Исследование способов динамического определения наличия привязки процесса к процессору в ядре Linux

Н.В. Кудрявцев

***Аннотация*—В статье проводится обобщенный анализ миграции процесса между процессорами в ядре Linux. Проблема рассматривается применительно к суперкомпьютерным комплексам, где нежелательна установка дополнительных модулей и стороннего программного обеспечения. Производится сравнение методов определения привязки процесса к процессору и выявляется оптимальный метод по критериям : вероятность определения факта миграции, время выполнения задачи и степень влияния метода на выполнение других задач.**

***Ключевые слова*—ядро Linux, миграция процесса, привязка процесса, kprobes, переменная окружения LD_PRELOAD, модуль ядра.**

I. Введение

Как в научных исследованиях, так и в промышленности все чаще возникают задачи, требующие использования высокопроизводительных вычислений. Вычислительные системы, разработанные и оптимизированные для работы с параллельными задачами, осуществляют одновременное исполнение нескольких задач путем разделения вычислительных элементов (CPU) между задачами по времени или степени параллелизма. Наряду с преимуществами, которые предоставляют высокопроизводительные вычислительные системы, возникает ряд проблем, связанных с энергоэффективностью, планированием и оптимизацией выполнения вычислительных задач, и многими другими проблемами. Зачастую отсутствие учета аппаратных возможностей вычислительной машины приводит к миграции процесса с одного CPU на другой, а неоптимизированное выравнивание данных по страницам памяти увеличивает время миграции процесса в десятки, а то и тысячи раз [1], что увеличивает время выполнения задачи, повышает энергопотребление, снижается эффективность. Статья посвящена исследованию и сравнению методов определения миграции процесса с одного процессора на другой в распределенных вычислительных системах. Такая постановка задачи вызвана тем, что выявление задач с активной миграцией процессов является важной задачей для повышения эффективности работы суперкомпьютера.

Проблема миграции широко распространена. Своевременное определение недопустимого количества миграции процесса между CPU может помочь в оптимизации загрузки процессоров, в частности, уменьшении времени простоя процессоров, что позволяет увеличить количество одновременно запущенных распределенных вычислительных задач на кластере, а также уменьшить время работы вычислительных задач.

В данной статье исследование проводилось исключительно для операционных систем с ядром Linux. Исследование проводилось с помощью суперкомпьютера «Ломоносов», тестовые программы компилировались с использованием библиотеки OpenMPI.

II. Функции привязки процесса к процессору

Операционная система Linux позволяет осуществить привязку процесса к процессору двумя способами: системным вызовом ***sched_setaffinity*** из библиотеки libc и функцией ***pthread_setaffinity_np***. Для каждой из этих функций существует пара – функция для получения маски привязки процесса к процессору: **sched_getaffinity** и **pthread_getaffinity_np**.

Существует большое количество библиотек для параллельных вычислений: MPI (libmpi), NUMA (libnuma), предоставляющих свои методы для привязки процесса к процессору: в случае использования библиотеки MPI – это параметры запуска **–bind-to** для привязки и **–report-bindings** для определения привязки, в случае библиотеки NUMA – **numa_sched_setaffinity** и **numa_sched_getaffinity** соответственно. Исследование показало, что привязка и определение привязки процесса к процессору реализованы с помощью функций **sched_setaffinity** и **sched_getaffinity** с дополнительным функционалом. Анализируя исходный код функции привязки **sched_setaffinity,** становится понятно, что в дескрипторе процесса ***task_struct*** проверяется, выставлен ли флаг **PF_NO_SETAFFINITY**, запрещающий привязку, если он не выставлен, производится установка маски **cpu_allowed** привязки процесса к CPU. Помимо всего прочего, функция должна проверить, стоит ли процесс в очереди на выполнение у процессора, который отсутствует в обновленной маске привязки. В худшем случае придется перенести процесс в другую очередь на выполнение.

III. Способы определения факта привязки процесса к процессору

Конечно, наличие параметра запуска **–report-bindings** у библиотеки MPI может сказать о наличии факта привязки процесса к процессору, более того, в некоторых случаях параметрами запуска вычислительных задач можно управлять, но, зачастую,

Статья получена 17 апреля 2016. Данная работа является частью магистерской диссертации “Определение наличия миграции процесса между ядрами по грубым данным аппаратной статистики”, Кудрявцев Н.В, магистр ВМК МГУ имени Ломоносова (email: nkudrjavcev@gmail.com).

вычислительные задачи поставляются в виде пакета, который представляет из себя черный ящик, поэтому нет возможности управлять конфигурацией, или исследование становится достаточно трудоемким. В таком случае возникает задача динамического определения (то есть во время выполнения программы) факта привязки процесса к процессору. В таком случае, наличие привязки стоит определять другими способами:

- использовать механизм kprobes [4];
- использовать механизм *LD_PRELOAD;*

Начиная с версии ядра Linux 2.6 появилась возможность использовать Kernel probes – обработчики событий, выполняющиеся на уровне ядра (kernel space) [4]. Различают 3 обработчика событий: kprobes, kretprobes, jprobes, которые можно виртуально “поставить” на функцию в ядре (kprobes), начало обработки обработки функции (jprobes) для доступа к аргументам функции и окончание обработки функции (kretprobes). Поведение обработчиков описывается в виде модуля ядра, непосредственная установка происходит во время регистрации модуля макросом module_init(), снятие обработчика производится при выгрузге модуля ядра макросом module_exit(). Функция __switch_to выполняет основную работу по переключению процессов. В качестве аргументов передаются дескрипторы task_struct старого и нового процессов. Отсортировать и исключить “ненужные” переключения возможно исследовав структуру task_struct процесса. Таким образом, постановка kprobes на функцию __switch_to даст полную картину переключения контекста. Аналогичным способом ставится обработчик на системный вызов sys_sched_setaffinity. К преимуществам данного метода относятся:

- наиболее четкое определение наличия переключения контекста процесса;
- возможность получения и исследования дополнительной информации, например причины переключения контекста;
-

Недостатки данного метода очевидны:

- ядро не всегда компилируется с поддрежкой механизма kprobes;
- количество вызовов функции __switch_to ~1000 раз в секунду [2], несмотря на сравнительно небольшое время переключения на обработчик ~ 0.1 микросекунд, обработка такой функции на уровне ядра значительно скажется на производительности вычислительной системы из-за количества срабатываний;
- при постановке kprobes обработчики будут вызываться не только для конкретной задачи, но для всех вычислительных задач в целом, сортировка вычислительных задач производится путем анализа дескриптора процесса task_struct;
- случайные ошибки в коде модуля ядра приведут к отказу работы ядра в целом (kernel panic);
- для разработчика необходим опыт написания модулей ядра и понимание работы ядра;
- необходимо знать особенности работы kprobes, например если количество вызовов функции не соответствует количеству возвратов (например функция объявлена с атрибутом noreturn) это приведет к неопределенной ситуации;
-

Таким образом, использование механизма kprobes является достоверным, но малоэффективным способом (по выбранным критериям) исследования наличия привязки процесса к процессору.

Использование механизма LD_PRELOAD также позволяет решить задачу определения факта миграции процесса между CPU. Переменная окружения LD_PRELAOD – стандартная возможность динамического линковщика. Если файл некоторой разделяемой библиотеки указан в переменной окружения LD_PRELOAD, эта библиотека принудительно загружается и ее символы считаются более “приоритетными” и перекрывают одноименные символы, если таковы существую в других библиотеках, загружаемых ld.so при запуске на выполнение бинарного файла ELF.
Алгоритм использования данного метода следующий: слинковать пакет задач со специально подготовленной динамической библиотекой с помощью переменной окружения LD_PRELOAD, в которой осуществить перехват и логгирование функций **sched_setaffinity**, **sched_getaffinity**, ***pthread_setaffinity_np***, ***pthread_getaffinity_np***, а также некоторых функций, о которых будет описано ниже.
Далее представлен пример постановки обработчика на функцию *sched_setaffinity:*

```
int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask) {


    if(_sched_setaffinity == NULL) {
        if(!(handle = dlopen (currentLibcPath, RTLD_LAZY))) {
            fprintf(stderr, "error dlopen libc while sched_setaffinity\n");
            return -1;
        }
        _sched_setaffinity = (int (*)(int, unsigned int, long unsigned int*))dlsym(handle, "sched_setaffinity");
    }
    int status = _sched_setaffinity(pid, len, mask);
    createRecordBind("sched_setaffinity\0", pid, mask, status);
    return status;
}
```

К преимуществам данного метода можно отнести:

- отсутствие нагрузки на систему в целом;
- ошибки в коде влияют исключительно на выполнение конкретной вычислительной задачи;
-

К недостаткам данного метода можно отнести:

- достаточно высокую, но не 100%-ю вероятность определения наличия привязки процесса к процессору;

– необходимость установки дополнительных обработчиков на функции, в которых может произойти переключение контекста;

Начиная с версии ядра Linux 3.5 появился механизм, аналогичный kprobes, но нацеленный на анализ поведения выполняемых в пространстве пользователя приложений – механизм uprobes [6]. Основная идея использования данного подхода схожа с механизмом LD_PRELOAD.

Таким образом для данной задачи наиболее эффективным применением является использование механизма LD_PRELOAD. Подробнее о механизме работы этого метода будет рассказано далее.

IV. Проверка факта привязки процесса к процессору

Операционная система Linux может вовсе не поддерживать механизм привязки процесса. В общем случае непонятно, как отреагирует на такое поведение произвольный пакет с вычислительной задачей. Возможным вариантом развития событий в такой ситуации может быть продолжение выполнения задачи без привязки, что скорее всего приведет к периодическому переключению контекста процесса, что скажется на производительности вычислительной машины [1].

Миграция процесса с одного процессора на другой (то есть из очереди на выполнение одного процессора в очередь на выполнение другого) происходит при вызове планировщиком функции ядра **try_to_wake_up** [2]. Эта функция будит спящий или остановленный процесс, переводя его в состояние TASK_RUNNING и занося его в очередь на выполнение процессора. Функция выбирает целевую очередь на выполнение, руководствуясь некоторыми эвристическими правилами:

– Если какой-нибудь процессор в системе простаивает, функция выбирает его очередь на выполнение в качестве целевой. Предпочтение отдается процессору, ранее выполнявшему процесс или локальному процессору.
– Если рабочая нагрузка процессора, выполнявшего процесс последним, значительно ниже, чем у локального процессора, функция выбирает прежнюю очередь на выполнение в качестве целевой.
– Если процесс выполнялся недавно, функция выбирает прежнюю очередь на выполнение в качестве целевой.
– Если перенос процесса на локальный процессор уменьшает дисбаланс между процессорами, целевой становится локальная очередь на выполнение.

По окончании этого шага, функция определила целевой процессор, который будет выполнять разбуженный процесс.

Таким образом, процесс может мигрировать с одного процессора на другой только в том случае, если поле ***state*** дескриптора процесса не равно TASK_RUNNING. Изменение поля ***state*** обычно возникает при вызове функций, выполняющих ожидание какого-либо системного события (например, функции nanosleep, sleep, usleep, wait – переводят состояние текущего процесса в TASK_INTERRUPTIBLE, mmap, read, write – переводят состояние текущего процесса в TASK_UNINTERRUPTIBLE). Количество функций, при которых может может произойти переключение контекста ограничено, поставив обработчики на такие функции и сравнив текущий CPU процесса до и после выполнения функции (получить номер CPU процесса можно с помощью функции sched_getcpu [3]) можно определить факт переключения контекста процесса. Вызов функций **sched_setaffinity** или ***pthread_setaffinity_np***, *а также отсутсвие* миграций процесса между CPU будет являться подтверждением факта привязки процесса к процессору.

V. Заключение

В статье был произведен обобщенный анализ миграции процесса между процессорами в ядре Linux. Были подвергнуты сравнению два разных подхода динамического анализа наличия миграций процесса между процессорами : с помощью механизма kprobes и с помощью переменной окружения LD_PRELOAD. Были приведены аргументы в пользу выбора метода с помощью переменной окружения LD_PRELOAD.

Метод определения наличия факта миграции процесса между процессорами с помощью переменной окружения LD_PRELOAD не является 100% эффективным методом определения миграции процесса между процессорами, но с достаточно большой вероятностью может сказать нам об этом, программные ошибки не приведут к ошибкам ядра, но повлияют лишь на выполнение исследуемого процесса. Конечно, наиболее эффективным методом определения миграции процесса является исследование task_struct на уровне ядра с помощью механизма kbrobes, но это приведет к значительному (недопустимому с точки зрения исследования) уменьшению эффективности программы, требует обширных знаний и пониманий в работе ядра Linux и работы механизма kprobes.

Существует ряд инструментов, с помощью которых можно также решить описанную задачу, таких как Utrace, SystemTap, Uprobes [6], но алгоритм работы для этих случаев будет схож с методом с использованием LD_PRELOAD.

Библиография

[1] C. Li, C. Ding, and K. Shen, “Quantifying the cost of context switch” in Proc. of ACM Workshop on Experimental Computer Scienc (ExpCS’07), San Diego, California, June 13-14, 2007.
[2] Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel, 3rd Edition”, Publisher: O'Reilly , 2005

[3] Linux. URL : http://linux.die.net/man/3/sched_getcpu
[4] Linux. URL : https://www.kernel.org/doc/Documentation/kprobes.txt
[5] Linux. URL : http://www.thinkingparallel.com/2006/08/18/more-information-on- pthread_setaffinity_np-and-sched_setaffinity/
[6] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In Linux Symposium, Ottawa, Canada, June 2007.

Research of methods for dynamic determining the presence of a binding process to the processor in Linux kernel

N.V. Kudryavtsev

Abstract— **This article presents a generalized analysis of the migration process between the processors in the Linux kernel. The problem is considered as applied to supercomputers systems, which is undesirable for extra modules and third-party software. It is performed a comparison of methods for determining the binding process to the processor and identified the optimal method for determining the criteria of likelihood of migration, of the tasks execution time and of the influence degree of the method to perform other tasks.**

Keywords—Linux kernel, process migration, process binding, kprobes, LD_PRELOAD environment variable, kernel module.