нового события в буфер происходит пробуждение все ожидающих на очереди процессов вызовом *wake\_up\_interruptible\_sync()* (sync означает, что текущий процесс не будет вытеснен до разблокировки всех процессов).

## Набор отслеживаемых процессов

Набор отслеживаемых процессов представляется в виде битовой карты на 65536 записей. При запуске нового процесса ядро дает ему PID, на 1 больший, чем PID предыдущего процесса. При достижении очередным процессом PID, равного 65536, новым процессам PID выделяется с единицы (т.е. с первого положительного незанятого). Лишь когда количество процессов в системе превышает эту цифру, ядро начинает выделять бОльшие номера. Данная ситуация крайне маловероятна и драйвером не обрабатывается.

Для добавления и удаления PID'ов отслеживаемых процессов создается файл */proc/memmon/watch-pids* с обработчиками *open(), release(), read(), write()*.

Обработчик *open()*, в случае, если файл открыт для чтения, выделяет буфер ядра и распечатывает туда содержимое текущей битовой карты (при помощи функции *bitmap\_scnlistprintf()*). Попытка открыть файл одновременно на чтение и запись приводит к ошибке **EINVAL**.

Обработчик *read()* считывает запрошенный пользователем блок данных из этого буфера

Обработчик *write()* считывает одно число (возможно, со знаком) из пользовательского буфера. Если оно положительное, оно воспринимается как PID процесса, который необходимо добавить в битовую карту. Если отрицательное – соответствующий PID удаляется. Если 0 – битовая карта обнуляется (не отслеживается ни один процесс).

## Перехват страничных ошибок и выделений страничных фреймов

Возможно осуществить перехват страничных отказов, подменив смещение обработчика исключения в IDT, однако данный метод требует повторения того же анализа сбойного адреса, который делает и стандартный системный обработчик, (файл mm/memory.c) что привело бы к падению производительности системы. В связи с этим было решено внести еще одну модификацию в ядро. При обработке страничного отказа ядро сначала определяет, относится ли сбойная страница к виртуальной памяти процесса (если нет, ему посылается сигнал **SIGSEGV**), после чего вызывает функцию *handle\_pte\_fault()*. Та анализирует причину отказа и либо подгружает страницу из своп-файла / файла отображения, либо выделяет процессу новую страницу, либо посылает ему **SIGSEGV** (например, при попытке записи в регион памяти, доступной только для чтения), либо происходит ситуация **OOM** (out of memory), в результате которой сбойный процесс уничтожается с целью освобождения памяти. Модификация ядра добавляет глобальную переменную-указатель на внешнюю функцию, которую каждый раз вызывает *handle\_pte\_fault*. При инициализации драйвер заносит туда адрес своей функции *mm\_handle\_fault()*, которая, в зависимости от причины страничного отказа, заносит в буфер одно из трех событий (вместе с адресом и типом доступа – чтение либо запись):

**ANON\_PF** – сбой при обращении к новой (еще не выделенной из физической памяти) страницы;

**SWAP\_PF** – сбой при обращении к странице, которая находится в файле подкачки;

**FILE\_PF** – сбой при обращении к странице, которая может быть загружена из файла, отображенного в память (например, код разделяемой библиотеки).

Алгоритм анализа причины страничного отказа продемонтсрирован на рис. 3.7.

## Синхронизация

Так как доступ к кольцевому буферу происходит из различных процессов, необходимо какое-то средство предохранения от «гонок» (race conditions). Ядро Linux предоставляет целый набор примитивов синхронизации. Однако, т. к. блокировка выполняется в том числе и в атомарном контексте, т.е. когда текущий процесс не может быть переведен в режим ожидания (например, при помещении события из обработчика прерывания), единственный подходящий примитив – спин-блокировка (spin-lock), т.е. простая бинарная переменная (со значением 0 либо 1) и активным ожиданием на ней. Захват спин-блокировки происходит при начале операции чтения буфера событий и перед добавлением в буфер нового события. Освобождение – по завершении чтения и добавления события, а так же перед переводом текущего процесса в режим ожидании в случае, когда новых данных в буфере нет.

# Технологический раздел

## Язык и средства программирования

Ядро Linux написано на языке С с небольшим количеством кода на ассемблере, и его система сборки, вообще говоря, поддерживает только данные языки для использования в модулях. Выбор из них был сделан в пользу С по причине значительно большей структурированности написанного на нем кода. Однако, небольшой объем кода драйвера пришлось написать на ассемблере, в силу чего он не является платформенно-независмым и привязан к архитектуре x86.

Для сборки драйвера используется сборочная система ядра make/Kbuild и компилятор С из gcc (GNU Compiler Collection). Особенностью ядра Linux является отсутствие совместимости на бинарном уровне, совместимость присутствует лишь на уровне исходных текстов, вследствие чего все модули и само ядро должны быть собраны одной и той же версией одного и того же компилятора. Само ядро изначально написано для компиляции посредством gcc – основного компилятора С в GNU‑системах, поэтому он же должен использоваться и при компиляции модулей для него.

## Компиляция драйвера

Сначала необходимо применить исправление к ядру и перекомпилировать его. Это делается командами:

# cd /usr/src/linux

# patch – p0 < memmon.patch

# make

# make install INSTALL\_PATH=/boot modules\_install

# lilo

# reboot

После этого возможна сама сборка драйвера. Для этого из папки с его исходниками надо выполнить команду $ make.

## Работа с драйвером

Для загрузки драйвера необходимо выполнить команду

# insmod procmon.ko

Можно указать размер буфера событий:

# insmod procmon.ko buflen=65536

Добавить процесс с PID = 3000 к отслеживаемым:

$ echo 3000 > /proc/memmon/watch-pids

Удалить его:

$ echo -3000 > /proc/memmon/watch-pids

Добавить процесс с именем test:

$ echo `pgrep test` > /proc/memmon/watch-pids

Удалить все процессы:

$ echo 0 > /proc/memmon/watch-pids

Просмотреть буфер событий:

$ cat /proc/memmon/events

Выгрузка драйвера делается командой

# rmmod procmon.ko

(возможно, только если файлы драйвера никем не открыты)

# Экспериментальный раздел

С целью изучения особенностей выделения памяти в Linux был проведен ряд экспериментов с выделением памяти, ее чтением / записью и освобождением.

1) Запрашиваем небольшое количество (3–4) страниц памяти, обращаемся ко всей памяти в этом диапазоне, затем освобождаем ее.

Журнал событий:

29305: anon page @b7ee1fa0 (R)

29305: fsync(0)

29305: anon page @b7f22d4e (R) – страничные отказы в сегменте кода libc.so

29305: anon page @b7ee2000 (R)

29305: anon page @b7e85180 (R)

29305: anon page @b7e86330 (R)

29305: anon page @b7f1f680 (R)

29305: anon page @b7ef6470 (R)

29305: anon page @b7e83140 (R)

29305: anon page @b7e82370 (R)

29305: anon page @b7e87de0 (R)

29305: brk(00000000)

29305: brk -> 134520832 (0804a000)

29305: brk(0806e000)

29305: brk -> 134668288 (0806e000) – malloc выделил 144 кб

29305: anon page @0804a004 (W) – здесь malloc заносит метки в начало и конец выделенного региона

29305: anon page @0804d00c (W)

29305: fsync(1)

29305: anon page @0804b000 (R) – сбои при обращении к страницам из цикла

29305: anon page @0804c000 (R)

29305: fsync(2)

29305: brk(0806b000) – free возвращает часть памяти

29305: brk -> 134656000 (0806b000)

29305: fsync(0) – `дальнейшие циклы уже не выделяют памяти

29305: fsync(1)

29305: fsync(2)

В приведенном примере видно, что при выделении 12К памяти *malloc()* выделяет изначально гораздо бОльший объем (144К), однако реально эти страницы из физической памяти не выделяеются, и при обращениях к ним происходят страничные отказы. В первую и последнюю страницу выделенной секции *malloc* заносит свои метки (сбой происходит на операции записи).

2) Выделяем подряд блоки по 4 страницы (не освобождая), обращаясь ко всем страницам:

21049: brk(00000000)

21049: brk -> (0804a000)

21049: brk(0806e000)

21049: brk -> (0806e000) – выделение первого блока, malloc выделяет 144 кб

21049: anon page @0804a004 (W)

21049: anon page @0804d00c (W) – запись меток

21049: fsync(1)

21049: anon page @0804b000 (R)

21049: anon page @0804c000 (R) – обращение к выделенной области

21049: fsync(2)

21049: fsync(0)

21049: anon page @08050014 (W) – выделение следующих 12 кб (запись метки)

21049: fsync(1)

21049: anon page @0804e000 (R)

21049: anon page @0804f000 (R) – обращения к выделенной области

21049: fsync(2)

…

21049: brk(0808f000) – очередной вызов malloc(), расширяем сегмент данных

21049: brk -> 134803456 (0808f000)

21049: anon page @0806e064 (W)

21049: fsync(1)

21049: fsync -> -22 (ffffffea)

21049: anon page @0806c000 (R)

21049: anon page @0806d000 (R)

…

Таким образом, видно, что *malloc* выделяет память блоками по 128 с небольшим кб при помощи вызова *brk().* Рассмотрим, что происходит при увеличении размера запроса.

3) Запрашиваем последовательно 4, 8, 12, 16К, и т.д., обращаясь в цикле ко всем страницам. При этом, как только размер выделения превышает 128К, *malloc* выделяет память уже не из области сегмента данных программы (расширяемого вызововм *brk()*), а при помощи *mmap()*, после чего *free()* его освобождает последующим *munmap()*:

789: mmap (00000000, 139264, rw-, PRIVATE | ANON, fd -1, @f019a000)

789: mmap -> -1210519552 (b7d8f000)

789: anon page @b7d8f004 (W)

789: fsync(1)

789: anon page @b7d90000 (R)

…

789: anon page @b7db0000 (R)

789: fsync(2)

789: munmap (b7d8f000, 139264)

789: munmap -> 0 (00000000)

При этом выделяется немного боьше памяти, чем было запрошено, и каждый раз она вся освобождается (т.е. следующие запросы опять приведут к выделениям). Опять-таки, страницы изначально возвращаются невыделенными.

3) Третий пример запрашивает память куда большими блоками, увеличивающимися по 100 Мб. При отключенном overcommit'e память быстро заканчивается, и очередной mmap возвращает **ENOMEM**:

1536: mmap (00000000, 629149696, rw-, PRIVATE | ANON)

1536: mmap -> -12 (fffffff4)

1536: anon page @b7e06de0 (R)

1536: brk(00000000)

1536: brk -> 134520832 (0804a000)

1536: brk(2d86b000)

1536: brk -> 134520832 (0804a000)

1536: mmap (00000000, 629280768, rw-, PRIVATE | ANON)

1536: mmap -> -12 (fffffff4)

1536: mmap (00000000, 2097152, rw-, PRIVATE | ANON)

1536: mmap -> -1212555264 (b7b9e000)

1536: munmap (b7b9e000, 401408)

1536: munmap -> 0 (00000000)

1536: munmap (b7d00000, 647168)

1536: munmap -> 0 (00000000)

1536: anon page @b7c00008 (W)

1536: mmap (00000000, 629149696, rw-, PRIVATE | ANON)

1536: mmap -> -12 (fffffff4)

Библиотека **libc** при этом пытается сначала выделить ту же память при помощи вызова *brk()*, затем снова при помощи вызова *mmap(),* но уже несколько меньший объем, однако и эти вызовы проходят неудачно.

4) Включим overcommit. Теперь, пока программа не обращается ко всем страницам из выделенного региона, а лишь к некоторым (не расходуя при этом всю физическую память), выполнение проходит нормально, и вызов *mmap()* успешно выделяет блоки памяти, значительно превыщающие объем свободной оперативной памяти (порядка 600 M):

2515: mmap (00000000, 996151296, rw-, PRIVATE | ANON)

2515: mmap -> 2089299968 (7c883000)

2515: anon page @7c883004 (W)

2515: fsync(1)

2515: anon page @8b603008 (R)

2515: anon page @9a383008 (R)

2515: anon page @a9103008 (R)

2515: fsync(2)

2515: munmap (7c883000, 996151296)

2515: munmap -> 0 (00000000)

В данном примере вызов *mmap()* выделяет 900М, из которых мы обращаемся лишь к четырем страницам.

5) При попытке попытаться обратиться ко всем страницам из выделенного региона, очень скоро память исчерпывается и возникает ситуация, называемая **OOM** – Out Of Memory. В этом случае ядро вызывает функцию *oom\_kill()*, которая выбирает процесс-жертву для уничтожения, основываясь на расходах памяти, времени работы, приоритете и т.п., и убивает выбранный процесс, с целью освободить память. При этом в журнал отладочных сообщений ядра выдается уведомление о том, что сработал *oom\_kill*:

kernel: kwin invoked oom-killer: gfp\_mask=0x201d2, order=0, oomkillad

6) Включим теперь файл подкачки. В случае, если объем выделяемого региона превышает размер доступной физической памяти, но меньше суммарного размера ее и файла подкачки, страницы из региона сначала выделяются по мере обращения к ним, затем старые начинают выгружаться в swap‑файл, после чего на втором цикле считывания происходит их подкачка оттуда:

19225: anon page @b7418bb0 (R)

…

19225: anon page @b7602893 (R)

19225: swapfile page @0ae1507c (R)

19225: swapfile page @0d8cb0e6 (R)

…

19225: swapfile page @0af146b0 (R)

7) Системные вызовы *mlock()/mlockall()* делают невыгружаемой определенную страницу виртуальной памяти процесса или все его страницы – текущие и / или выделенные в будущем, в зависимости от передаваемых в вызов *mlockall()* флагов.

Сделаем в начале выполнения программы невыгружемыми все страницы, выделяемые в будущем, после чего выделяем блоки по 12 кб (не освобождая):

13749: brk(00000000)

13749: brk -> 134520832 (0804a000)

13749: brk(0806e000)

13749: anon page @0804a000 (W)

…

13749: anon page @0806c000 (W)

13749: anon page @0806d000 (W)

13749: brk -> 134668288 (0806e000)

…

В данном случае ядро сразу после выделения виртуальной памяти обращается ко всем ее страницам с целью их загрузки в физическую память.

Таким образом, можно сделать следующие выводы:

1. При выделении памяти ядро не выделяет сразу все физические страницы (если в *mmap()* не был передан флаг **MAP POPULATE** или страницы процесса не были заблокированы в физической памяти вызовами *mlock[all]() –* в этих случаях страницы всего региона подгружаются сразу)
2. Для выделения небольших объемов памяти библиотека **libc** расширяет сегмент данных программы вызовом *brk()*, для запросов, бОльших 128К – использует *mmap()*.

# Overcommit является довольно мощным средством, позволяющим выделять гораздо больше виртуальной памяти, чем доступно на самом деле, при условии использования лишь доступного ее объема (данная возможность может быть полезна для различных научно-инженерных приоложений). Однако, в случае, если реально затребованная память превысит суммарноый объем доступной и файла подкачки, возникнет критическая ситуация нехватки памяти. Заключение

В рамках данной работы были исследованы вопросы, связанные с разработкой драйверов под OS Linux, работой ядра Linux с виртуальной памятью и перехватом системных вызовов.

Драйвер может быть загружен и выгружен без перезагрузки системы. Он не влияет на работу других устройств и всей системы в целом, и не приводит к ощутимым задержкам в работе.

Было произведено исследование механизма выделения памяти в ядре Linux и библиотеке **libс**, исследована технология overcommit.

Возможность отследить операции процессов в памятью зачастую может быть весьма полезной при отладке программ, мониторинге или настройке системы (например, для подбора оптимальных параметров сервера, расходующего большой объем памяти).

# Список использованной литературы

1. Jonathan Corbet. Linux Device Drivers, 3rd Edition.
2. Роберт Лав. Разработка ядра Linux, 2-е издание.
3. Peter Salzman. The Linux Kernel Module Programming Guide, 3rd Edition.
4. Клаудия Родригес. Азбука ядра Linux.
5. Исходники ядра и документация к ним.

# Приложения

## Код драйвера

### mmon.c

/\*

\* Main module and entry point of memmon.

\*/

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/proc\_fs.h>

#include «common.h»

#include «watch-pids.h»

#include «mm-fault.h»

#include «events.h»

#include «syscalls.h»

/\*\*\* Internal data \*\*\*/

/\*

\* procfs directory entry

\*/

struct proc\_dir\_entry \*procdir = NULL;

/\*

\* Init entry point

\*/

static int \_\_init init(void)

{

int ret = – EBUSY;

procdir = proc\_mkdir (PROCDIR, NULL);

if (! procdir)

goto out;

if (! init\_watch\_pids())

goto out\_procdir;

if (! init\_events())

goto out\_watch\_pids;

if (! capture\_syscalls())

goto out\_events;

capture\_mmfault();

return 0;

out\_events:

fini\_events();

out\_watch\_pids:

fini\_watch\_pids();

out\_procdir:

remove\_proc\_entry (PROCDIR, NULL);

out:

return ret;

}

module\_init(init);

/\*

\* Exit point

\*/

static void \_\_exit exit(void)

{

release\_mmfault();

restore\_syscalls();

fini\_events();

fini\_watch\_pids();

remove\_proc\_entry (PROCDIR, NULL);

}

module\_exit(exit);

/\*\*\* Module info \*\*\*/

MODULE\_LICENSE («GPL»);

MODULE\_AUTHOR («Ivan Korotkov»);

MODULE\_DESCRIPTION («Linux Virtual Memory Monitor»);

### events.h

/\*

\* Events ringbuffer.

\*/

#ifndef MEMMON\_EVENTS\_H

#define MEMMON\_EVENTS\_H

/\* Filename in procfs directory \*/

#define EVENTS\_ENTRY «events»

/\* Types of events \*/

enum memmon\_event\_type

{

NOTUSED = 0, /\* to prevent treating zero-filled region as event struct \*/

MMAP2,

MUNMAP,

MREMAP,

MLOCK,

MUNLOCK,

MLOCKALL,

MUNLOCKALL,

BRK,

FSYNC,

ANON\_PF,

SWAP\_PF,

FILE\_PF,

SYSCALLRET

};

/\*

\* Struct describing each event

\*/

struct memmon\_event

{

/\* Type \*/

enum memmon\_event\_type type;

/\* Caller PID \*/

pid\_t pid;

/\* Per-type data \*/

union

{

struct

{

void \_\_user \*start;

size\_t len;

} munmap;

struct

{

void \_\_user \*start;

size\_t len;

unsigned long prot, flags;

unsigned long fd, off;

} mmap2;

struct

{

void \_\_user \*start[2];

size\_t len[2];

unsigned flags;

} mremap;

struct

{

void \_\_user \*start;

size\_t len;

} mlock, munlock;

struct

{

unsigned long flags;

} mlockall;

struct

{

void \_\_user \*addr;

} brk;

struct

{

int fd;

} fsync;

struct

{

void \_\_user \*addr;

int write;

} pagefault;

struct

{

char \*callname;

long ret;

} callret;

};

};

#define NEVENTS (EVENTS\_BUFLEN/sizeof (struct memmon\_event))

/\*

\* Initializes event ringbuffer & creates /proc entry

\*/

int init\_events(void);

/\*

\* Destroys ringbuffer & removes /proc entry

\*/

void fini\_events(void);

/\*

\* Adds events to ringbuffer tail

\*/

void put\_event (const struct memmon\_event \*ev);

#endif // MEMMON\_EVENTS\_H

### events.c

/\*

\* Events ringbuffer.

\*/

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/seq\_file.h>

#include <linux/proc\_fs.h>

#include <linux/poll.h>

#include <linux/mman.h>

#include «common.h»

#include «events.h»

/\*\*\* Forward declarations \*\*\*/

static int events\_open (struct inode \*i, struct file \*filp);

static unsigned events\_poll (struct file \*filp, struct poll\_table\_struct \*pt);

static void \*events\_seqstart (struct seq\_file \*m, loff\_t \*pos);

static void events\_seqstop (struct seq\_file \*m, void \*p);

static void \*events\_seqnext (struct seq\_file \*m, void \*p, loff\_t \*pos);

static int events\_seqprint (struct seq\_file \*m, void \*p);

/\* Default ringbuffer size \*/

#define EVENTS\_BUFLEN (32\*1024)

/\* Min ringbuffer size \*/

#define MIN\_EVENTS\_BUFLEN (8\*1024)

/\*\*\* Module parameters \*\*\*/

/\* Actual ringbuffer size \*/

static int buflen = EVENTS\_BUFLEN;

module\_param (buflen, int, 0444);

/\*\*\* File operations \*\*\*/

static const struct file\_operations events\_fops =

{

owner = THIS\_MODULE,

open = events\_open,

read = seq\_read,

release = seq\_release,

poll = events\_poll

};

static const struct seq\_operations events\_seqop =

{

start = events\_seqstart,

stop = events\_seqstop,

next = events\_seqnext,

show = events\_seqprint

};

/\*\*\* Internal data \*\*\*/

/\* Ringbuffer \*/

static struct memmon\_event \*events;

/\* Last entry left in ringbuffer

\* (where 1st read should begin) \*/

static int ev\_start;

/\* Current write position \*/

static int ev\_end;

/\* Whether there was ringbuffer overflow \*/

static int ev\_ovf = 0;

DECLARE\_WAIT\_QUEUE\_HEAD (ev\_waitq);

spinlock\_t ev\_lock = SPIN\_LOCK\_UNLOCKED;

/\* Damn seq\_file doesn't update file pos when we return NULL iterator,

\* so we first return this one and then NULL on next seqnext() call \*/

static void \*dummy\_ptr = &dummy\_ptr;

/\*\*\* Entry points \*\*\*/

/\*

\* open() handler

\*/

static int events\_open (struct inode \*i, struct file \*filp)

{

int ret;

/\*

\* Ringbuffer is not seekable

\*/

nonseekable\_open (i, filp);

/\*

\* Open seq\_file and set its initial pos

\*/

ret = seq\_open (filp, &events\_seqop);

if (! ret)

{

struct seq\_file \*m = filp->private\_data;

m->private = filp;

m->index = ev\_start;

}

return ret;

}

/\*

\* poll/epoll() handler

\*/

static unsigned events\_poll (struct file \*filp, struct poll\_table\_struct \*pt)

{

struct seq\_file \*m = filp->private\_data;

unsigned mask = 0;

spin\_lock (&ev\_lock);

poll\_wait (filp, &ev\_waitq, pt);

/\*

\* The only poll event we can trigger is normal read event

\*/

if (m->index!= ev\_end)

mask = POLLIN | POLLRDNORM;

spin\_unlock (&ev\_lock);

return mask;

}

/\*

\* Called by seq\_file within read() request

\*/

static void \*events\_seqstart (struct seq\_file \*m, loff\_t \*pos)

{

struct file \*filp = m->private;

spin\_lock (&ev\_lock);

/\*

\* Wait for data become available

\*/

while (\*pos == (loff\_t) ev\_end)

{

void \*err = NULL;

/\* Can't schedule while atomic \*/

spin\_unlock (&ev\_lock);

if (filp->f\_flags & O\_NONBLOCK)

err = ERR\_PTR(-EAGAIN);

else if (wait\_event\_interruptible (ev\_waitq, \*pos!= (loff\_t) ev\_end))

err = ERR\_PTR(-ERESTARTSYS);

/\*

\* There IS a slim chance, that we loose waiting condition

\* between awakening and acquiring spinlock – hence while() loop

\*/

spin\_lock (&ev\_lock);

if (err)

return err;

}

return events + \*pos;

}

/\*

\* Finish read() request

\*/

static void events\_seqstop (struct seq\_file \*m, void \*p)

{

spin\_unlock (&ev\_lock);

}

/\*

\* Iterate to next event

\*/

static void \*events\_seqnext (struct seq\_file \*m, void \*p, loff\_t \*pos)

{

struct memmon\_event \*ev;

/\* Dummy iterator – time to exit \*/

if (p == dummy\_ptr)

return NULL;

++\*pos;

ev = events + \*pos;

/\* Overflow \*/

if (ev – events > NEVENTS)

\*pos = 0;

/\*

\* We reached end. Decrement file pos ('coz it will be incremented then back)

\* and return dummy iterator (otherwise file pos won't be updated at all)

\*/

if (\*pos == (loff\_t) ev\_end)

{

–\*pos;

return dummy\_ptr;

}

return events + \*pos;

}

/\*

\* Actually prints current iterator to read buffer

\*/

static int events\_seqprint (struct seq\_file \*m, void \*p)

{

struct memmon\_event \*ev = p;

if (ev == dummy\_ptr)

return 0;

seq\_printf (m, «%d:», ev->pid);

switch (ev->type)

{

case MMAP2:

seq\_printf (m, «mmap (%p,%u,», ev->mmap2.start, ev->mmap2.len);

if (ev->mmap2.prot & PROT\_READ)

seq\_puts (m, «r»);

else

seq\_puts (m, «–»);

if (ev->mmap2.prot & PROT\_WRITE)

seq\_puts (m, «w»);

else

seq\_puts (m, «–»);

if (ev->mmap2.prot & PROT\_EXEC)

seq\_puts (m, «x,»);

else

seq\_puts (m, «-,»);

if (ev->mmap2.flags & MAP\_SHARED)

seq\_puts (m, «SHARED»);

else if (ev->mmap2.flags & MAP\_PRIVATE)

seq\_puts (m, «PRIVATE»);

if (ev->mmap2.flags & MAP\_LOCKED)

seq\_puts (m, «| LOCKED»);

if (ev->mmap2.flags & MAP\_ANON)

seq\_puts (m, «| ANON»);

if (ev->mmap2.flags & MAP\_POPULATE)

seq\_puts (m, «| READAHEAD»);

if (ev->mmap2.flags & MAP\_ANON)

seq\_puts (m,»)\n»);

else

seq\_printf (m,», fd% ld, @%p)\n», (long) ev->mmap2.fd,

(void \*) ev->mmap2.off);

break;

case MUNMAP:

seq\_printf (m, «munmap (%p,%d)\n», ev->munmap.start, ev->munmap.len);

break;

case MREMAP:

seq\_printf (m, «mremap (%p,%d ->%p,%d)\n», ev->mremap.start[0], ev->mremap.len[0],

ev->mremap.start[1], ev->mremap.len[1]);

break;

case MLOCK:

seq\_printf (m, «mlock (%p,%d)\n», ev->mlock.start, ev->mlock.len);

break;

case MUNLOCK:

seq\_printf (m, «munlock (%p,%d)\n», ev->munlock.start, ev->munlock.len);

break;

case MLOCKALL:

seq\_puts (m, «mlockall(»);

if (ev->mlockall.flags & MCL\_CURRENT)

{

seq\_puts (m, «CURRENT»);

if (ev->mlockall.flags & MCL\_FUTURE)

seq\_puts (m, «| FUTURE»);

}

else if (ev->mlockall.flags & MCL\_FUTURE)

seq\_puts (m, «FUTURE»);

seq\_puts (m,»)\n»);

break;

case MUNLOCKALL:

seq\_puts (m, «munlockall()\n»);

break;

case BRK:

seq\_printf (m, «brk(%p)\n», ev->brk.addr);

break;

case FSYNC:

seq\_printf (m, «fsync(%d)\n», ev->fsync.fd);

break;

case ANON\_PF:

seq\_printf (m, «anon page @%p (%s)\n», ev->pagefault.addr,

ev->pagefault.write? «W»: «R»);

break;

case SWAP\_PF:

seq\_printf (m, «swapfile page @%p (%s)\n», ev->pagefault.addr,

ev->pagefault.write? «W»: «R»);

break;

case FILE\_PF:

seq\_printf (m, «shared file page @%p (%s)\n», ev->pagefault.addr,

ev->pagefault.write? «W»: «R»);

break;

case SYSCALLRET:

seq\_printf (m, «%s ->%ld (%p)\n», ev->callret.callname, ev->callret.ret,

(void \*) ev->callret.ret);

break;

default:

printk («memmon: Unexpected event% d\n», ev->type);

return 1;

}

return 0;

}

/\*\*\* Exported entries \*\*\*/

/\*

\* Initializes event ringbuffer & creates /proc entry

\*/

int init\_events(void)

{

struct proc\_dir\_entry \*entry;

buflen = max (buflen, MIN\_EVENTS\_BUFLEN);

events = kzalloc (buflen, GFP\_KERNEL);

if (! events)

{

printk («memmon: Event ringbuffer too big!\n»);

return 0;

}

ev\_start = ev\_end = 0;

entry = create\_proc\_entry (EVENTS\_ENTRY, 0444, procdir);

if (entry)

entry->proc\_fops = &events\_fops;

else

{

kfree(events);

return 0;

}

return 1;

}

/\*

\* Destroys ringbuffer & removes /proc entry

\*/

void fini\_events(void)

{

remove\_proc\_entry (EVENTS\_ENTRY, procdir);

kfree(events);

}

/\*

\* Adds events to ringbuffer tail

\*/

void put\_event (const struct memmon\_event \*ev)

{

spin\_lock (&ev\_lock);

events [ev\_end] = \*ev;

/\* Overflow \*/

if (++ev\_end > NEVENTS)

{

ev\_start = ev\_end = 0;

ev\_ovf = 1;

}

/\*

\* If overflow happened at least once, ev\_start must be next to ev\_end.

\* Otherwise, it remains zero.

\*/

if (ev\_ovf && ++ev\_start > NEVENTS)

ev\_start = 0;

spin\_unlock (&ev\_lock);

wake\_up\_interruptible\_sync (&ev\_waitq);

}

### watch-pids.h

/\*

\* Selection of PIDs to watch for.

\*/

#ifndef MEMMON\_WATCH\_PIDS\_H

#define MEMMON\_WATCH\_PIDS\_H

/\*

\* Checks whether PID @pid is present in PID set

\* Returns 1 if present

\*/

int pid\_present (pid\_t pid);

/\*

\* Initializes PID set & creates /proc entry

\*/

int init\_watch\_pids(void);

/\*

\* Destroys PID set & removes /proc entry

\*/

void fini\_watch\_pids(void);

#endif // MEMMON\_WATCH\_PIDS\_H

### watch-pids.c

/\*

\* Selection of PIDs to watch for.

\*/

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/proc\_fs.h>

#include <linux/bitmap.h>

#include <asm/uaccess.h>

#include <asm/bitops.h>

#include «common.h»

#include «watch-pids.h»

/\*\*\* Forward declarations \*\*\*/

static int watch\_pids\_open (struct inode \*i, struct file \*filp);

static int watch\_pids\_release (struct inode \*i, struct file \*filp);

static ssize\_t watch\_pids\_read (struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*off);

static ssize\_t watch\_pids\_write (struct file \*filp, const char \_\_user \*buf,

size\_t count, loff\_t \*offp);

/\*\*\* Internal data \*\*\*/

/\* Filename in procfs directory \*/

#define WATCHPID\_ENTRY «watch-pids»

#define PID\_COUNT PID\_MAX\_DEFAULT + 1

/\* PIDs are stored in one single bitmap for 8192 entries

\* This is VERY RARELY unacceptable \*/

static DECLARE\_BITMAP (watched\_pids, PID\_COUNT);

/\*\*\* File operations \*\*\*/

static const struct file\_operations watch\_pids\_fops =

{

owner = THIS\_MODULE,

open = watch\_pids\_open,

read = watch\_pids\_read,

write = watch\_pids\_write,

release = watch\_pids\_release

};

/\*\*\* Entry points \*\*\*/

/\*

\* open() handler

\*/

static int watch\_pids\_open (struct inode \*i, struct file \*filp)

{

try\_module\_get (THIS\_MODULE);

/\*

\* If file opened for read, print PID set to internal buffer

\*/

if (filp->f\_mode & FMODE\_READ)

{

const int FDATA\_SIZ = 32\*1024;

char \*fdata;

int len;

/\*

\* Disallow mixed RW-access

\*/

if (filp->f\_mode & FMODE\_WRITE)

return – EINVAL;

fdata = kzalloc (FDATA\_SIZ, GFP\_KERNEL);

len = bitmap\_scnlistprintf (fdata, FDATA\_SIZ – 1,

watched\_pids, PID\_COUNT);

/\* Append \n \*/

if (len)

{

fdata [len++] = '\n';

fdata[len] = 0;

}

filp->private\_data = fdata;

}

return 0;

}

/\*

\* close() handler

\*/

static int watch\_pids\_release (struct inode \*i, struct file \*filp)

{

module\_put (THIS\_MODULE);

if (filp->private\_data)

kfree (filp->private\_data);

return 0;

}

/\*

\* read() handler – simply return chunk of data from

\* previously allocated and formatted buffer

\*/

static ssize\_t watch\_pids\_read (struct file \*filp, char \_\_user \*buf,

size\_t count, loff\_t \*offp)

{

size\_t len = strlen (filp->private\_data);

char \*fdata = filp->private\_data;

if (\*offp >= len)

return 0;

len = min (count, len – (size\_t) (\*offp));

if (copy\_to\_user (buf, fdata + (\*offp), len))

return – EFAULT;

\*offp += len;

return len;

}

/\*

\* write() handler

\* Buffer must hold ASCII representation of single integer

\* if positive, it's value is PID to add to set

\* if negative, it's absolute value is PID to remove from set

\* if zero, PID set is cleared

\*/

static ssize\_t watch\_pids\_write (struct file \*filp, const char \_\_user \*buf,

size\_t count, loff\_t \*offp)

{

const size\_t maxlen = 4096;

size\_t len;

pid\_t new\_pid;

char \*data;

ssize\_t res = – ENOMEM;

/\* copy up to one page to our buffer \*/

len = min (maxlen, count);

data = kzalloc (len, GFP\_KERNEL);

if (unlikely(! data))

return – ENOMEM;

if (copy\_from\_user (data, buf, len))

res = – EFAULT;

else if ((sscanf (data, «%d», &new\_pid) == 1) &&

new\_pid <= PID\_COUNT && new\_pid >= – PID\_COUNT)

{

if (new\_pid > 0)

set\_bit (new\_pid, watched\_pids);

else if (new\_pid < 0)

clear\_bit (-new\_pid, watched\_pids);

else

bitmap\_zero (watched\_pids, PID\_COUNT);

res = len;

}

else

/\* buffer doesn't represent a number in PID range \*/

res = – EIO;

kfree(data);

return res;

}

/\*\*\* Exported entries \*\*\*/

/\*

\* Checks whether PID @pid is present in PID set

\* Returns 1 if present

\*/

int pid\_present (pid\_t pid)

{

if (pid > PID\_COUNT || pid <= 0)

return 0;

return test\_bit (pid, watched\_pids)? 1: 0;

}

/\*

\* Initializes PID set & creates /proc entry

\*/

int init\_watch\_pids(void)

{

struct proc\_dir\_entry \*entry;

entry = create\_proc\_entry (WATCHPID\_ENTRY, 0666, procdir);

if (entry)

entry->proc\_fops = &watch\_pids\_fops;

else

return 0;

bitmap\_zero (watched\_pids, PID\_COUNT);

return 1;

}

/\*

\* Destroys PID set & removes /proc entry

\*/

void fini\_watch\_pids(void)

{

remove\_proc\_entry (WATCHPID\_ENTRY, procdir);

}

### syscalls.h

/\*

\* Syscall capture facility.

\*/

#ifndef MEMMON\_SYSCALLS\_H

#define MEMMON\_SYSCALLS\_H

/\*

\* Installs handlers.

\*/

int capture\_syscalls(void);

/\*

\* Uninstalls handlers

\*/

void restore\_syscalls(void);

#endif //MEMMON\_SYSCALLS\_H

### syscalls.c

/\*

\* Syscall capture facility.

\*/

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/proc\_fs.h>

#include «common.h»

#include «syscalls.h»

#include «events.h»

#include «watch-pids.h»

/\*\*\* Syscalls \*\*\*/

/\*

\* They just put an appropriate event into ringbuffer

\*/

asmlinkage void sys2\_mmap2 (void \_\_user \*start, size\_t length,

unsigned long prot, unsigned long flags,

unsigned long fd, unsigned long pgoff)

{

struct memmon\_event ev = {.type = MMAP2.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.mmap2.start = start;

ev.mmap2.len = length;

ev.mmap2.prot = prot >> 3;

ev.mmap2.flags = flags;

ev.mmap2.fd = fd;

ev.mmap2.off = pgoff;

put\_event(&ev);

}

asmlinkage void sys2\_mmap2\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «mmap»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_munmap (void \_\_user \*start, size\_t length)

{

struct memmon\_event ev = {.type = MUNMAP.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.munmap.start = start;

ev.munmap.len = length;

put\_event(&ev);

}

asmlinkage void sys2\_munmap\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «munmap»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_mremap (void \_\_user \*addr1, size\_t length1,

unsigned long length2, unsigned long flags,

void \_\_user \*addr2)

{

struct memmon\_event ev = {.type = MREMAP.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.mremap.start[0] = addr1;

ev.mremap.start[1] = addr2;

ev.mremap.len[0] = length1;

ev.mremap.len[1] = length2;

ev.mremap.flags = flags;

put\_event(&ev);

}

asmlinkage void sys2\_mremap\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «mremap»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_mlock (void \_\_user \*start, size\_t length)

{

struct memmon\_event ev = {.type = MLOCK.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.mlock.start = start;

ev.mlock.len = length;

put\_event(&ev);

}

asmlinkage void sys2\_mlock\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «mlock»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_munlock (void \_\_user \*start, size\_t length)

{

struct memmon\_event ev = {.type = MUNLOCK.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.munlock.start = start;

ev.munlock.len = length;

put\_event(&ev);

}

asmlinkage void sys2\_munlock\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «munlock»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_mlockall (unsigned long flags)

{

struct memmon\_event ev = {.type = MLOCKALL.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.mlockall.flags = flags;

put\_event(&ev);

}

asmlinkage void sys2\_mlockall\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «mlockall»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_munlockall()

{

struct memmon\_event ev = {.type = MUNLOCKALL.pid = current->pid};

if (! pid\_present (ev.pid)) return;

put\_event(&ev);

}

asmlinkage void sys2\_munlockall\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «munlockall»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_brk (void \_\_user \*start)

{

struct memmon\_event ev = {.type = BRK.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.brk.addr = start;

put\_event(&ev);

}

asmlinkage void sys2\_brk\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «brk»;

ev.callret.ret = ret;

put\_event(&ev);

}

asmlinkage void sys2\_fsync (int fd)

{

struct memmon\_event ev = {.type = FSYNC.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.fsync.fd = fd;

put\_event(&ev);

}

asmlinkage void sys2\_fsync\_exit (long ret)

{

struct memmon\_event ev = {.type = SYSCALLRET.pid = current->pid};

if (! pid\_present (ev.pid)) return;

ev.callret.callname = «fsync»;

ev.callret.ret = ret;

put\_event(&ev);

}

/\*\*\* Handler tables \*\*\*/

/\* Kernel syscall table \*/

extern void \*sys\_call\_table[];

/\* Our table w/saved offsets \*/

void \*old\_sys\_call [NR\_syscalls];

/\* Our pre-call handlers \*/

void \*sys\_call\_trap [NR\_syscalls];

/\* Our post-call handlers \*/

void \*sys\_call\_exit [NR\_syscalls];

/\*

\* Struct describind our handler

\*/

struct syscall\_handler

{

/\* Syscall nr \*/

int nr;

/\* Pre-call & post-call handler \*/

void \*hand1, \*hand2;

};

#define SYSCALL\_HANDLER(name) {\_\_NR\_##name, sys2\_##name, sys2\_##name##\_exit}

#define SYSCALL\_HANDLERS\_END() {0, 0, 0}

/\*

\* Main handler table

\* Each SYSCALL\_HANDLER(name) entry installs handlers

\* «sys2\_name/sys2\_name\_exit for sys\_name call.

\*/

struct syscall\_handler syscalls[] =

{

SYSCALL\_HANDLER(mmap2),

SYSCALL\_HANDLER(munmap),

SYSCALL\_HANDLER(mremap),

SYSCALL\_HANDLER(mlock),

SYSCALL\_HANDLER(munlock),

SYSCALL\_HANDLER(mlockall),

SYSCALL\_HANDLER(munlockall),

SYSCALL\_HANDLER(brk),

SYSCALL\_HANDLER(fsync),

SYSCALL\_HANDLERS\_END()

};

/\* Located in syscall-entry.S \*/

void syscalls\_entry(void);

/\*\*\* Exported entries \*\*\*/

/\*

\* Installs handlers.

\*/

int capture\_syscalls(void)

{

int i;

for (i = 0; syscalls[i].hand1; ++i)

{

int nr = syscalls[i].nr;

sys\_call\_trap[nr] = syscalls[i].hand1;

sys\_call\_exit[nr] = syscalls[i].hand2;

old\_sys\_call[nr] = sys\_call\_table[nr];

sys\_call\_table[nr] = syscalls\_entry;

}

return 1;

}

/\*

\* Uninstalls handlers

\*/

void restore\_syscalls(void)

{

int i;

for (i = 0; syscalls[i].hand1; ++i)

{

int nr = syscalls[i].nr;

sys\_call\_table[nr] = old\_sys\_call[nr];

}

}

### syscalls-entry.S

/\*

\* Syscall entry/exit capture

\*/

#include «offsets.h»

/\* Entry handler table \*/

extern sys\_call\_trap

/\* Exit handler table \*/

extern sys\_call\_exit

/\* Global entry for our syscalls \*/

syscalls\_entry:

/\* Save registers in order syscall handlers expect 'em \*/

pushl %eax

pushl %ebp

pushl %edi

pushl %esi

pushl %edx

pushl %ecx

pushl %ebx

/\* Save eax \*/

movl %eax, TI\_stk0 (%ebp)

/\* Call our handler \*/

call \*sys\_call\_trap (,%eax, 4)

/\* Fake return address \*/

movl 28 (%esp),%eax

movl %eax, TI\_stk0 + 4 (%ebp)

movl $sysreturn, 28 (%esp)

/\* Restore context \*/

popl %ebx

popl %ecx

popl %edx

popl %esi

popl %edi

popl %ebp

popl %eax

/\* Jump to default system handler \*/

jmpl \*old\_sys\_call (,%eax, 4)

sysreturn:

/\* Save registers \*/

pushal

/\* Pass new% eax to exit handler \*/

pushl %eax

/\* Restore original% eax \*/

movl TI\_stk0 (%ebp),%eax

/\* Call our exit handler \*/

call \*sys\_call\_exit (,%eax, 4)

/\* Restore context \*/

popl %eax

popal

/\* Jump back to syscall dispatcher entry \*/

jmpl \*TI\_stk0 + 4 (%ebp)

globl syscalls\_entry

### gen-offsets.c

//

#define \_\_KERNEL\_\_

/\* bugoga \*/

#include <linux/kernel.h>

#include <linux/autoconf.h>

#include <linux/thread\_info.h>

#include <stdio.h>

int main()

{

printf («#define TI\_stk0% d\n», offsetof (struct thread\_info, supervisor\_stack));

return 0;

}

### mm-fault.h

/\*

\* Pagefault interception.

\*/

#ifndef MEMMON\_MM\_FAULT\_H

#define MEMMON\_MM\_FAULT\_H

/\*

\* Install pagefault handler

\*/

void capture\_mmfault(void);

/\*

\* Uninstall handler

\*/

void release\_mmfault(void);

#endif // MEMMON\_MM\_FAULT\_H

### mm-fault.c

/\*

\* Pagefault interception.

\*/

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/mm.h>

#include «common.h»

#include «mm-fault.h»

#include «events.h»

#include «watch-pids.h»

/\*

\* Dirty kernel hack: PF hook that is called every time

\* some process PF's for some page that BELONGS to his VMA space.

\*/

extern void (\*mm\_handle\_fault\_hook) (struct mm\_struct \*mm, struct vm\_area\_struct \*vma,

void \_\_user \*address, pte\_t \*pte,

pmd\_t \*pmd, int write\_access);

/\*

\* Pagefault handler

\*/

void mm\_handle\_fault (struct mm\_struct \*mm, struct vm\_area\_struct \*vma,

void \_\_user \*address, pte\_t \*pte,

pmd\_t \*pmd, int write\_access)

{

struct memmon\_event ev = {.pid = current->pid};

pte\_t entry = \*pte;

/\*

\* If PF happened due to R/W or U/S access violation, ignore it

\*/

if (! pid\_present (current->pid) || pte\_present(entry))

return;

/\*

\* Faulted page is either backed by swapfile, some shared executable file

\* or no file yet at all (anonymous page)

\*/

if (pte\_none(entry))

ev.type = ANON\_PF;

else if (pte\_file(entry))

ev.type = FILE\_PF;

else

ev.type = SWAP\_PF;

ev.pagefault.addr = address;

ev.pagefault.write = write\_access;

put\_event(&ev);

}

/\*\*\* Exported entries \*\*\*/

/\*

\* Install pagefault handler

\*/

void capture\_mmfault(void)

{

mm\_handle\_fault\_hook = mm\_handle\_fault;

}

/\*

\* Uninstall handler

\*/

void release\_mmfault(void)

{

mm\_handle\_fault\_hook = NULL;

}

### common.h

/\*

\* Common defines and global data

\*/

#ifndef MEMMON\_COMMON\_H

#define MEMMON\_COMMON\_H

/\* procfs directory name \*/

#define PROCDIR «memmon»

/\*

\* procfs directory entry

\*/

extern struct proc\_dir\_entry \*procdir;

#endif // MEMMON\_COMMON\_H

### Makefile

#

ifneq ($(KERNELRELEASE),)

obj-m:= memmon.o

memmon-objs:= mmon.o events.o watch-pids.o syscalls.o syscalls-entry.o mm-fault.o

else

KERNELDIR?= /lib/modules/$(shell uname – r)/build

PWD:= $(shell pwd)

all: offsets.h modules

offsets.h: $(KERNELDIR)/include/asm/thread\_info.h

$(MAKE) gen-offsets

gen-offsets > offsets.h

$(RM) gen-offsets

clean modules:

$(MAKE) – C $(KERNELDIR) M=$(PWD) $(MAKECMDGOALS)

PHONY: modules.DEFAULT all

endif

## Испрвление для для ядра (2.6.20.1)

diff – arNC 3 linux‑2.6.20.1‑j/kernel/kallsyms.c linux‑2.6.20.1‑a/kernel/kallsyms.c

\*\*\* linux‑2.6.20.1‑j/kernel/kallsyms.c 2007–02–20 09:34:32.000000000 +0300

– linux‑2.6.20.1‑a/kernel/kallsyms.c 2007–05–26 22:27:23.000000000 +0400

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* 452,454 \*\*\*\*

– 452,460 –

\_\_initcall (kallsyms\_init);

EXPORT\_SYMBOL (\_\_print\_symbol);

+

+ /\* HACK \*/

+

+ extern void \*sys\_call\_table[];

+

+ EXPORT\_SYMBOL\_GPL (sys\_call\_table);

diff – arNC 3 linux‑2.6.20.1‑j/mm/memory.c linux‑2.6.20.1‑a/mm/memory.c

\*\*\* linux‑2.6.20.1‑j/mm/memory.c 2007–02–20 09:34:32.000000000 +0300

– linux‑2.6.20.1‑a/mm/memory.c 2007–05–28 22:08:41.000000000 +0400

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* 2369,2374 \*\*\*\*

– 2378,2390 –

return VM\_FAULT\_MAJOR;

}

+ /\* DIRTY HACK \*/

+ void (\*mm\_handle\_fault\_hook) (struct mm\_struct \*mm,

+ struct vm\_area\_struct \*vma, unsigned long address,

+ pte\_t \*pte, pmd\_t \*pmd, int write\_access) = NULL;

+

+ EXPORT\_SYMBOL\_GPL (mm\_handle\_fault\_hook);

+

/\*

\* These routines also need to handle stuff like marking pages dirty

\* and/or accessed for architectures that don't do it in hardware (most

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* 2390,2395 \*\*\*\*

– 2406,2414 –

pte\_t old\_entry;

spinlock\_t \*ptl;

+ if (mm\_handle\_fault\_hook)

+ mm\_handle\_fault\_hook (mm, vma, address, pte, pmd, write\_access);

+

old\_entry = entry = \*pte;

if (! pte\_present(entry)) {

if (pte\_none(entry)) {