Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе**

**Дисциплина**: Проектирование ОС и компонентов

**Тема**: Анализ исходного кода системных вызовов и их частичная замена

Выполнил студент гр. 13541/4 Степанов Д.С.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт - Петербург

2017

**Цель работы**

Изучить системные вызовы sys\_sched\_setscheduler, sys\_sched\_getscheduler, sys\_sched\_yield - их назначение, функциональность и параметры. Написать код, который является потребителем данных системных вызовов, а также перехватывает их.

**Выполнение работы**

В операционной системе Linux предусмотрено семейство системных функций, предназначенных для изменения параметров работы планировщика задач. С их помощью можно изменять приоритет процесса, стратегию планирования, привязку к процессору (processor affinity), а также явно передать (yield) процессор в использование другим задачам.

Системный вызов sched\_getscheduler запрашивает политику планирования, действующую в отношении процесса, идентифицируемого параметром pid. Если pid равен 0, считывается политика вызвавшего процесса. В случае успеха системный вызов возвращает политику sched\_fifo, sched\_ rr или sched\_normal (последняя также называется sched other). Соответствующая служебная процедура sys\_sched\_getscheduler вызывает функцию find\_process\_by\_pid , которая находит дескриптор процесса по переданному значению pid и возвращает значение его поля policy.

Системный вызов sched\_setscheduler устанавливает как политику планирования, так и соответствующие параметры для процесса, идентифицируемого параметром pid. Если pid равен 0, устанавливаются параметры планировщика, применяемые к вызвавшему процессу.

Соответствующая служебная процедура sys\_sched\_setscheduler просто вызывает функцию do\_sched\_setscheduler . Эта функция проверяет допустимость политики планирования, определяемой параметром policy, и нового приоритета, определяемого параметром param->sched\_priority. Она также проверяет, есть ли у процесса способность cap\_sys\_nice, или наличие прав суперпользователя у его владельца. Если все в порядке, она удаляет процесс из очереди на выполнение (если он выполняемый), обновляет статический и динамический приоритеты и приоритет реального времени у процесса, возвращает процесс в очередь на выполнение и, если необходимо, вызывает функцию resched\_task для вытеснения текущего процесса, принадлежащего данной очереди.

Системный вызов sched\_yieid позволяет процессу добровольно освободить процессор без приостановки своего выполнения. Процесс остается в состоянии task\_running, а планировщик заносит его либо в набор процессов с истекшими квантами времени (если это обычный процесс), либо в конец списка в очереди на выполнение (если это процесс реального времени). Затем вызывается функция schedule. В результате у других процессов с тем же динамическим приоритетом появляется возможность поработать. Данный вызов используется, в основном, процессами реального времени, принадлежащими классу SCHED\_FIFO.

Прототипы функций:

int sched\_getscheduler(pid\_t pid);

int sched\_setscheduler(pid\_t pid, int policy, const struct sched\_param \*param);

int sched\_yield(void);

Приведем код, использующий перечисленные системные вызовы:

#include <sched.h>

#include <stdio.h>

void printPolicy(){

int policy = sched\_getscheduler(0);

switch (policy) {

case SCHED\_OTHER:

printf ("Обычная политика\n");

break;

case SCHED\_RR:

printf ("Политика карусели\n");

break;

case SCHED\_FIFO:

printf("Политика FIFO\n");

break;

case -1:

printf("sched\_getscheduler");

break;

default:

printf("Неизвестная политика!\n");

}

}

int main(){

printPolicy();

struct sched\_param sp = { .sched\_priority = 1 };

int ret;

ret = sched\_setscheduler(0, SCHED\_RR, &sp);

printPolicy();

if (sched\_yield() == 0)

printf("sched\_yield\n");

return 0;

}

Вызов sched\_setscheduler() устанавливает policy в качестве политики планирования процесса, определенного pid. Любые параметры, связанные с policy, устанавливаются через sp. Если pid равен 0, политика планирования и параметры устанавливаются для вызывающего процесса. В случае успеха вызов возвращает 0. При неудаче вызов возвращает –1 и errno заполняется соответствующим образом. Допустимые поля внутри структуры sched\_param зависят от того, какие политики планирования поддерживаются операционной системой. Для SCHED\_RR и SCHED\_FIFO требуется одно поле, где указывается статический приоритет. SCHED\_OTHER не

использует полей параметров, но политики планирования, которые будут поддерживаться в будущем, могут потребовать использования новых полей. Таким образом, в переносимых и разрешенных программах не следует делать предположений относительно компоновки структуры.

Установить политику планирования процесса и параметры очень просто:

Данный фрагмент кода устанавливает политику карусели со статическим приоритетом 1 в качестве политики планирования вызывающего процесса. Мы предполагаем, что 1 — это допустимая величина приоритета, хотя технически так будет не всегда. Установка политики планирования, кроме SCHED\_OTHER, требует наличия характеристики CAP\_SYS\_NICE. Следовательно, пользователь, обладающий правами root, как правило, запускает процессы реального времени. Начиная с версии ядра 2.6.12

пределы ресурсов RLIMIT\_RTPRIO позволяют пользователям без прав root устанавливать политики реального времени до определенной величины приоритета.

В случае ошибки возможны следующие значения errno:

* EFAULT — указатель sp определяет недопустимый или недоступный фрагмент памяти;
* EINVAL — политика планирования, определенная в policy, недопустима, или величина, указанная в sp, не имеет смысла для данной политики (касается только sched\_setscheduler());
* EPERM — вызывающий процесс не имеет необходимых характеристик;
* ESRCH — отсутствует работающий процесс с указанной величиной pid.

Вызов sched\_yield() в случае успеха вызов возвращает 0; при неудаче — –1 и отправляет errno в соответствующем коде ошибки. В Linux, а также, скорее всего, в большинстве других UNIX-систем, sched\_yield() всегда выполняется успешно, возвращая 0.

На практике существует не так много вариантов правомерного использования sched\_yield() в действительно многозадачной системе с вытеснением наподобие Linux. Ядро отлично справляется с принятием оптимальных и наиболее выгодных решений по планированию самостоятельно — действительно, ведь ядро лучше, чем отдельное приложение, подготовлено к задачам очередности вытеснения.

Почему же тогда в POSIX вообще существует такой системный вызов, как

«переделка расписания»? Ответ заключен в приложениях, которым приходится ожидать событий вовне, которые могут быть вызваны действиями пользователя, аппаратного обеспечения или другими событиями. В частности, если один процесс должен ожидать другого, самое простое решение — «просто уступи ресурсы процессора, пока другой процесс не завершится». В качестве примера можно привести реализацию простого потребителя в паре «потребитель — производитель»:

/\* потребитель... \*/

do {

while (producer\_not\_ready ())

sched\_yield ());

process\_data ();

} while (!time\_to\_quit ());

Обычно работа программ UNIX управляется событиями и использует какой-либо механизм блокировки (например, конвейер) между потребителем и производителем, а не sched\_yield(). В данном случае потребитель получает с конвейера информацию, блокируя себя, если нужные данные пока недоступны. Производитель в это же время отправляет данные в конвейер по мере появления. Таким образом, пользовательское пространство не влияет на координацию процесса, которой занимается ядро, способное оптимизировать ситуацию: приостанавливать процессы в нужный момент и возобновлять их по мере необходимости. В целом программы UNIX должны стремиться к работе, управляемой происходящими событиями, которые основываются на дескрипторах файла блокировки.

Результат работы приведенного кода:

danya-virtual-machine lab6 # ./sched

Обычная политика

Политика карусели

sched\_yield

С помощью утилиты strace проверим, действительно ли запускались нужные системные вызовы:

danya-virtual-machine lab6 # strace ./sched

...

sched\_getscheduler(0) = 0 (SCHED\_OTHER)

...

sched\_setscheduler(0, SCHED\_RR, [1]) = 0

sched\_getscheduler(0) = 2 (SCHED\_RR)

...

sched\_yield() = 0

write(1, "sched\_yield\n", 12sched\_yield

) = 12

exit\_group(0) = ?

+++ exited with 0 +++

Теперь перехватим системные вызовы с помощью системы SystemTap. SystemTap позволяет пользователям разрабатывать и повторно использовать простые сценарии для осуществления глубокого анализа операций, выполняющихся в рамках функционирующей системы на основе ядра Linux. Эти сценарии могут быть разработаны с целью извлечения значений параметров системы, их фильтрации и быстрого (а также безопасного) накопления, что позволяет производить диагностику сложных проблем с производительностью (или даже функциональных проблем) системы.

Основная идея, положенная в основу механизма сценариев SystemTap, заключается в именовании событий (events) и связи с ними обработчиков (handlers). В момент, когда SystemTap начинает исполнение сценария, на самом деле средствами SystemTap инициируется мониторинг события; как только событие наступает, ядро Linux выполняет связанный с ним обработчик как быструю подпрограмму, после чего продолжает работу в обычном режиме.

Существует несколько видов событий; вход в функцию/выход из функции, истечение времени таймера, завершение сессии, и.т.д. Обработчик является набором объявлений языка сценариев, устанавливающих операции, которые необходимо выполнить при наступлении заданного события. Эти операции обычно заключаются в извлечении данных из контекста события, сохранении этих данных во внутренних переменных, а также в выводе результатов.

Скрипт, выводящий информацию о перехваченных вызовах:

#! /bin/sh

# suppress some run-time errors here for cleaner output

//bin/true && exec stap --suppress-handler-errors --skip-badvars $0 ${1+"$@"}

/\* configuration options; set these with stap -G \*/

global follow\_fork = 0 /\* -Gfollow\_fork=1 means trace descendant processes too \*/

global timestamp = 1 /\* -Gtimestamp=0 means don't print a syscall timestamp \*/

global elapsed\_time = 0 /\* -Gelapsed\_time=1 means print a syscall duration too \*/

global thread\_argstr%

global thread\_time%

global syscalls\_nonreturn[2]

probe begin

{

/\* list those syscalls that never .return \*/

syscalls\_nonreturn["exit"]=1

syscalls\_nonreturn["exit\_group"]=1

}

function filter\_p()

{

if (target() == 0) return 0; /\* system-wide \*/

if (!follow\_fork && pid() != target()) return 1; /\* single-process \*/

if (follow\_fork && !target\_set\_pid(pid())) return 1; /\* multi-process \*/

return 0;

}

probe nd\_syscall.\*

{

if (filter\_p()) next;

thread\_argstr[tid()]=argstr

if (timestamp || elapsed\_time)

thread\_time[tid()]=gettimeofday\_us()

if (name in syscalls\_nonreturn)

report(name,argstr,"")

}

probe nd\_syscall.sched\_\*.return

{

if (filter\_p()) next;

report(name,thread\_argstr[tid()],retstr)

}

function report(syscall\_name, syscall\_argstr, syscall\_retstr)

{

if (timestamp || elapsed\_time)

{

now = gettimeofday\_us()

then = thread\_time[tid()]

if (timestamp)

prefix=sprintf("%s.%06d ", ctime(then/1000000), then%1000000)

if (elapsed\_time && (now>then)) {

diff = now-then

suffix=sprintf(" <%d.%06d>", diff/1000000, diff%1000000)

}

delete thread\_time[tid()]

}

/\* add a thread-id string in lots of cases, except if

stap strace.stp -c SINGLE\_THREADED\_CMD \*/

if (tid() != target()) {

prefix .= sprintf("%s[%d] ", execname(), tid())

}

printf("%s%s(%s) = %s%s\n",

prefix,

syscall\_name, syscall\_argstr, syscall\_retstr,

suffix)

delete thread\_argstr[tid()]

}

Результаты работ скрипта:

danya@danya-virtual-machine ~/OS/lab6 $ sudo stap hello.stp -w -c ./sched

Обычная политика

Политика карусели

sched\_yield

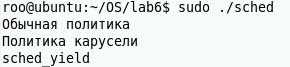
Sun May 14 12:17:12 2017.946931 sched\_getscheduler(0) = 0

Sun May 14 12:17:12 2017.947121 sched\_setscheduler(0, SCHED\_RR, 0x7fffd04cca90) = 0

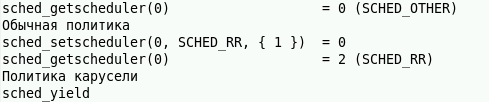
Sun May 14 12:17:12 2017.947144 sched\_getscheduler(0) = 2

Sun May 14 12:17:12 2017.947168 sched\_yield() = 0

Как видно из вывода, все необходимые системные вызовы были перехвачены и обработаны. Теперь проверим работу приведенных выше программ на ядре версии 2.6. Теоретически проблем возникнуть не должно, поскольку назначение и прототип функций в ядре версии 2.6. ничем не отличается.



Перехват:



**Вывод**

В ходе выполнения данной лабораторной работы были изучены системные вызовы sys\_sched\_setscheduler, sys\_sched\_getscheduler, sys\_sched\_yield. Данные системные вызовы относятся к часто используемым, и применяются для управления политикой планирования в ОС Linux. От данных системных вызовов зависит быстродействие работы операционной системы. Также в ходе работы была написана программа, которая является потребителем перечисленных вызовов. Далее с помощью системы SystemTap системные вызовы были перехвачены и обработаны. В завершении работы экспериментальным путём было доказано, что в ядрах более ранней версии назначение и работа системных вызовов никак не изменилась.

**Список использованных источников**

1. Лав Р. Ядро Linux //Описание процесса разработки: Пер. с англ.–3-е изд.− М.: Вильямс. – 2013.
2. Интернет-источник. Официальный сайт SystemTap. https://sourceware.org/systemtap/ Дата обращения – 14.05.2017
3. Интернет источник. Man на русском языке. <http://ru.manpages.org/> Дата обращения – 14.05.2017