

Implementação *system calls* no kernel linux

V. 4.13.12

João Paulo de Oliveira joaopaulodeoliveira123@gmail.com
Lucas Rossi Rabelo lucasrossi98@hotmail.com
Matheus Pimenta Reis matheuspr96@hotmail.com

12 de dezembro de 2017

Sumário

1	Introdução	3
2	Implementação System Calls no Linux	3
	Download do Kernel	4
	Compilação do kernel	5
	Criação da System Call	6
	Tempo na CPU	6
	Tempo de vida do processo	8
	Nº de vezes que o processo passou pela CPU	8
	Retornar -1 caso o processo não exista	9
	Código da system call	10
	Compilação parcial do Kernel	11
	Programa usuário da system call	11
	Conclusão	12

1 Introdução

Este relatório visa mostrar o processo de criação de uma system call para o sistema Linux cujo a função da mesma é obter:

- O tempo na qual um processo passou executando na CPU;
- Quantas vezes um processo passou pela CPU;
- O tempo desde a criação de um processo até o momento de execução da system call(Tempo de vida do processo).

O parâmetro necessário para obter tais informações, é o PID de um processo (Process Identifier ou Identificador do Processo).

Foi utilizada como base para a criação dessa system call, a versão 14.13.12 do kernel do Linux.

2 Implementação System Calls no Linux

As *System Calls* fazem o interfaceamento entre o hardware e os processo do espaço de usuário, elas também servem para três propósitos principais:

1. Provém abstração com o hardware para que o usuário tenha um maior rendimento. Por exemplo, o usuário não se preocupa com o tipo de partição em que ele lerá um arquivo;
2. As *system calls* garantem a segurança e estabilidade para que um usuário relativamente leigo possa ter um alto rendimento com gerência de permissão, usuários e outros critérios de gerência do kernel;
3. Uma camada entre espaço de usuário e o resto do sistema permite fornece o sistema virtualizado para os processos.

As chamadas syscalls em linux são, na maioria dos casos, acessadas pelas funções definidas na C library. As syscalls em si retornam um valor do tipo long4 que, quando negativo, em geral, denota um erro, já o retorno com valor zero (nem sempre) é um sinal de sucesso. A C library, quando uma *system call* retorna um erro, ela grava o código desse erro na variável global errno, que pode ser traduzida para um texto que fala sobre o erro através de funções de biblioteca.

Para a implementação de uma *system call* deve-se ter acesso ao código do kernel do sistema operacional para tanto, foi escolhido o kernel Linux que é open source.

Download do Kernel

Por ser open source, o código fonte do kernel do Linux é mantido online para livre acesso no GitHub(<https://github.com/torvalds/linux>), e também em The Linux Kernel Archives (<https://www.kernel.org/>) em várias versões e formatos de arquivos(compactados). A versão mais recente dado a data de início do TCD foi a versão **4.14.4**. de gerência do kernel. A Free Software Foundation mantém, gerencia e presta suporte para o The Linux Kernel Archives.O download do kernel pode ser feito facilmente, além de outras funções com perguntas frequentes, download de versões em teste (beta) para ser compilado em qualquer distribuição linux ou em outras plataformas que suportam o kernel. Como pode ser visto na imagem abaixo:



Figura 1: The Linux Kernel Archives: Local de download do Kernel

Compilação do kernel

Nesta seção vamos compilar o Kernel que previamente fizemos o Download para trabalhar com ele e assim poder criar nossas System Calls. Primeiramente precisamos instalar na nossa máquina um conjunto de ferramentas que nos permitirá compilar nosso kernel, basta abrir seu terminal e digitar o seguinte comando:

```
sudo apt-get install libncurses5dev gcc make git exuberantctags
```

Criamos uma pasta no nosso espaço de trabalho, posteriormente extraímos todo o conteúdo do Kernel com o seguinte comando:

```
tar xvf linux*.tar.xz
```

onde * é a versão do kernel que acabamos de fazer o download, neste caso usamos linux-12.13.12, então ficou

```
tar xvf linux3.17.1.tar.xz
```

A seguir abrimos nossa pasta pelo terminal e acessamos onde está o kernel, fazendo uso do comando cd, que permite acessar um diretório.

Agora iremos compilar nosso kernel completamente, para isso basta usar o comando make, porém existe algumas opções bem úteis para agilizar o processo para isso usamos:

```
make -j4 CONFIG_LOCALVERSION="tutorial".
```

Ao usar o -j4 dizemos que o processo pode usar quantos processadores estejam disponíveis, no caso do exemplo são 4, ou seja, a máquina usada possui 4 processadores para realizar a tarefa, se sua máquina possuir 6, 8, etc você pode substituir pelo número de processadores da sua máquina, ou seja, jX, sendo x o número de processadores da máquina, vale ressaltar se existe a necessidade de usar a máquina durante o processo é recomendável deixar ao menos 1 processador para realizar atividades. Já o "tutorial" será o nome atribuído ao kernel, que receberá após ser compilado e que será acessado quando estiver iniciando o ubuntu, pode-se alterar para o nome desejado e fica entre , pois é um conjunto de caracteres, este processo pode demorar , pois é um processo extremamente grande, irá depender também da velocidade de processamento da máquina, em testes feitos normalmente demorou entre 2 a 4 horas usando 4 processadores.

Quando por fim a compilação acabar precisa-se substituir o kernel para assim que todas as modificações que serão feitas posteriormente possam ser carregadas com o sistema. Para isso abrir o terminal digir-se na pasta anteriormente criada conforme a imagem anterior, e proseguimos com os seguintes comandos:

```
make modules
```

```
sudo make modules_install
```

E por fim instala-se o novo kernel, este comando instala o Kernel, gera a imagem, copia os arquivos para o diretório /boot e atualiza o gerenciador de boot.

```
sudo make install
```

Agora está tudo pronto para acessar o novo kernel usamos terminal com comando:

```
reboot
```

Quando estiver na opção de escolher o Sistema Operacional basta acessar opções avançadas do ubuntu e selecionar o kernel, que estará identificado pelo nome inserido no começo, vale resaltar que quando você selecionar o kernel uma vez não é necessário selecionar-lo toda vez que der boot, a máquina carrega o último kernel instalado.

Criação da System Call

A *system call* foi criada dentro na pasta kernel no diretório principal deixando o arquivo `scall.c` nessa pasta. Após isso foi adicionado o arquivo objeto no arquivo de *makefile* como descrito no tutorial, dessa forma, foi encontrada no arquivo `syscall_64.tbl` na forma [5]

333 common scall sys_det

Assim a função pode ser chamada pela função ***syscall(333)*** presente na *unistd.h*. Por fim, foi adicionado o cabeçalho da função no arquivo *syscall.h*. Depois disso foi compilado o kernel seguindo o tutorial proposto [2].

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
```

Vejamos agora o código do kernel utilizado para:

Tempo na CPU

Como primeira tentativa para obter o tempo de CPU de um processo, acessamos a variável *sum_exec_runtime* na estrutura *task_cputime* [1], com base nos comentários da estrutura, concluímos que tal variável era o que desejávamos como segue na figura 2:

```

/**
 * struct task_cputime - collected CPU time counts
 * @utime:           time spent in user mode, in nanoseconds
 * @stime:           time spent in kernel mode, in nanoseconds
 * @sum_exec_runtime: total time spent on the CPU, in nanoseconds
 *
 * This structure groups together three kinds of CPU time that are tracked for
 * threads and thread groups. Most things considering CPU time want to group
 * these counts together and treat all three of them in parallel.
 */
struct task_cputime {
    u64                utime;
    u64                stime;
    unsigned long long sum_exec_runtime;
};

/* Alternate field names when used on cache expirations: */
#define virt_exp      utime
#define prof_exp      stime
#define sched_exp     sum_exec_runtime

```

Figura 2: Estrutura task_cputime

No entanto, com a tentativa não obtivemos o resultado esperado. como segunda tentativa, encontramos uma função chamada `get_sum_exec_runtime`. A partir de ai decidimos usar o que a função fornece, ou seja, chamar direto na propria task o desejado usando esse "su" que posteriormente seria a `shed_entity` como segue a imagem:

```

struct sched_entity {
    /* For load-balancing: */
    struct load_weight    load;
    struct rb_node        run_node;
    struct list_head      group_node;
    unsigned int          on_rq;

    u64                   exec_start;
    u64                   sum_exec_runtime;
    u64                   vruntime;
    u64                   prev_sum_exec_runtime;

    u64                   nr_migrations;

    struct sched_statistics statistics;
};

```

Figura 3: Parte da estrutura sched_entity

Que é o resultado que procurávamos, decidimos usar direto pois chamar

uma função pode ser menos eficiente que chama-la na própria `task_struct` assim obtemos em nano-segundos o tempo que o processo passou pela CPU.

Tempo de vida do processo

O tempo de vida do processo não pode ser extraído diretamente, assim a estratégia adotada foi usar a variável interna à `task_struct`, a `start_time` presente na `shed.h`, que é o tempo em nanosegundos contando a partir do boot da máquina (Monotonic) [4]. A outra variável usada foi a função `ktime_get_boottime()`

```
725
726
727
/* Monotonic time in nsecs: */
u64 start_time;
```

Figura 4: `start_time`: Tempo de vida do processo em nanosegundos contando a partir do boot

presente em `linux/timekeeping.h`. Essa função retorna, intuitivamente, o `boot time` que é o tempo em que a máquina está ligada, desde o boot até o momento atual. Tendo essas duas variáveis, a `start_time` foi subtraída do retorno da função `ktime_get_boottime()`, como se segue na figura. Assim foi possível obter o tempo de vida do processo.

```
life = ktime_get_boot_ns() - task->start_time;
```

Figura 5: Tempo de vida do processo em nanosegundos presente na `system call`

Nº de vezes que o processo passou pela CPU

Verificando a `task_struct` percebemos que tinha uma estrutura que nos auxiliaria chamada `shed_info` como segue na figura 5:

basta retornar o `pcount` que obteremos o resultado desejado


```

struct sched_info {
#ifdef CONFIG_SCHED_INFO
    /* Cumulative counters: */

    /* # of times we have run on this CPU: */
    unsigned long                pcount;

```

Figura 6: Sched_info

Retornar -1 caso o processo não exista

A função *pid_task* retorna um ponteiro para *task_struct* dado um PID caso o PID seja zero ou ele não exista, a função retorna NULL [3].

```

436 struct task_struct *pid_task(struct pid *pid, enum pid_type type)
437 {
438     struct task_struct *result = NULL;
439     if (pid) {
440         struct hlist_node *first;
441         first = rcu_dereference_check(hlist_first_rcu(&pid->tasks[type]),
442                                     lockdep_tasklist_lock_is_held());
443         if (first)
444             result = hlist_entry(first, struct task_struct, pids[(type)].node);
445     }
446     return result;
447 }
448 EXPORT_SYMBOL(pid_task);
449

```

Figura 7: Estrutura da função que retorna um ponteiro para *task_struct* dado um PID

Assim, na *system call* foi colocada uma condição na qual, caso a função retorne NULL, a *system call* retornará -1 em todas as variáveis de retorno.

Código da system call

Segue o código:

```
#include <linux/linkage.h>      //Syscall espera argumento da pilha
#include <linux/kernel.h>      //Printk
#include <linux/pid.h>          //pid_task(), PIDTYPE_PID
#include <linux/sched.h>        //task_struct
#include <linux/uaccess.h>      //copy_to_user
#include <linux/timekeeping.h>  //ktime_get_boottime()

asmlinkage long sys_det(int pid, long int *n_in_CPU, long int *CPU_time, long long int *lifetime){
    struct task_struct *task = NULL;          //inicialização do ponteiro para a estrutura
    long long int life;                       //inicialização do da variavel do tempo de vida
    task = pid_task(find_vpid(pid), PIDTYPE_PID); //Captura o ponteiro para task_struct através do pid pa

    if(task == NULL) {                        //se a task_struct não existir
        int err = -1;                         //variável de retorno caso haja erro
        if(copy_to_user(n_in_CPU ,&err,sizeof(int)))
            return -1;

        if(copy_to_user(lifetime ,&err,sizeof(int)))
            return -1;
        if(copy_to_user(CPU_time ,&err,sizeof(int)))
            return -1;
        return -1;
        //retorna -1 em todas as variáveis de entrada e no retorno da função
    }

    life = ktime_get_boot_ns() - task->start_time;
    //tempo que a maquina está ligada em ns - o momento que o processo foi iniciado contando apartido do bo

    if(copy_to_user(n_in_CPU ,&(task->sched_info.pcount),sizeof(long int))) //retorna 0 quando copia para o
        return -1;

    if(copy_to_user(lifetime ,&life,sizeof(long long int)))//retorna 0 quando copia para o user_space com s
        return -1;
    if(copy_to_user(CPU_time ,&(task->se.sum_exec_runtime),sizeof(long int)))//retorna 0 quando copia para
        return -1;
    //dá o print no dmesg
    printk("O pid eh %d\n",task->pid);
    printk("Quantidade de vezes que o processo passa pela CPU: %ld\n",*n_in_CPU);
    printk("Time CPU: %ld\n",*CPU_time);
    printk("Tempo de vida do processo: %lld s\n", *lifetime);
    return 0;                                //retorna 0 caso obtenha sucesso
}
```

Figura 8: Código da system call

Foi usada também a função `copy_to_user` para copiar um bloco de dados do kernel para o espaço de usuário para que a função possa retornar por parâmetro

Compilação parcial do Kernel

Agora basta compilar nossa system call porém, diferente da primeira vez vamos fazer uma compilação parcial do kernel, assim todas nossas alterações feitas na system call serão carregadas para isso usamos o comando:

```
make bzImage
```

após usar este comando basta carregar e instalar os módulos com os seguintes comandos:

```
make modules
```

```
sudo make modules_install
```

```
sudo make install
```

Agora nossa system call está pronta basta reiniciar nossa máquina e estar nossa system call.

Programa usuário da system call

```
#include<stdio.h>
#include<linux/kernel.h>
#include<sys/syscall.h>
#include<unistd.h>
#include<errno.h>
int main(){
    int x,y;
    printf("Digite um pid: ");
    scanf("%d",&y);

    long int ncpu, cputime;
    long long int lifetime=0;

    x = syscall(333, y, &ncpu, &cputime, &lifetime);
    if(x == 0)
        printf("Numero de vezes na CPU: %ld\n Tempo na do processo na CPU: %ld\n Tempo de vida do processo: %lld\n",ncpu, cputime,lifetime);
    else return -1;
}
```

Figura 9: Código para chamar system call

Basta agora compilar nosso programa no nível de usuário e testar nossa system call e conferir os resultados!

Conclusão

Algumas das principais dificuldades que foram encontradas durante a execução deste trabalho foram:

- O tempo necessário para compilar o kernel;
- Algumas restrições do próprio kernel, como por exemplo o fato de não poder utilizar o tipo float;
- Desconhecimento das funções do kernel;
- A constante atualização das versões do kernel.

Embora tenhamos tido algumas dificuldades, a execução desse trabalho mostrou-se gratificante, pois até então, nunca tivemos a necessidade de programar dentro do kernel de um sistema operacional, porém, essa experiência além de nos possibilitar adquirir um conhecimento maior sobre o funcionamento do kernel de um SO, também nos possibilitou evoluir nossos conhecimentos em programação devido a necessidade de pesquisar sobre outros tipos de dados e outras funções que até então, nunca havíamos utilizado ou sequer ouvido falar.

Referências

- [1] Free electrons embedded linux experts, disponível em: <<http://elixir.free-electrons.com/linux/v4.13.12>>.
- [2] System call, disponível em em <<http://www.facom.ufu.br/~rivalino/gbc045/>>
- [3] Kernel: efficient way to find task_struct by pid?, disponível em <<https://stackoverflow.com/questions/8547332/kernel-efficient-way-to-find-task-struct-by-pid>>, jan 2012.
- [4] L. Robert. *Linux Kernel Development*. Pearson Education India.
- [5] A. Rubini and J. Corbet. *Linux device drivers*. "O'Reilly Media, Inc.", 2001.