УДК 681.3.066

Обнаружение тупиков на мьютексах в многопоточных приложениях

© В.С. Белоус, В.А. Крищенко, Н.Ю. Рязанова

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Статья посвящена разработке метода обнаружения тупиков при использовании в приложениях мьютексов, реализованных по стандарту POSIX. В рамках теории тупиков проанализирован и реализован способ получения информации о блокированных на мьютексах процессах. Показано, что обнаружение замкнутой цепи блокировок соответствует обнаружению цикла в графе запросов — распределений. Описан механизм включения в ядро средств перехвата функции ядра, которая управляет захватом и освобождением потоков на мьютексах. Предложен алгоритм обнаружения тупика на основе полученной информации.

***Ключевые слова*:** *тупик, взаимоисключение, мьютекс, функция-перехватчик, обнаружение тупиков.*

Введение. Современные приложения, как правило, являются многопоточными. Часто в процессе выполнения параллельные потоки обращаются к одним и тем же структурам данных и пытаются выполнить какие-то действия с этими структурами. Например, несколько потоков могут пытаться одновременно добавить в список, такой как стек или очередь, новый элемент. Добавление элемента в список выполняется с помощью последовательности действий показанной на рис. 1:

```
//Для первого потока p1:
 new_p1->next = old->next;
 old->next = new_p1;
```

```
//Для второго потока p2:
   new_p2->next = old->next;
   old->next = new_p2;
```

Рис. 1. Пример параллельного выполнения вставки элемента в очередь

В результате выполнения данной последовательности операций каждым потоком, новый элемент включается в цепочку элементов. Если данная последовательность операций в список будет прервана, то возможна потеря связи с одним из новых элементов.

Например, если после выполнения первой строки в коде первого потока будет прервано его выполнение, то элемент, добавляемый вторым потоком, может быть утерян (рис. 2).

Чтобы таких проблем не возникало, последовательность действий, выполняемая над структурами данных, должна быть неделимой. Необходимо предпринять специальные меры, чтобы такую последовательность действий нельзя было прервать. В ОС Linux для этого могут использоваться такие средства взаимоисключения, как

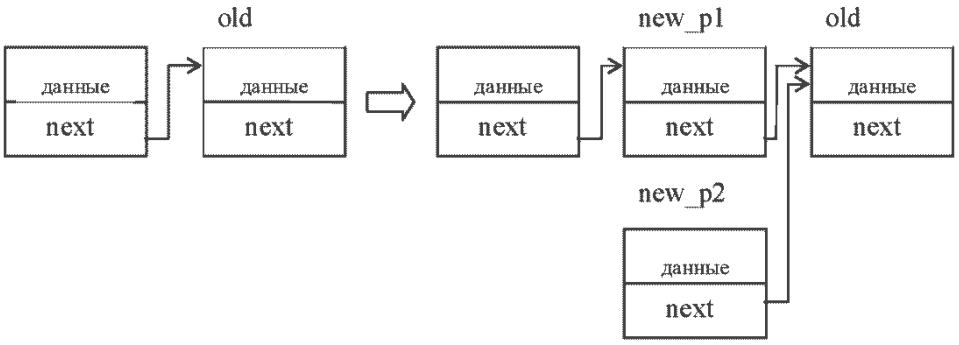

Рис. 2. Потеря связи при попытке включения элемента new_p2 в очередь

мьютексы, семафоры, спин-блокировки и т. п. Спин-блокировки реализуются через активное ожидание на процессоре — это значит, что процессор тратит время на ее проверку. Мьютексы и семафоры являются средствами блокировки процессов, и процессорное время не тратится на проверку их занятости. В отличие от семафоров, мьютексы могут освобождаться только тем потоком, который их удерживал [1], поэтому поведение программ, использующих мьютексы, более предсказуемо. На рис. 3 показан пример кода с использованием мьютексов. Критическая секция кода начинается захватом мьютекса, а заканчивается его освобождением. В результате код критической секции становится атомарным или неделимым.

```
//Для первого потока p1:
     pthread_mutex_lock(&mutex);
     new_p1->next = old->next;
     old->next = new_p1;
     pthread_mutex_unlock(&mutex);
```

```
//Для второго потока p2:
   pthread_mutex_lock(&mutex);
   new_p2->next = old->next;
   old->next = new_p2;
   pthread_mutex_unlock(&mutex);
```

Рис. 3. Использование мьютексов для реализации атомарности

Если поток пытается войти в критическую секцию, захваченную другим потоком, то он будет заблокирован до тех пор, пока другой поток не выйдет из критической секции.

Использование средств взаимоисключения приводит к другим проблемам, основной из которых является взаимоблокировка потоков, или тупиковая ситуация, возникающая из-за многократного захвата и освобождения мьютексов в неправильной последовательности.

Обнаружение тупиков на мьютексах. Средства обнаружения тупиковых ситуаций для прикладных процессов в ОС Linux отсутствуют. Своевременное обнаружение тупиков и последующая их ликвидация является актуальной задачей в интерактивных системах и системах реального времени.

Для обнаружения тупиков используется бихроматический направленный граф Холта [2], в котором множество вершин разбито на два не пересекающихся подмножества: вершин-процессов и вершин-ресурсов. Дуги графа могут соединять только вершины из разных подмножеств. Дуга, направленная от ресурса к процессу, означает приобретение ресурса процессом; в обратном же направлении — запрос процессом ресурса. Доказанная в [2] теорема утверждает, что замкнутая цепь запросов является необходимым условием тупика. Для обнаружения тупика применяется метод редукции (сокращения) графа. Если граф полностью сокращается и все вершины графа становятся изолированными, то система не находится в тупике. Если в результате сокращения графа обнаружена замкнутая цепь запросов, то система попала в тупик.

В результате вызова одного из вариантов библиотечной функции mutex_lock(&mutex) поток, пытающийся войти в критическую секцию, захваченную другим потоком, будет блокирован на мьютексе до тех пор, пока другой поток не освободит этот мьютекс вызовом mutex_unlock(&mutex). При этом для блокировки потока на мьютексе в результате работы библиотечной функции вызывается соответствующая функция ядра. Замкнутая цепочка запросов к мьютексам означает, что процессы попали в тупиковую ситуацию.

В отличие от семафоров, мьютексы, созданные потоками, не отражены в соответствующей системной таблице. Каждый поток имеет собственный список мьютексов в защищенном адресном пространстве процесса, доступ к которому из другого пользовательского процесса невозможен. Однако на уровне ядра можно получить информацию о состоянии блокировки потока на мьютексе. Поиск замкнутой цепи запросов на захват мьютексов как ресурсов можно заменить поиском замкнутой цепи состояний блокировок на мьютексах.

Методика обнаружения блокировок потоков на мьютексах. Для определения того, какая функция ядра блокирует поток на мьютексе, можно использовать следующую технологию:

– написать программу, в которой искусственно создается блокировка потока на мьютексе;

– с помощью утилиты strace получить все системные вызовы, которые были выполнены программой;

– проанализировать набор полученных системных вызовов и путем сопоставления последовательности системных вызовов и программного кода найти функцию, которая блокирует процесс на мьютексе (рис. 4).

Из полученного с помощью программы strace фрагмента дампа видно, что библиотечная функция printf() вызывает системный вызов write(), функция sleep() — выполнение функции ядра nanosleep(), а вызов функции pthread_mutex_lock(&mutex1) заменяется функцией ядра futex() (рис. 5).

```
void *thread_func1(void *arg){
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    /*какие-то действия*/
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return 0;
}
int main(void){
    pthread_t thread1, thread2;
    printf("i'm %d\n", getpid());
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
    pthread_create(&thread1, NULL, &thread_func1, NULL);
    pthread_mutex_lock(&mutex2);
    sleep(1);
    pthread_mutex_lock(&mutex1);
    pthread_mutex_destroy(&mutex2);
    pthread_mutex_destroy(&mutex1);
    return 0;
}
```

Рис. 4. Пример программы, искусственно создающей тупиковую ситуацию

```
set_tid_address(0x7ffcb12a89d0)         = 2147
set_robust_list(0x7ffcb12a89e0, 24)     = 0
. . .
write(1, "i'm 2147\n", 9i'm 2147) = 9
. . .
nanosleep({1, 0}, 0x7fff0290b8c0) = 0
futex(0x600dc0, FUTEX_WAIT_PRIVATE, 2, NULL
```

Рис. 5. Фрагмент дампа, полученного в программе strace

Функция futex() при блокировке, в качестве одного из параметров принимает FUTEX_WAIT_PRIVATE, а при освобождении — FUTEX_WAKE_PRIVATE. В результате анализа данных параметров можно однозначно определить, какое действие будет выполнено — блокировка или освобождение. Перехват функции futex() позволяет

получать информацию о захвате и освобождении мьютексов. Для перехвата вызова системной функции futex() необходимо изменить код ядра. В нем изменяются файлы futex.c и futex.h. В файл futex.h добавляется объявление — указатель на функцию и объявление самой функции, устанавливающей ловушку:

```
typedef long (*futex_hook_t)(u32 __user *uaddr, int op,
u32 val);
futex_hook_t set_futex_hook(futex_hook_t hook);
```

В файл futex.c добавляется описание функции-ловушки:

```
static futex_hook_t futex_hook;
futex_hook_t set_futex_hook(futex_hook_t hook) {
 futex_hook_t old_hook = futex_hook;
 futex_hook = hook;
 printk(KERN_INFO "futex hook %p\n", hook);
 return old_hook;
}
EXPORT_SYMBOL_GPL(set_futex_hook);
```

Данная функция сохраняет в переменную futex_hook адрес функции, переданной в качестве аргумента, и возвращает предыдущее значение данной переменной.

В функцию ядра futex() также необходимо внести изменения — добавить вызов нашей функции-ловушки. Вызов функции-ловушки (futex- hook) записывается в конец функции futex() до вызова функции do_futex() следующим образом:

```
if (futex_hook) futex_hook(uaddr, op, val);
```

Описанный механизм позволяет устанавливать в качестве функции-ловушки любую функцию, определенную в пространстве ядра (в том числе — в загружаемом модуле ядра), прототип которой соответствует заданному:

```
long (*futex_hook_t)(u32 __user *uaddr, int op, u32 val);
```

Внедренная в код ядра функция позволит получить следующую информацию об освобождении/блокировке потоков на мьютексах:

- текущее действие («+» — поток заблокирован на данном мьютексе, «–» — мьютекс освобожден и блокированный поток может продолжить свое выполнение);
- идентификатор текущего потока, который будет заблокирован/разбужен;
- идентификатор потока, захватившего мьютекс, на котором будет заблокирован текущий поток;
- адрес мьютекса в пространстве пользователя;

• минимальный и максимальный идентификаторы потоков, текущего приложения (используется для фильтрации результатов).

Полученная информация заносится в связный список для последующего анализа. Код функции-ловушки приведен на рис. 6. Функция получает три параметра: указатель uaddr — смещение мьютекса в пространстве пользователя; значение op — операция, выполняемая над мьютексом (FUTEX_WAIT_PRIVATE — мьютекс занят, FUTEX_WAKE_PRIVATE — мьютекс свободен) и значение val, находящее по адресу uaddr — при (val == 2) происходит блокировка потока на мьютексе, а при (val == 1) — его освобождение.

```
long futex_monitor(u32 __user *uaddr, int op, u32 val) {
if((val==2&&op==FUTEX_WAIT_PRIVATE)||(val==1&&op==FUTEX_WAKE_PRIVATE)
{
    int min = current->pid, max = current->pid;
    int futex_owner = 0;
    getMinMaxPids(&min, &max);
    if(get_user(futex_owner, (uaddr+2)) == 0){
     if((min<=futex_owner&&futex_owner<=max)||(contains_list(current-
       >pid)))
    { /*ignore read errors*/
     futex_inf_t *temp = (futex_inf_t *)kmalloc(sizeof(futex_inf_t),
     GFP_KERNEL);
     if(temp){
        temp->current_pid = current->pid;
        temp->futex_owner = futex_owner;
        temp->futex_addr = (size_t)uaddr;
        temp->min_process_tid = min;
        temp->max_process_tid = max;
        if(val == 2)
          temp->action = '+';
        else
          temp->action = '-';
        list_add(&(temp->list), &list);
        counter++;
     }
 } } }
return 0; }
```

Рис. 6. Функция-ловушка

Для передачи данных из пространства ядра в пространство пользователя используется файловая система ProcFS, связанная с каталогом /proc. Эта файловая система позволяет установить функции, вы-

зываемые при чтении или записи пользователем в каталоге /proc. В данном случае функция, вызываемая при чтении копирует элементы из списка, находящегося в ядре, в буфер в пространстве пользователя. Для получения этих данных из пространства пользователя нужно вызвать функцию чтения из файла (/proc/futex_hook).

Алгоритм обнаружения тупиков. Обнаружение тупика выполняется в режиме пользователя специальным приложением, которое считывает данные из файла /proc/futex_hook, строит по ним граф и ищет в графе запросов-распределений замкнутые цепи запросов (циклы).

В классическом алгоритме обнаружения тупиков [2] используется два массива: один из них содержит информацию о запросах процессов ресурсов, второй — о распределении ресурсов процессам/потокам).

Основой для обнаружения тупиков является следующая структура:

```
struct parse_t{
 int current_pid; /*идентификатор потока*/
 int futex_owner; /*идентификатор владельца мьютекса*/
 size_t futex_addr; /*адрес мьютекса в пространсве пользователя*/
 int min_process_tid; /*исп. при выборке из общего списка*/
 int max_process_tid; /*исп. при выборке из общего списка*/
};
```

В предлагаемом подходе граф представляется массивом структур типа *struct parse_t.* Вместо массивов запросов и распределений используются массивы заблокированных и освобожденных потоков. Для обнаружения тупиков на первом этапе создается список необработанных потоков. Исходная информация выбирается из файла /proc/futex_hook, сформированного функцией-ловушкой и содержащего информацию обо всех потоках, запрашивавших мьютексы. Алгоритм создания списка показан на рис. 7.

```
начало
 открыть файл . /proc/futex_hook;
 цикл пока не конец файла /proc/futex_hook
   считать символ;
   считать данные в структуру data  типа parse_t;
    если символ == '+'
      добавить структуру data в массив заблокированных потоков;
    иначе
      добавить структуру data в массив освобожденных потоков;
 конец цикла
 цикл для каждого элемента из массива освобожденных потоков
   удалить текущий элемент из массива заблокированных потоков;
 конец цикла
конец
```

Рис. 7. Алгоритм получения списка необработанных потоков

В результате работы алгоритма создается массив необработанных (заблокированных) потоков. В результате его анализа по предлагаемому алгоритму (рис. 8) определяется состояние системы потоков (не находятся ли потоки в тупике).

```
:
начало
 текущий_поток = первый элемент из списка необработанных потоков;
 добавить в результирующий список текущий_поток;
 цикл пока (не рассмотрены все потоки или не обнаружен тупик)
   следующий_поток = поток, удерживающий мьютекс, на котором
заблокирован текущий_поток;
   если (следующий поток определен)
     если (следующий_поток уже находится в результирующем списке)
       обнаружен тупик;
       конец
     иначе
       добавить следующий_поток в результирующий список;
     текущий_поток = следующий_поток;
     конец если;
   иначе
   очистить результирующий список;
   текущий_поток = первый элемент из списка необработанных потоков;
  конец если;
 конец цикла пока;
  если (количество записей в результирующем списке == 1)
   очистить результирующий список;
 конец если;
конец.
```

Рис. 8. Алгоритм обнаружения тупиков

Автоматизация внесения изменений в ядро. Вместо внедрения изменений в ядро вручную, можно с помощью системы контроля версий git сгенерировать патч (рис. 9). Для его применения, необходимо:

1) сохранить его в файл, например kernel.patch (расширение не имеет значения);

2) скачать исходные коды ядра с сайта www.kernel.org;

3) в режиме командной строки выполнить следующую последовательность команд:

- скопировать исходники ядра в папку /usr/src/linux:

```
# cp linux-<версия ядра>.tar.bz2 /usr/src
```

- распаковать исходники:

```
# tar -xjf /usr/src/linux-<версия ядра> -C /usr/src/
```

- скопировать патч в папку с исходниками ядра:

```
# cp kernel.patch /usr/src/linux-<версия ядра>
```

- перейти в папку с исходниками ядра:

```
$ cd /usr/src/linux-<версия ядра>
```

- применить патч:

```
# patch -p0 < kernel.patch
```

```
diff --git a/include/linux/futex.h b/include/linux/futex.h
index b0d95ca..706e384 100644
--- a/include/linux/futex.h
+++ b/include/linux/futex.h
@@ -11,6 +11,10 @@ union ktime;
 long do_futex(u32 __user *uaddr, int op, u32 val, union ktime *timeout,
       u32 __user *uaddr2, u32 val2, u32 val3);
+typedef long (*futex_hook_t)(u32 __user *uaddr, int op, u32 val);
+
+futex_hook_t set_futex_hook(futex_hook_t hook);
+
 extern int
 handle_futex_death(u32 __user *uaddr, struct task_struct *curr, int pi);
diff --git a/kernel/futex.c b/kernel/futex.c
index 8879430..587f7ef 100644
--- a/kernel/futex.c
+++ b/kernel/futex.c
@@ -2693,6 +2693,17 @@ long do_futex(u32 __user *uaddr, int op, u32 val, ktime_t *timeout,
  return -ENOSYS;
 }
+static futex_hook_t futex_hook;
+
+futex_hook_t set_futex_hook(futex_hook_t hook)
+{
+  futex_hook_t old_hook = futex_hook;
+  futex_hook = hook;
+  printk(KERN_INFO "futex hook %p\n", hook);
+  return old_hook;
+}
+
+EXPORT_SYMBOL_GPL(set_futex_hook);
 SYSCALL_DEFINE6(futex, u32 __user *, uaddr, int, op, u32, val,
       struct timespec __user *, utime, u32 __user *, uaddr2,
@@ -2724,6 +2735,9 @@ SYSCALL_DEFINE6(futex, u32 __user *, uaddr, int, op, u32, val,
     cmd == FUTEX_CMP_REQUEUE_PI || cmd == FUTEX_WAKE_OP)
       val2 = (u32) (unsigned long) utime;
+ if(futex_hook)
+   futex_hook(uaddr, op, val);
+
  return do_futex(uaddr, op, val, tp, uaddr2, val2, val3); }
```

Рис. 9. Патч для перехвата функции ядра futex

4) перекомпилировать ядро, для этого в терминале выполнить следующие команды:

```
# make mrproper #очищает результаты предыдущей сборки

# make menuconfig # ручная конфигурация ядра

# make install -j<кол-во процессоров + 1> #выполняет сборку ядра

# make modules_install #необходимо только при первой сборке

# cp arch/x86/boot/bzImage /boot/vmlinuz-YourKernelName #скопировать ядро в папку /boot

# mkinitcpio -k /boot/vmlinux-YourKernelName -c /etc/mkinitcpio. conf -g /boot/initramfs-YourKernelName.img

# создать начальный RAM диск для загрузки ядра.
```

Заключение. При работе многопоточных приложений с разделяемыми структурами данных необходимо обеспечивать монопольное пользование потоками этих структур. В качестве средств взаимоисключения часто используются мьютексы. Необдуманное использование мьютексов может явиться причиной попадания системы потоков в тупик. Средства обнаружения тупиковых ситуаций для прикладных процессов в ОС Linux отсутствуют. Своевременное обнаружение тупиков и последующая их ликвидация является актуальной задачей в интерактивных системах и системах реального времени. Предлагаемая методика позволяет обнаружить тупики, используя список блокированных на мьютексах потоков, полученный средствами, дополнительно включенными в ядро ОС Linux. Методика была реализована и протестирована на известных алгоритмах, что подтвердило ее работоспособность.

ЛИТЕРАТУРА

[1] Хьюз Л., Хьюз Т. *Параллельное и распределенное программирование на C++*. Москва, Издательский дом «Вильямс», 2004, 672 с.

[2] Шоу А. *Логическое проектирование операционных систем*. Москва, Мир, 1981, 288 с.

Статья поступила в редакцию 10.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Белоус В.С., Крищенко В.А., Рязанова Н.Ю. Обнаружение тупиков на мьютексах в многопоточных приложениях. *Инженерный журнал: наука и инновация*, 2013, вып. 6. URL: http://engjournal.ru/catalog/it/hidden/771.html

Белоус Вячеслав Сергеевич родился в 1993 г. Студент 3-го курса кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана. e-mail: spectr0@mail.ru

Крищенко Всеволод Александрович родился в 1975 г., окончил магистратуру МГТУ им Н.Э. Баумана в 1998 г. Канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии». Автор более 15 научных трудов, научные интересы - статический и динамический анализ и верификации программного обеспечения и сетевых протоколов. e-mail: kva@bmstu.ru

Рязанова Наталья Юрьевна родилась в 1951 г. Канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана. Автор 36 печатных работ. Область научных интересов: разработка системного программного обеспечения, алгоритмы машинной графики. e-mail: ryaz_nu@mail.ru