# Лабораторная работа 10. Планирование и диспетчеризация процессов в Linux

**Аннотация:** Цель работы: получить представление о стратегиях и алгоритмах планирования процессов в Linux, изучить основные функции планировщика процессов в Linux.

**Задание 1**. Изучите особенности планировщика Linux и системный вызов *nice()*.

Ход выполнения:

1. Прочитайте вводную информацию о планировщике Linux.

Планировщик процессов Linux менялся со временем. В настоящий момент планировщик, доступный, начиная с версии ядра Linux 2.6.23, называется CFS (Completely Fair Scheduler). Название происходит от внедрения в планировщик принципа равноправной очереди, планирующего алгоритма, который старается обеспечить равный доступ к ресурсам всем конкурирующим потребителям. CFS резко отличается от других планировщиков процессов в UNIX, включая своего предшественника, планировщик ввода-вывода.

CFS представляет значительно отличающийся алгоритм, называемый *беспристрастным планированием*, в котором квантов времени как единиц контроля доступа к процессору уже нет. Вместо них CFS назначает каждому процессу долю процессорного времени. Алгоритм очень прост: CFS запускается, назначая N процессам каждому по 1/N времени процессора. Затем эти доли уточняются, «взвешивая» каждый процесс по его точной величине. Процессы с нулевой точной величиной по умолчанию имеют вес, равный единице, следовательно, их пропорция не меняется. Процессы с меньшей точной величиной (высший приоритет) получают больший вес, увеличивая свою долю ресурсов процессора, в то время как процессы Completely Fair Scheduler с большей точной величиной (низший приоритет) получают меньший вес, уменьшая свой расход ресурсов процессора.

2. Определите версию ядра своей операционной системы Linux и проверьте, что она не менее 2.6.23.

uname -r

3. Ознакомьтесь с системным вызовом *nice().*

Linux предоставляет несколько вызовов для получения и установки значения любезности процесса. Самый простой из них — nice():

#include <unistd.h>

int nice (int inc);

Успешная работа *nice()* увеличивает приоритет (значение любезности) процесса на значение *inc* и возвращает обновленную величину.

В случае ошибки *nice()* возвращает *–1*.

4. Из-за того, что *nice()* возвращает новое значение приоритета, *–1* может быть возвращено и в случае успеха. Чтобы различать успешное срабатывание и сбой, можно обнулить *errno* перед запуском, а затем проверить ее значение. Наберите следующую программу и запустите. Что выводится в результате работы программы?

int ret;

errno = 0;

ret = nice (10); /\* увеличение значения приоритета на 10 \*/

if (ret == –1 &&errno != 0)

perror ("nice");

else

printf ("значение приоритета теперь равно %d\n", ret);

5. Как изменится результат работы программы, если *ret = nice (10)* заменить на *ret = nice (0)*?

6. С помощью программы увеличьте приоритет процесса на 10.

7. С помощью программы увеличьте приоритет процесса на 30.

8. С помощью программы уменьшите приоритет процесса на 30.

**Задание 2**. Изучите системные вызовы *getpriority()* и *setpriority()*.

Ход выполнения:

1. Прочитайте информацию о системных вызовах *getpriority()* и *setpriority()*.

#include <sys/time.h>

#include <sys/resource.h>

int getpriority (int which, int who);

int setpriority (int which, int who, int prio);

Эти вызовы воздействуют на процесс, группу процессов или пользователей, что определено с помощью *which* и *who*. Значение *which* должно быть PRIO\_PROCESS, PRIO\_PGRP или PRIO\_USER, в то время как *who* указывает идентификатор процесса, группы процессов или идентификатор пользователя соответственно. Если значение *who* равно *0*, то вызов работает с текущими идентификатором процесса, группы процессов или идентификатором пользователя соответственно.

Вызов *getpriority()* возвращает наибольший приоритет (наименьшую численную величину значения любезности) каждого из указанных процессов. Вызов *setpriority()* устанавливает значение приоритета каждого из указанных процессов, равное *prio*.

Вызов *getpriority()* возвращает *–1* в случае ошибки. Это может быть и возвращенный результат при успешном срабатывании, поэтому программист должен очистить *errno* перед вызовом, если нужно обработать условия ошибки. Вызов *setpriority()* не приводит к таким проблемам, он всегда возвращает 0 в случае успеха и *–1* – в случае ошибки.

2. Выясните текущую величину приоритета процесса с помощью кода:

int ret;

ret = getpriority (PRIO\_PROCESS, 0);

printf ("значение любезности равно %d\n", ret);

3. Установите приоритеты всех процессов текущей группы процессов равными 10:

int ret;

ret = setpriority (PRIO\_PGRP, 0, 10);

if (ret == –1)

perror ("setpriority");

**Задание 3**. Изучите особенности привязки процессов к процессорам в Linux.

Ход выполнения:

1. Прочитайте вводную информацию о необходимости привязки процессов к процессорам.

Linux поддерживает работу нескольких процессоров в одной системе.

*Привязка процессов к процессору* – это вероятность, что процесс будет постоянно запланирован на один и тот же процессор. Термин *«мягкая привязка»* означает естественное для планировщика стремление продолжать выполнение процесса на одном и том же процессоре. Планировщик Linux старается запланировать одни и те же процессы на одни и те же процессоры насколько возможно долго, перемещая процессы с одного CPU на другой только в крайнем случае, когда дисбаланс нагрузки очень велик. Это позволяет планировщику минимизировать эффекты кэширования, возникающие при миграции, при одновременной гарантии, что все процессоры в системе загружены примерно одинаково.

Принцип разработки современных многопроцессорных систем таков, что кэши, связанные с каждым процессором, независимы друг от друга и самостоятельны. Таким образом, данные в кэше одного процессора совсем иные, чем другого. Когда процесс перемещается с одного процессора на другой, получается два взаимосвязанных источника затрат: кэшированные данные, более недействительные для перемещенного процесса, и данные в оригинальном кэше процесса, которые должны быть объявлены недействительными.

Иногда, однако, пользователь или приложение хочет принудительно связать процесс с каким-либо процессором. Чаще всего это обусловлено высокой чувствительностью процесса к кэшированию, из-за чего процессу лучше выполняться на одном и том же процессоре. Привязывая процесс к определенному процессору и поддерживая эту связь на уровне ядра, мы обеспечиваем *жесткую привязку*.

2. Ознакомьтесь с системными вызовами *sched\_getaffinity()* и *sched\_setaffinity()*.

Linux предоставляет два системных вызова для получения и установки жесткой привязки процесса:

#define \_GNU\_SOURCE

#include <sched.h>

typedef struct cpu\_set\_t;

size\_t CPU\_SETSIZE;

void CPU\_SET (unsigned long cpu, cpu\_set\_t \*set);

void CPU\_CLR (unsigned long cpu, cpu\_set\_t \*set);

int CPU\_ISSET (unsigned long cpu, cpu\_set\_t \*set);

void CPU\_ZERO (cpu\_set\_t \*set);

int sched\_setaffinity (pid\_t pid, size\_t setsize, const cpu\_set\_t \*set);

int sched\_getaffinity (pid\_t pid, size\_t setsize, cpu\_set\_t \*set);

Вызов *sched\_getaffinity()* получает привязку к CPU процесса с идентификатором *pid* и сохраняет ее в специальном типе *cpu\_set\_t*, куда можно получить доступ через специальный макрос. Если *pid* равен *0*, вызов возвращает привязку текущего процесса. Параметр *setsize* определяет размер типа *cpu\_set\_t*, который может быть использован для обеспечения совместимости при изменении размера типа, возможного в будущем. В случае успеха *sched\_getaffinity()* возвращает *0*; в случае неудачи он возвращает *–1* и устанавливает переменную *errno*.

3. Наберите, от компилируйте и запустите код:

cpu\_set\_t set;

int ret, i;

CPU\_ZERO (&set);

ret = sched\_getaffinity (0, sizeof (cpu\_set\_t), &set);

if (ret == –1)

perror ("sched\_getaffinity");

for (i = 0; i <CPU\_SETSIZE; i++) {

int cpu;

cpu = CPU\_ISSET (i, &set);

printf ("cpu=%i is %s\n", i, cpu ? "set" : "unset");

}

Перед выполнением используется *CPU\_ZERO* для обнуления всех битов в наборе *set*. Затем пробегаемся по набору от *0* до *CPU\_SETSIZE*. Для проверки, привязан ли данный процессор системы, *i,* к нашему процессу используется *CPU\_ISSETN*. Макрос возвращает *0*, если процессор не привязан, и ненулевую величину, если это так.

4. Чему равен *CPU\_SETSIZE*? Информация о каком количестве процессоров выведена на экран? Сколько из них находятся в состоянии *«set»(установлены)*?

5. Поменяйте в настройках виртуальной машины количество используемых процессоров на 2. Запустите программу определения установленных процессоров. Изменился ли результат ее работы? Какое максимальное количество процессоров можно подключить в виртуальной машине?

6. Проверьте работу следующего программного кода, осуществляющего привязку процесса к процессору 0:

cpu\_set\_t set;

int ret, i;

CPU\_ZERO (&set); /\* очистить набор процессоров \*/

CPU\_SET (0, &set); /\* разрешить процессор 0 \*/

CPU\_CLR (1, &set); /\* запретить процессор 1 \*/

ret = sched\_setaffinity (0, sizeof (cpu\_set\_t), &set);

if (ret == –1)

perror ("sched\_setaffinity");

for (i = 0; i <CPU\_SETSIZE; i++) {

int cpu;

cpu = CPU\_ISSET (i, &set);

printf ("cpu=%i is %s\n", i, cpu ? "set" : "unset");

}

Как обычно, мы начинаем с обнуления набора с помощью *CPU\_ZERO*. Затем мы устанавливаем процессор *0* посредством *CPU\_SET* и отбрасываем (очищаем) процессор 1 с помощью *CPU\_CLR*. Операция *CPU\_CLR*, по сути дела, здесь не нужна, так как мы только что обнулили весь набор, но все же выполним ее для законченности.

7. Будет ли отличаться результат работы программы из п. 6 по сравнению с программой из п. 3 данного задания?

8. Выполните привязку выполнения процесса на процессор 3. Убедитесь в правильности изменений.

**Задание 4**. Изучите особенности использования политик планирования процессов в Linux.

Ход выполнения:

1. Прочитайте вводную информацию о политиках планирования в Linux.

Поведение планировщика Linux по отношению к процессам зависит от политики планирования процесса, известной также как класс планирования. В дополнение к обычной политике по умолчанию Linux предоставляет две политики планирования процессов реального времени. Предпроцессорный макрос из заголовка *<sched.h>* представляет каждую из них: *SCHED\_FIFO*, *SCHED\_RR* и *SCHED\_OTHER*.

***Класс FIFO*** (first in, first out – «первый вошел – первый вышел») – очень простая политика реального времени без использования квантов времени. Процесс класса FIFO будет выполняться, пока не появится другой готовый к работе процесс с более высоким приоритетом. Класс FIFO представлен макросом *SCHED\_FIFO*. Фактически можно сказать, что процессы FIFO класса выполняются столько времени, сколько хотят, если они являются процессами с самым высоким приоритетом в системе.

***Класс RR*** (round-robin — «карусель») идентичен FIFO за исключением того, что он включает дополнительные правила для процессов, имеющих равные приоритеты. Этот класс представляется макросом *SCHED\_RR*.

Планировщик назначает квант времени каждому процессу RR-класса. Когда процессы класса RR исчерпают свой квант, планировщик передвигает их в конец списка процессов того же приоритета. Таким образом, RR-процессы данного приоритета планируются как бы в виде карусели. Если существует только один процесс данного приоритета, RR-класс идентичен FIFO. В этом случае, когда квант исчерпан, процесс просто немедленно продолжает выполнение.

***Обычная политика***. Макрос *SCHED\_OTHER* представляет стандартную политику планирования, класс без поддержки реального времени, используемый по умолчанию. Все процессы обычного класса имеют статический приоритет, равный *0*. Следовательно, все работоспособные процессы класса FIFO или RR будут замещать работоспособные процессы обычного класса.

Планировщик использует значение любезности, обсуждавшееся выше, чтобы расставить приоритеты процессов внутри обычного класса. Значение любезности не влияет на статический приоритет, который остается равным 0.

2. Прочитайте информацию о системных вызовах *sched\_getscheduler()* и *sched\_setscheduler()*:

#include <sched.h>

struct sched\_param {

/\* ... \*/

int sched\_priority;

/\* ... \*/

};

int sched\_getscheduler (pid\_t pid);

int sched\_setscheduler (pid\_t pid,

int policy,

const struct sched\_param \*sp);

Успешный вызов *sched\_getscheduler()* возвращает политику планирования процесса, определенного через *pid*. Если *pid* равен *0*, вызов возвращает политику планирования вызывающего процесса. Величина целого типа, указанная в *<sched.h>*, представляет политику планирования: «первый вошел – первый вышел» обозначена константой SCHED\_FIFO, политика карусели – SCHED\_RR, а обычная политика – SCHED\_OTHER. В случае ошибки вызов возвращает *–1* (политика планирования никогда не может быть представлена этим числом), и *errno* заполняется соответствующим образом.

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.

3. Наберите следующий программный код и проанализируйте его работу:

int policy;

/\* получаем нашу политику планирования\*/

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:

perror ("sched\_getscheduler");

break;

default:

fprintf(stderr, "Неизвестная политика!\n");

}

4. Установите политику планирования процесса и параметры:

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

int ret;

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

if (ret == –1) {

perror ("sched\_setscheduler");

return 1;

}

Данный фрагмент кода устанавливает политику карусели со статическим приоритетом *1* в качестве политики планирования вызывающего процесса.

5. Требуются ли права администратора для изменения политики планирования?

6. Проверьте, действительно ли была установлена политика карусели.

7. Поменяйте программный код так, чтобы он устанавливал политику планирования процессов FIFO.

**Задание 5**. Изучите особенности установки параметров планирования процессов в Linux.

Ход выполнения:

1. Изучите особенности использования интерфейсов *sched\_getparam()* и *sched\_setparam()*.

Интерфейсы *sched\_getparam()* и *sched\_setparam()*, определенные POSIX, используются для получения и установки параметров, связанных с уже выставленной политикой планирования:

#include <sched.h>

struct sched\_param {

/\* ... \*/

int sched\_priority;

/\* ... \*/

};

int sched\_getparam (pid\_t pid, struct sched\_param \*sp);

int sched\_setparam (pid\_t pid, const struct sched\_param \*sp);

Интерфейс *sched\_getscheduler()* возвращает только политику планирования без связанных с ней параметров. Вызов *sched\_getparam()* возвращает через *sp* параметры планирования процесса с идентификатором *pid*.

Если *pid* равен *0*, возвращаются параметры вызывающего процесса. В случае успеха вызов возвращает *0*. При неудаче возвращается *–1* и *errno* заполняется соответствующим образом.

Поскольку *sched\_setscheduler()* также устанавливает некоторые сопредельные параметры планирования, *sched\_setparam()* полезен только для изменения параметров в дальнейшем.

В случае успеха параметры планирования процесса, определенного через *pid*, устанавливаются согласно указанному в *sp* и вызов возвращает *0*. При неудаче вызов возвращает *–1*, а *errno* заполняется соответствующим образом.

2. Наберите, откомпилируйте и запустите следующий программный код. Какие операции в нем выполняются?

struct sched\_param sp;

int ret;

ret = sched\_getparam (0, &sp);

if (ret == –1) {

perror ("sched\_getparam");

return 1;

}

printf ("Приоритет равен %d\n", sp.sched\_priority);

3. Наберите, откомпилируйте и запустите следующий программный код. Какие операции в нем выполняются?

struct sched\_param sp;

int ret;

sp.sched\_priority = 1;

ret = sched\_setparam (0, &sp);

if (ret == –1) {

perror ("sched\_setparam");

return 1;

}

4. Запустите два фрагмента кода (из п.2 и из п.3) по порядку. Что будет получено на выходе?

**Задание 6**. Изучите особенности определения диапазона допустимых значений приоритета в Linux.

Ход выполнения:

1. Изучите системные вызовы для получения диапазона допустимых значений.

В предыдущих примерах в системных вызовах жестко прописывались значения приоритета. POSIX не гарантирует, что какие-то приоритеты планирования непременно будут существовать в данной системе, кроме указания, что между минимальной и максимальной величинами должно быть как минимум 32 приоритета.

Linux предоставляет два системных вызова для получения диапазона допустимых значений. Один возвращает минимальное значение, другой – максимальное:

#include <sched.h>

int sched\_get\_priority\_min (int policy);

int sched\_get\_priority\_max (int policy);

В случае успеха вызов *sched\_get\_priority\_min()* возвращает минимальное, а *sched\_get\_priority\_max()* – максимальное допустимое значение приоритета, связанное с политикой планирования, определенной в *policy*. При неудаче оба вызова возвращают *–1*.

2. Напишите, запустите следующий программный код. Проанализируйте его работу.

int min, max;

min = sched\_get\_priority\_min (SCHED\_RR);

if (min == –1) {

perror ("sched\_get\_priority\_min");

return 1;

}

max = sched\_get\_priority\_max (SCHED\_RR);

if (max == –1) {

perror ("sched\_get\_priority\_max");

return 1;

}

printf ("Диапазон приоритетов для SCHED\_RR равен %d - %d\n", min, max);

3. Определите диапазоны приоритетов для обычной политики и политики FIFO.

4. Присвойте процессу высший приоритет в его политике планирования:

int set\_highest\_priority (pid\_t pid)

{

struct sched\_param sp;

int policy, max, ret;

policy = sched\_getscheduler (pid);

if (policy == –1)

return –1;

max = sched\_get\_priority\_max (policy);

if (max == –1)

return –1;

memset (&sp, 0, sizeof (struct sched\_param));

sp.sched\_priority = max;

ret = sched\_setparam (pid, &sp);

return ret;

}

5. Какая политика планирования была у процесса в п.4? Какое значение приоритета получил процесс?

6. Назначьте процессу политику планирования FIFO. Назначьте ему высший приоритет и проверьте правильность результата.

**Задание 7**. Изучите функцию определения размера временного кванта процесса.

Ход выполнения:

1. Ознакомьтесь с назначением и особенностями использования функции *sched\_rr\_get\_interval()*.

Как уже обсуждалось ранее, процессы SCHED\_RR ведут себя так же, как и SCHED\_FIFO, кроме того, что планировщик назначает этим процессам кванты времени.

POSIX определяет интерфейс для получения размера кванта времени данного процесса:

#include <sched.h>

struct timespec {

time\_t tv\_sec; /\* секунды \*/

long tv\_nsec; /\* наносекунды \*/

};

int sched\_rr\_get\_interval (pid\_t pid, struct timespec \*tp);

В случае успеха вызов *sched\_rr\_get\_interval()* сохраняет в структуре *timespec*, указанной *tp*, продолжительность кванта времени процесса, определенного через *pid*, и возвращает *0*. При неудаче вызов возвращает *–1*, а *errno* присваивается соответствующее значение.

2. Наберите, откомпилируйте и запустите следующий программный код. Для какой политики будет определен размер кванта?

struct timespec tp;

int ret;

/\* получение длины временного кванта текущей задачи \*/

ret = sched\_rr\_get\_interval (0, &tp);

if (ret == –1) {

perror ("sched\_rr\_get\_interval");

return 1;

}

/\* преобразование секунд и наносекунд в миллисекунды \*/

printf("Наш временной квант равен %.2lfmilliseconds\n",

(tp.tv\_sec \* 1000.0f) + (tp.tv\_nsec / 1000000.0f));

3. Определите размеры квантов для остальных политик планирования.

**Задание 8**. Изучите функцию определения размера временного кванта процесса.

Ход выполнения:

1. Ознакомьтесь с информацией о лимитах ресурсов в операционной системе Linux.

Ядро Linux накладывает на процессы ряд лимитов ресурсов. Они достаточно жестко ограничивают количество ресурсов ядра, которые может использовать процесс, — это, например, количество открытых файлов, страниц памяти, ожидающих сигналов, и т. д. Лимиты строго соблюдаются. Ядро не допускает никаких действий, в результате которых они будут нарушены.

Linux предоставляет два системных вызова для управления лимитами ресурсов. Оба интерфейса стандартизированы POSIX, но Linux поддерживает несколько лимитов в дополнение к прописанным в стандарте. Пределы можно проверить с помощью *getrlimit()* и установить посредством *setrlimit()*:

#include <sys/time.h>

#include <sys/resource.h>

struct rlimit {

rlim\_t rlim\_cur; /\* мягкий лимит \*/

rlim\_t rlim\_max; /\* жесткий лимит \*/

};

int getrlimit (int resource, struct rlimit \*rlim);

int setrlimit (int resource, const struct rlimit \*rlim);

Структура *rlimit* представляет имеющиеся пределы. Структура определяет два ограничения: ***мягкий лимит*** и ***жесткий лимит***. Ядро соблюдает мягкие лимиты ресурсов для процессов, но сам процесс может свободно менять свой мягкий лимит, присваивая ему любую величину от 0 до значения жесткого лимита.

Все ресурсные лимиты имеют два особых значения: *0* и *бесконечность*. Первое из них вообще исключает возможность использования данного ресурса. Бесконечность обозначается ядром с помощью присваивания RLIM\_INFINITY значения *–1* (что может вызвать некоторую путаницу, так как *–1* – это еще и значение, возвращаемое при ошибке). Если RLIMIT\_CORE равно бесконечности, ядро может создавать файлы ядра любого размера. Функция *getrlimit()* помещает текущие мягкий и жесткий лимиты для ресурса, определенного как *resource*, в структуру, определенную как *rlim*. В случае успеха вызов возвращает *0*. При неудаче возвращается *–1* и переменная *errno* принимает соответствующее значение.

Соответственно, функция *setrlimit()* устанавливает мягкий и жесткий лимиты, связанные с *resource*, равными величинам, на которые указывает *rlim*. В случае успеха вызов возвращает *0* и ядро обновляет пределы ресурсов, как было указано. В случае неудачи вызов возвращает *–1* и переменная *errno* принимает соответствующее значение.

Linux в настоящее время предоставляет 16 ресурсных лимитов.

Лимиты по умолчанию, доступные процессу, зависят от трех переменных: начального мягкого лимита, начального жесткого лимита и администратора вашей системы. Ядро предписывает мягкий и жесткий лимиты; они перечислены в табл.1. Ядро устанавливает эти лимиты для процесса *init*, и, поскольку все дочерние процессы наследуют свои лимиты от родительских, все последующие процессы наследуют мягкие и жесткие лимиты *init*.

Табл. 1. Лимиты ресурсов по умолчанию.

| Лимит ресурса | Мягкий лимит | Жесткий лимит | Описание |
| --- | --- | --- | --- |
| RLIMIT\_AS | RLIM\_INFINITY | RLIM\_INFINITY | максимальный размер адресного пространства процесса в байтах |
| RLIMIT\_CORE | 0 | RLIM\_INFINITY | максимальный размер файлов ядра в байтах |
| RLIMIT\_CPU | RLIM\_INFINITY | RLIM\_INFINITY | максимальное процессорное время, которое может использовать процесс, в секундах |
| RLIMIT\_DATA | RLIM\_INFINITY | RLIM\_INFINITY | максимальный размер сегмента данных и кучи процесса в байтах |
| RLIMIT\_FSIZE | RLIM\_INFINITY | RLIM\_INFINITY | максимальный размер в байтах файла, который может создать процесс |
| RLIMIT\_LOCKS | RLIM\_INFINITY | RLIM\_INFINITY | максимальное количество блокировок файлов, которые может держать процесс |
| RLIMIT\_MEMLOCK | 8 страниц | 8 страниц | максимальное количество байтов памяти, которое процесс может заблокировать в памяти |
| RLIMIT\_MSGQUEUE | 800 Кбайт | 800 Кбайт | максимальное количество байтов, которое пользователь может выделить для серии сообщений POSIX |
| RLIMIT\_NICE | 0 | 0 | максимальное значение, до которого процесс может снизить свое значение любезности (то есть повысить приоритет) |
| RLIMIT\_NOFILE | 1024 | 1024 | значение, на единицу превышающее максимальное число файловых дескрипторов, которые процесс может держать открытыми |
| RLIMIT\_NPROC | 0 (означает отсутствие ограничений) | 0 (означает отсутствие ограничений) | максимальное количество процессов, которое может выполняться в каждый момент времени от имени одного пользователя |
| RLIMIT\_RSS | RLIM\_INFINITY | RLIM\_INFINITY | максимальное количество страниц памяти, которое процессор может выделить |
| RLIMIT\_RTTIME |  |  | лимит процессорного времени в микросекундах, которое может использовать процесс реального времени без блокирующего системного вызова |
| RLIMIT\_RTPRIO | 0 | 0 | максимальный уровень приоритета процесса реального времени, который тот может запросить |
| RLIMIT\_SIGPENDING | 0 | 0 | максимальное количество сигналов, которые могут находиться в очереди для данного пользователя |
| RLIMIT\_STACK | 8 Мбайт | RLIM\_INFINITY | максимальный размер стека процесса в байтах |

Эти лимиты могут быть изменены двумя путями.

* Любой процесс может свободно повысить мягкий лимит от 0 до значения жесткого лимита или снизить жесткий. Дочерние процессы наследуют во время ветвления обновленные величины лимитов.
* Привилегированные процессы могут устанавливать любое значение жесткому лимиту. Дочерние процессы наследуют во время ветвления обновленные величины лимитов.

2. Определите величину ресурсного лимита:

struct rlimit rlim;

int ret;

/\* получение лимитов на размеры файлов ядра \*/

ret = getrlimit (RLIMIT\_CORE, &rlim);

if (ret == –1) {

perror ("getrlimit");

return 1;

}

printf ("Лимиты RLIMIT\_CORE: мягкий=%ld жесткий=%ld\n",

rlim.rlim\_cur, rlim.rlim\_max);

3. Установите максимальный размер файлов ядра 32 Мбайт:

struct rlimit rlim;

int ret;

rlim.rlim\_cur = 32 \* 1024 \* 1024; /\* 32 Мбайт \*/

rlim.rlim\_max = RLIM\_INFINITY; /\* не менять \*/

ret = setrlimit (RLIMIT\_CORE, &rlim);

if (ret == –1) {

perror ("setrlimit");

return 1;

}

4. Определите величину ресурсного лимита RLIMIT\_RTTIME.

5. Поменяйте величину любого ресурсного лимита, кроме максимального размера файлов ядра.