# 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

## Sumário

## Implementação System Calls no Linux

As System Calls fazer 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 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 14.13.12. de gerência do kernel. A Free Software Foundation mantém, gerencia e presta suporte para o The Linux Kernel Archives. O donwnload 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:

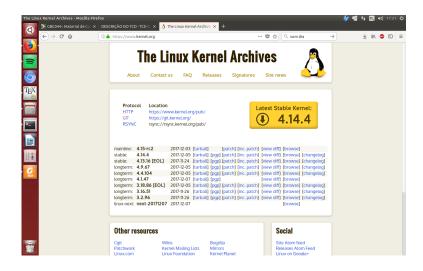


Figura 1: The Linux Kernel Archives: Local de download do Kernel

## 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:

#### 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

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  $task\_cputime$ , 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
            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 {
                                       utime;
       u64
        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 outra estrutura chamada *sched\_entity* na qual, na mesma, encontra-se uma outra variável chamada *sum\_exec\_runtime*, como segue na imagem abaixo:

```
struct sched_entity {
                load-balancing: */
        struct load_weight
                                          load:
        struct rb_node
                                          run node:
        struct list_head
                                          group_node;
        unsigned int
                                          exec_start;
                                          sum_exec_runtime;
vruntime;
        u64
        u64
        u64
                                          prev_sum_exec_runtime;
                                          nr_migrations;
        struct sched_statistics
                                          statistics;
```

Figura 3: Parte da estrutura sched\_entity

Depois procuramos na  $task\_struct$  a declaração de uma variável do tipo  $sched\_entity$ :

562	const struct sched_class	*sched_class;
563	struct <b>sched_entity</b>	se;
564	struct sched_rt_entity	rt;

Figura 4: Declaração da sched\_entity na linha 563 na task\_struct

### Tempo de vida do processo

 $N^{\circ}$  de vezes que o processo passou pela CPU

## 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

## Programa usuário da system call