UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU

Graduação em Ciência da Computação

**Trabalho de Conclusão de Disciplina**

**TCD**

GBC045 – Sistemas Operacionais

Uberlândia

2017



**Trabalho de Conclusão de Disciplina**

**TCD**

Trabalho apresentado à disciplina de Sistemas Operacionais (GBC045), ministrada pelo professor Rivalino Matias Júnior, para o curso de Bacharelado em Ciência da Computação, no período 2017-2, na Universidade Federal de Uberlândia.

**Grupo 04 – Integrantes:**

Antônio Carlos Neto

11611BCC054

Hígor Emanuel Souza Silva

11611BCC016

Marcelo Mendonça Borges

11611BCC020

Uberlândia

2017

**Sumário**

**1.** Introdução ------------------------------------------------------------------------ 4

**2.** Situação Proposta ao Grupo 4 ------------------------------------------------- 4

**3.** Fundamentação Teórica -------------------------------------------------------- 5

**3.1.** Informações básicas para a realização do Projeto ---------------- 5

**3.2.** Funções utilizadas ----------------------------------------------------- 7

**3.3.** Estruturas de Dados utilizadas -------------------------------------- 8

**4.** Fundamentação Prática --------------------------------------------------------- 9

**4.1.** Download e Compilação de um Novo Kernel -------------------- 9

**5.** Implementação da System Call Proposta ---------------------------------- 10

**5.1.** Detalhes da Implementação da System Call --------------------- 10

**5.2.** Programas Auxiliares ----------------------------------------------- 13

**5.3.** Códigos Fontes ------------------------------------------------------- 13

**6.** Conclusão ----------------------------------------------------------------------- 21

**7.** Bibliografia --------------------------------------------------------------------- 22

**1. Introdução**

Este relatório irá apresentar e detalhar o Trabalho de Conclusão de Disciplina realizado pelos alunos integrantes do Grupo 04 da disciplina de Sistemas Operacionais do curso de Bacharelado em Ciência da Computação da Universidade Federal de Uberlândia.

Inicialmente será apresentar alguns conceitos teóricos que servem de base para o desenvolvimento do projeto. Logo em seguida serão apresentados conceitos relativos ao Kernel do Linux, plataforma x86.

A parte principal deste relatório fica por conta da própria implementação da system call proposta, na área de gerenciamento de processos, além de todos os processos de criação de system call para um Kernel que são diretamente envolvidos.

O objetivo principal do projeto será a análise da estrutura de um Kernel de Sistema Operacional com a intenção de analisar e estudar pelo menos uma parte de seu funcionamento, que era de desconhecimento geral dos integrantes do grupo até então.

Além dessa descrição do problema, o relatório também possui uma descrição da instalação de outros Kernel em um Linux, da implementação de uma Syscall no Kernel, e descrição de alguns programas adicionais utilizados para testar a syscall, despois de implementá-la.

**2. Situação Proposta ao Grupo 4**

Área: Gerenciamento de Processos

Objetivo: Análise, projeto e implementação de uma system call para o Kernel Linux (x86).

Este grupo deve projetar uma system call para o Kernel Linux, plataforma x86. A system call deve receber como parâmetro o valor de um PID e retornar:

- O número de processos filhos desse processo identificado pelo PID;

- Para cada processo filho retornar:

- O PID do processo.

- O PPID do processo.

- O UID do processo.

- (-1) se o processo (PID) não existir.

**3. Fundamentação Teórica**

**3.1. Informações básicas para a realização do Projeto**

Uma system call (syscall), ou chamada de sistema em português, é uma função chamada pelos processos em execução do computador que solicita um serviço do Kernel do Sistema Operacional sobre o qual ele está sendo executado.

O conjunto de system calls do Kernel formam uma interface ou camada entre o Kernel mode e o User mode, de modo que essas chamadas padronizam os parâmetros passados entre o processo no User mode para o serviço executado no Kernel mode. A Figura 1 representa o funcionamento geral dessa interface e das system calls em si.

Sabe-se que system calls funcionam no Kernel mode e em contexto de processo. Os serviços executados pelas system calls normalmente realizam tarefas mais complexas e em níveis mais baixos dentro do sistema computacional, normalmente em localidades que o usuário não tem acesso, e por isso, criam uma certa transparência entre os processos em execução, e os serviços das system calls.

As chamadas de sistema podem ser divididas em quatro grupos: chamadas de sistema para gerenciamento de processos, para gerenciamento de diretórios, para gerenciamento de arquivos e chamadas de sistema restantes.



Figura 1 – Funcionamento da Interface de System Call

O bloco de controle de processo (ou PCB - Process Control Block, em inglês), é uma estrutura do Kernel do Sistema Operacional que armazena todas as informações a respeito do processo. Como ele guarda informações críticas sobre os processos é mantido fora do acesso do usuário, e é acessível somente por rotinas do Kernel. O bloco de controle de processo é importante para compilar as informações sobre o processo e ajudar a construir o conceito de processo dentro do sistema operacional.

Normalmente o bloco de controle de processo é implementado como uma lista ligado ou vetor de estruturas, e, além disso, guarda as seguintes informações sobre os processos:

- Identificador do processo (PID);

- Credenciais: UID, GID, etc;

- Registradores da CPU (incluindo o Program Counter);

- Estado do processo;

- Espaço de endereçamento do processo;

- Prioridade do processo;

- Informações sobre o escalonamento do processo;

- Informações de entrada/saída: lista de arquivos abertos, dispositivos de hardware ligados ao processo, etc;

- Ponteiro para o próximo bloco de controle de processo;

Processos Filhos são processos com códigos idênticos ao do Processo Pai do qual eles foram criados. Esses processos são criados utilizando, normalmente uma system call de criação de filhos (fork() para Linux), e o processo Filho criado segue sua execução no mesmo ponto do código em que o pai estava quando ele foi criado, no caso onde está a system call de criação de filhos.

Sobre as informações do processo temos o PID que é o identificador do processo, e que todas vez que um processo é criado o sistema operacional associa um PID à ele, ou seja, mesmo que um mesmo programa seja executado mais de uma vez, cada vez um novo processo é gerado com um novo PID. Também temos o PPID (Parent PID) que é o PID do processo pai desse processo, e UID (User ID) é o identificador do usuário que criou o processo.

**3.2. Funções utilizadas**

Algumas funções específicas foram utilizadas para a realização do projeto. Elas estão listadas a seguir, cada uma com seus respectivos argumentos:

- int printk(const char \*fmt, ...);

*Printk* é a função alternativa utilizada no lugar da *printf* que não pode ser utilizada no Kernel. *Printk* funciona de maneira semelhante ao *printf*, exceto pelo fato da saída ir para um buffer de saída, ao invés da tela. Esse buffer pode ser acessado usando o comando *dmesg*.

- extern struct pid \*find\_get\_pid(int nr);

*Find\_get\_pid* é uma função que retorna um ponteiro para a *struct pid* do processo do PID passado como parâmetro.

- extern struct task\_struct \*pid\_task(struct pid \*pid, enum pid\_type);

*Pid\_task* é uma função que recebe uma *struct pid* e retorna um ponteiro para a *task\_struct* do PID correspondente à *struct pid* passada como parâmetro.

-void list\_for\_each(list \*list, listIterator iterator);

*List\_for\_each* recebe uma lista e um iterador. Ele basicamente percorre a lista seguindo a lógica do iterador.

-#define list\_entry(ptr, type, member) \

container\_of(ptr, type, member)

#define container\_of(ptr, type, member) \

(type\*) ((char \*) (ptr) – (char \*) &((type \*)0)->member)

*List\_entry* essencialmente pega os dados da estrutura.

- copy\_to\_user (coid \_\_user \*to, const void \*from, unsigned long n);

*Copy\_to\_user* é uma forma de passar dados do Kernel level para o User level.

**3.3. Estruturas de Dados utilizadas**

Em relação as estruturas de dados utilizadas, precisávamos das estruturas que estivessem presentes o PID, o PPID e o UID. Assim, ao percorrer o Kernel encontramos o PID e o PPID na estrutura *task\_struct* no arquivo de biblioteca *sched.h* . Já o UID foi encontrado na estrutura *cred* no arquivo de biblioteca *cred.h*.

O PID estava definido do tipo *pid\_t*, que é um *typedef* de *\_\_kernel\_pid\_t*. Mas este é um *typedef* de *int*. Portanto *pid\_*t é *int*.

O PPID é acessado pela estrutura do pai do processo, pegando o PID de sua *task\_struct*.

O UID estava definido *k\_uidt*, que é um *typedef* de uma estrutura que possui um único elemento definido como *uid\_t*. Esse *uid\_t* por sua vez é um *typedef* de *\_\_kernel\_uid32\_t*. Por fim este é um *typedef* de *unsigned int*. Portanto *kuid\_t* é, de certa forma, *unsigned int*.

Juntamente com UID, que é UID real do usuário que criou o processo, é possível observar outros 3 tipos de UID:

- SUID (Saved UID): É o UID que o processo possui quando é criado. Ele é salvo para poder ser utilizado posteriormente;

- EUID (Effective UID): UID para averiguação de privilégios do processo para realizar determinação ação;

- FSUID: UID para operações de Virtual File System VFS.

**4. Fundamentação Prática**

**4.1.** **Download e Compilação de um Novo Kernel**

Para a criação de uma system call, se faz necessário a compilação do kernel, porque quando é recompilado com o arquivo da system call inserido e algumas modificações em certos arquivos, ele passa a reconhecê-la como válida. O kernel a ser compilado pode ser encontrado no site “http://www.kernel.com.br/” e para compila-lo a partir da fonte, você precisará de alguns pacotes como: vim, make, gcc, libssl-dev, chrpath, gawk, texinfo, libsdl1.2-dev, whiptail, diffstat, cpio. Podemos baixar esses pacotes com o comando “sudo apt-get install [pacotes]”, lembrando que esses passos são para as distribuições Ubuntu e Debian.

Depois de baixar o kernel, de preferência estável, no site, devemos descompactar e extrair os arquivos. Para extraí-los é usado o comando “tar xvf  linux­4.14.5.tar.xz”, note que a versão usada é 4.14.5 e o arquivo está com a extensão “tar.xz”, e deve estar na pasta do arquivo para a descompactação. O comando de descompactação gera um diretório do tipo linux-versãodokernel onde eles foram descompactados.

Agora, selecione o diretório recém-criado, e usa no terminal do seu linux o seguinte comando, “make menuconfig”, que é uma interface onde podemos personalizar o kernel para conseguir a melhor performance possível na sua máquina, assim que fizer as alterações que julgar necessária salve como “.menuconfig” (padrão), e saia da interface.

A partir deste momento a compilação do kernel está pronta para acontecer. Usando o comando make e definindo a quantidade de núcleos que serão disponibilizados para a compilação, caso não define será usado apenas um processador, executaremos uma série de comandos. Como observação, verifique se existe espaço no seu disco, pois esse processo pode ocupar cerca de 15GB, dependendo do kernel.

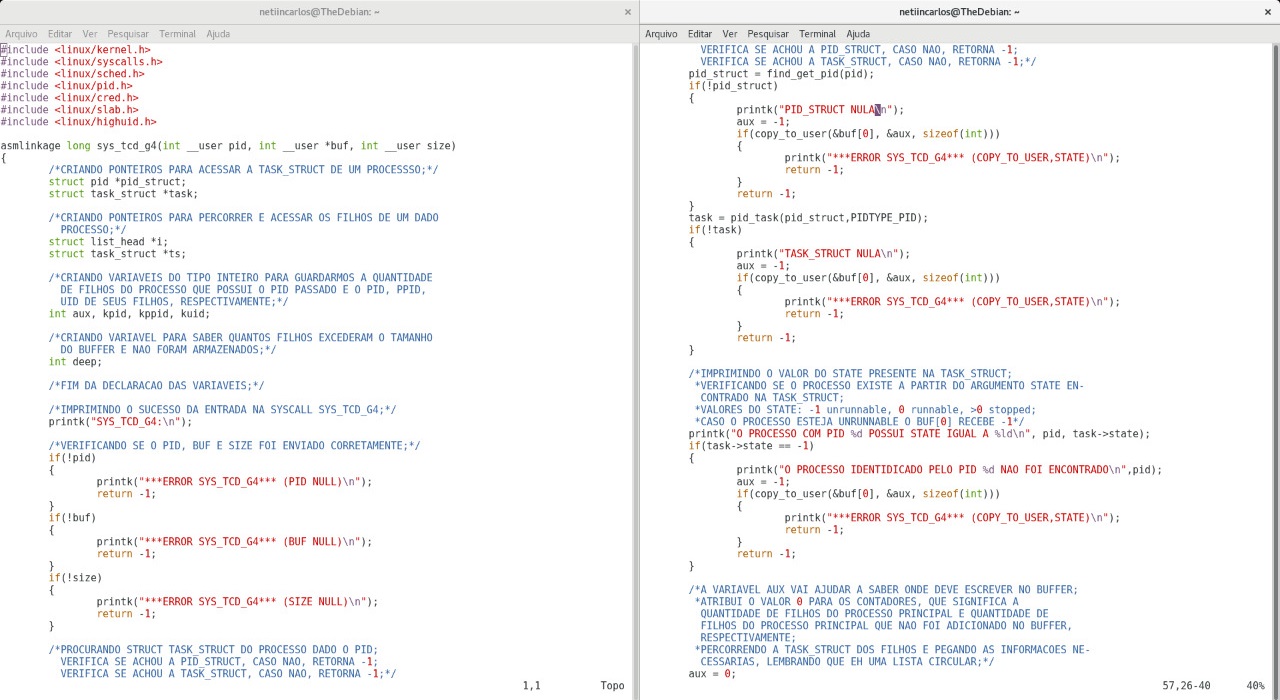
Como primeiro comando, usamos “make -j4” (disponibilizando 4 núcleos), depois usamos “make –j4 modules”, “make –j4 modules\_install”, “make –j4 install”, instalando os módulos e o kernel. Após a execução, que pode demorar dependendo da quantidade de processadores disponibilizado, devemos fazer um reboot para que o sistema operacional inicializa-se com as novas configurações, lembrando de escolher no seu boot-manager a nova versão compilada do kernel.

Tudo feito para a compilação de um novo kernel foi feito a partir do material que o professor Rivalino disponibilizou em seu site. A versão do Kernel utilizada foi a 4.13.12 (stable).

**5. Implementação da System Call Proposta**

**5.1. Detalhes da Implementação da System Call**

Com o intuito de desenvolver uma nova system call, primeiramente criamos um código em formato “.c”, na pasta “~linux-4.13.12/kernel”. Sua programação é um pouco diferente em comparação ao user space, pois a maioria das “funções” são diferentes em alguns aspectos. Depois de ter criado a syscall devemos alterar o “Makefile” daquela pasta, ou seja, na mesma pasta acionamos o comando “vim Makefile” e acrescentamos no final da linha do “obj-y” o nome do arquivo do código da syscall com a extensão “.o”, depois salvamos o arquivo e vamos para próxima etapa, que consiste em entrar na pasta “~linux-4.13.12/arch/x86/entry/syscalls” e abrir o arquivo “syscall\_64.tbl” e colocar no final, na parte de “x64 or x32”, o próximo número como primeiro argumento, common como segundo e completar com as informações da syscall criada, e depois entramos na pasta “~4.13.12/include/linux” e abrimos o arquivo “syscalls.h” e acrescentamos o protótipo da system call, assim como última etapa, compilamos o kernel novamente. Lembrando que essa é uma das formas de criar system call, podendo ser necessário ter que alterar ou criar mais arquivos, por exemplo, se for preciso criar uma “struct”. Segue as imagens, contendo passo a passo os comandos executados e a implementação da nossa syscall:



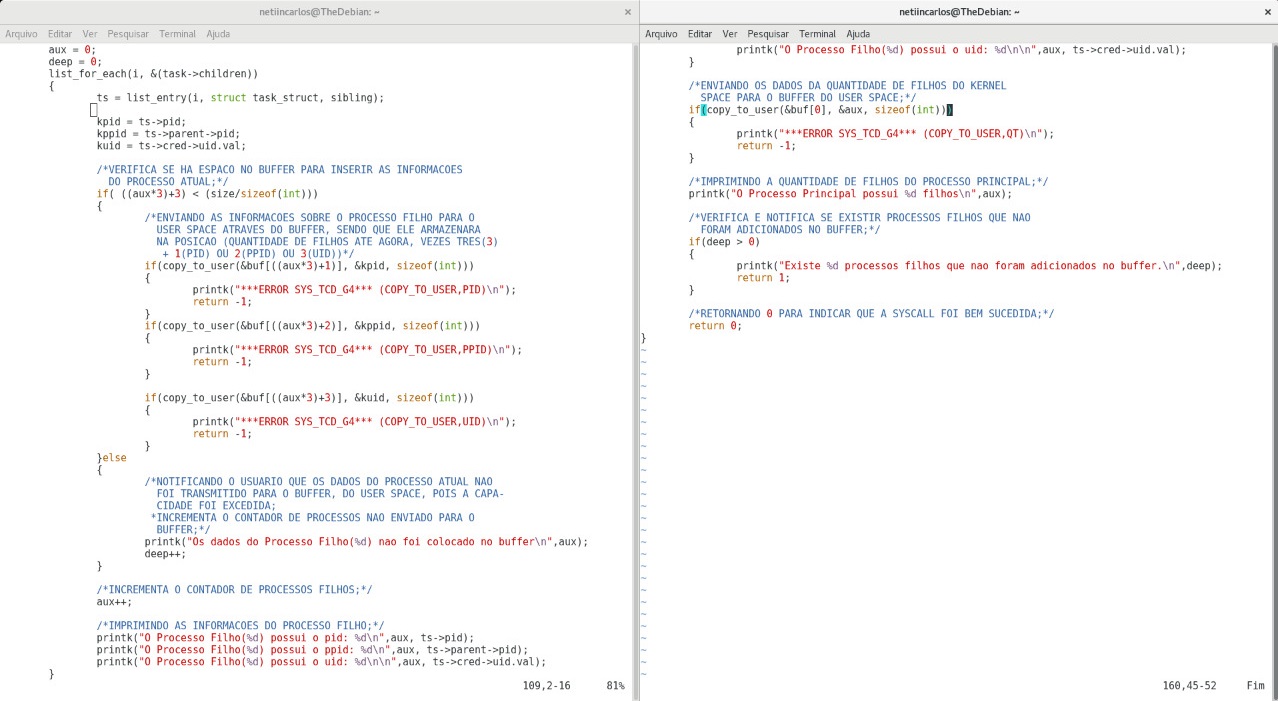


Figura 2: Código da System Call “sys\_tcd\_g4”, arquivo “tcd\_g4,c”

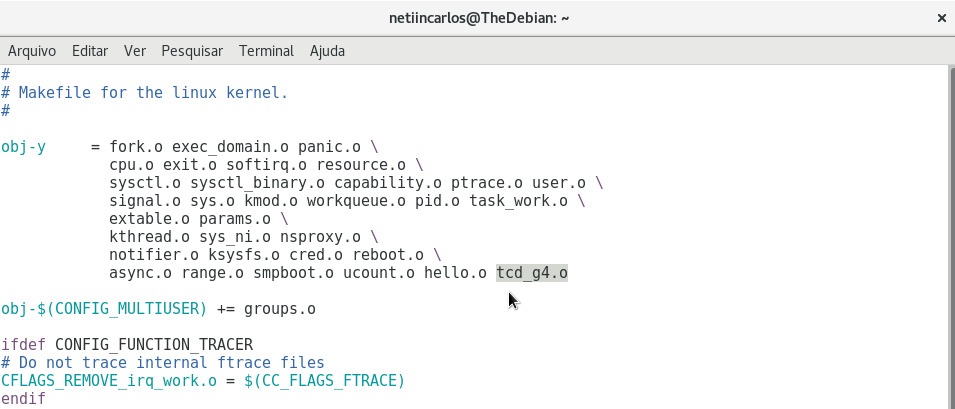


Figura 3: Arquivo “Makefile”

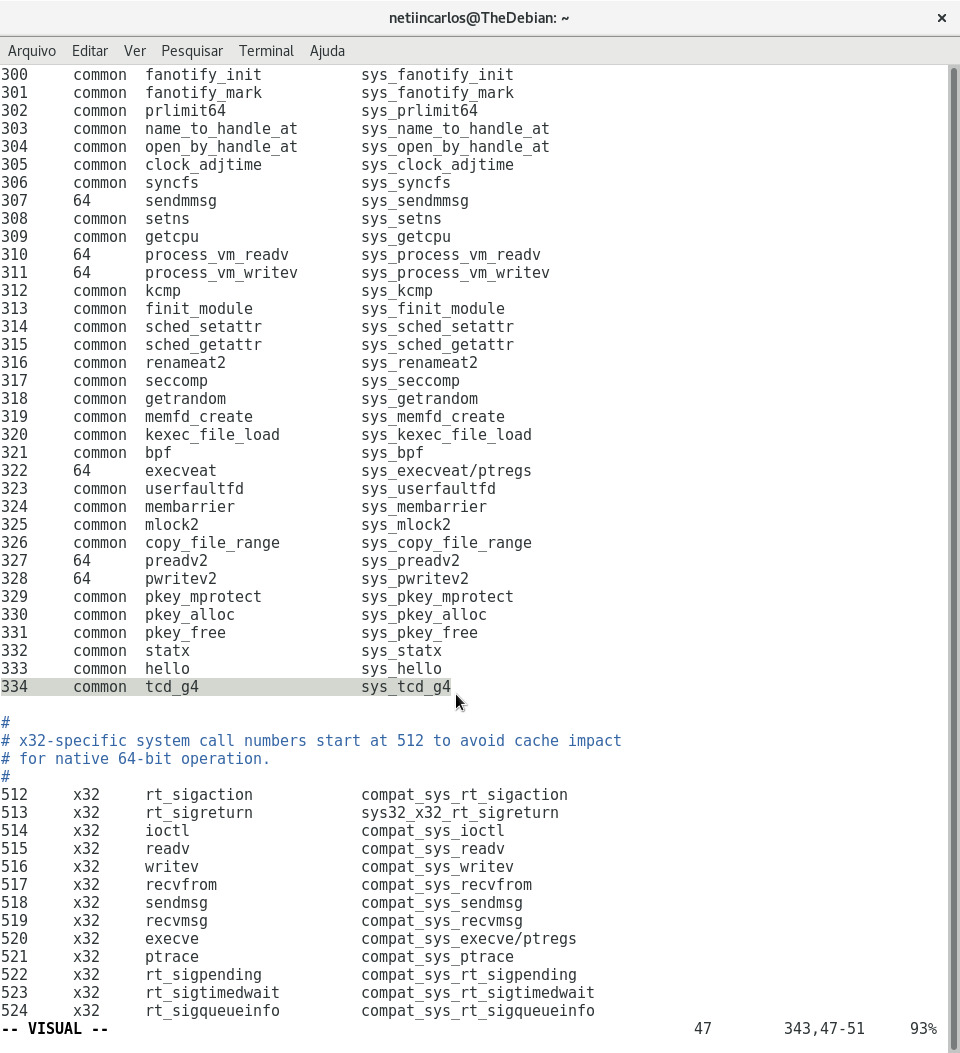


Figura 4: Arquivo “syscalls.h”

**5.2. Programas Auxiliares**

Neste trabalho, como forma de testar o funcionamento da system call, foram criado dois programas auxiliares, o primeiro e bem simples, chamado de “p1.c” tem o intuito de criar seis processos, nomeados de P1, P2, ..., P6. O processo P1 é o pai de todos outros processos, totalizando cinco filhos, e através da rotina de biblioteca “printf()” imprimimos na tela o pid de cada processo. Para gerar filhos usamos a system call “fork()”, que retorna um inteiro, zero para referenciar que está no filho e maior que zero para referenciar que está no processo pai.

O segundo programa auxiliar é chamado de “t.c” no qual lê um pid, cria um vetor de inteiro (buffer) de tamanho BUFF\_SIZE definido no código e chama a syscall sys\_tcd\_g4, e interpreta os valores passados para o buffer, sendo que sua primeira posição indica a quantidade de filhos e caso seja -1, indica que o processo não existe. A cada três posições temos o pid, ppid e uid dos processos filhos, por fim imprimimos esses dados.

**5.3. Códigos Fonte**

**5.3.1. “tcd\_g4.c”**

#include <linux/kernel.h>

#include <linux/syscalls.h>

#include <linux/sched.h>

#include <linux/pid.h>

#include <linux/cred.h>

#include <linux/slab.h>

#include <linux/highuid.h>

asmlinkage long sys\_tcd\_g4(int \_\_user pid, int \_\_user \*buf, int \_\_user size)

{

/\*CRIANDO PONTEIROS PARA ACESSAR A TASK\_STRUCT DE UM PROCESSSO;\*/

struct pid \*pid\_struct;

struct task\_struct \*task;

/\*CRIANDO PONTEIROS PARA PERCORRER E ACESSAR OS FILHOS DE UM DADO

PROCESSO;\*/

struct list\_head \*i;

struct task\_struct \*ts;

/\*CRIANDO VARIAVEIS DO TIPO INTEIRO PARA GUARDARMOS A QUANTIDADE

DE FILHOS DO PROCESSO QUE POSSUI O PID PASSADO E O PID, PPID,

UID DE SEUS FILHOS, RESPECTIVAMENTE;\*/

int aux, kpid, kppid, kuid;

/\*CRIANDO VARIAVEL PARA SABER QUANTOS FILHOS EXCEDERAM O TAMANHO

DO BUFFER E NAO FORAM ARMAZENADOS;\*/

int deep;

/\*FIM DA DECLARACAO DAS VARIAVEIS;\*/

/\*IMPRIMINDO O SUCESSO DA ENTRADA NA SYSCALL SYS\_TCD\_G4;\*/

printk("SYS\_TCD\_G4:\n");

/\*VERIFICANDO SE O PID, BUF E SIZE FOI ENVIADO CORRETAMENTE;\*/

if(!pid)

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (PID NULL)\n");

return -1;

}

if(!buf)

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (BUF NULL)\n");

return -1;

}

if(!size)

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (SIZE NULL)\n");

return -1;

}

/\*PROCURANDO STRUCT TASK\_STRUCT DO PROCESSO DADO O PID;

VERIFICA SE ACHOU A PID\_STRUCT, CASO NAO, RETORNA -1;

VERIFICA SE ACHOU A TASK\_STRUCT, CASO NAO, RETORNA -1;\*/

pid\_struct = find\_get\_pid(pid);

if(!pid\_struct)

{

printk("PID\_STRUCT NULA\n");

aux = -1;

if(copy\_to\_user(&buf[0], &aux, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,STATE)\n");

return -1;

}

return -1;

}

task = pid\_task(pid\_struct,PIDTYPE\_PID);

if(!task)

{

printk("TASK\_STRUCT NULA\n");

aux = -1;

if(copy\_to\_user(&buf[0], &aux, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,STATE)\n");

return -1;

}

return -1;

}

/\*IMPRIMINDO O VALOR DO STATE PRESENTE NA TASK\_STRUCT;

\*VERIFICANDO SE O PROCESSO EXISTE A PARTIR DO ARGUMENTO STATE EN-

CONTRADO NA TASK\_STRUCT;

\*VALORES DO STATE: -1 unrunnable, 0 runnable, >0 stopped;

\*CASO O PROCESSO ESTEJA UNRUNNABLE O BUF[0] RECEBE -1\*/

printk("O PROCESSO COM PID %d POSSUI STATE IGUAL A %ld\n", pid, task->state);

if(task->state == -1)

{

printk("O PROCESSO IDENTIDICADO PELO PID %d NAO FOI ENCONTRADO\n",pid);

aux = -1;

if(copy\_to\_user(&buf[0], &aux, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,STATE)\n");

return -1;

}

return -1;

}

/\*A VARIAVEL AUX VAI AJUDAR A SABER ONDE DEVE ESCREVER NO BUFFER;

\*ATRIBUI O VALOR 0 PARA OS CONTADORES, QUE SIGNIFICA A

QUANTIDADE DE FILHOS DO PROCESSO PRINCIPAL E QUANTIDADE DE

FILHOS DO PROCESSO PRINCIPAL QUE NAO FOI ADICIONADO NO BUFFER,

RESPECTIVAMENTE;

\*PERCORRENDO A TASK\_STRUCT DOS FILHOS E PEGANDO AS INFORMACOES NE-

CESSARIAS, LEMBRANDO QUE EH UMA LISTA CIRCULAR;\*/

aux = 0;

deep = 0;

list\_for\_each(i, &(task->children))

{

ts = list\_entry(i, struct task\_struct, sibling);

kpid = ts->pid;

kppid = ts->parent->pid;

kuid = ts->cred->uid.val;

/\*VERIFICA SE HA ESPACO NO BUFFER PARA INSERIR AS INFORMACOES

DO PROCESSO ATUAL;\*/

if( ((aux\*3)+3) < (size/sizeof(int)))

{

/\*ENVIANDO AS INFORMACOES SOBRE O PROCESSO FILHO PARA O

USER SPACE ATRAVES DO BUFFER, SENDO QUE ELE ARMAZENARA

NA POSICAO (QUANTIDADE DE FILHOS ATE AGORA, VEZES TRES(3)

+ 1(PID) OU 2(PPID) OU 3(UID))\*/

if(copy\_to\_user(&buf[((aux\*3)+1)], &kpid, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,PID)\n");

return -1;

}

if(copy\_to\_user(&buf[((aux\*3)+2)], &kppid, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,PPID)\n");

return -1;

}

if(copy\_to\_user(&buf[((aux\*3)+3)], &kuid, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,UID)\n");

return -1;

}

}else

{

/\*NOTIFICANDO O USUARIO QUE OS DADOS DO PROCESSO ATUAL NAO

FOI TRANSMITIDO PARA O BUFFER, DO USER SPACE, POIS A CAPA-

CIDADE FOI EXCEDIDA;

\*INCREMENTA O CONTADOR DE PROCESSOS NAO ENVIADO PARA O

BUFFER;\*/

printk("Os dados do Processo Filho(%d) nao foi colocado no buffer\n",aux);

deep++;

}

/\*INCREMENTA O CONTADOR DE PROCESSOS FILHOS;\*/

aux++;

/\*IMPRIMINDO AS INFORMACOES DO PROCESSO FILHO;\*/

printk("O Processo Filho(%d) possui o pid: %d\n",aux, ts->pid);

printk("O Processo Filho(%d) possui o ppid: %d\n",aux, ts->parent->pid);

printk("O Processo Filho(%d) possui o uid: %d\n\n",aux, ts->cred->uid.val);

}

/\*ENVIANDO OS DADOS DA QUANTIDADE DE FILHOS DO KERNEL

SPACE PARA O BUFFER DO USER SPACE;\*/

if(copy\_to\_user(&buf[0], &aux, sizeof(int)))

{

printk("\*\*\*ERROR SYS\_TCD\_G4\*\*\* (COPY\_TO\_USER,QT)\n");

return -1;

}

/\*IMPRIMINDO A QUANTIDADE DE FILHOS DO PROCESSO PRINCIPAL;\*/

printk("O Processo Principal possui %d filhos\n",aux);

/\*VERIFICA E NOTIFICA SE EXISTIR PROCESSOS FILHOS QUE NAO

FORAM ADICIONADOS NO BUFFER;\*/

if(deep > 0)

{

printk("Existe %d processos filhos que nao foram adicionados no buffer.\n",deep);

return 1;

}

/\*RETORNANDO 0 PARA INDICAR QUE A SYSCALL FOI BEM SUCEDIDA;\*/

return 0;

}

**5.3.2. “p1.c”**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

/\*PROCESSO QUE CRIA 5 FILHOS E ESPERA UM CHAR PARA FINALIZAR SUA EXECUCAO;\*/

void main()

{

pid\_t p1, p2, p3, p4, p5;

fflush(stdout);

printf("Processo P1: PID=%d\n", getpid());

fflush(stdout);

p1 = fork();

if(p1 == 0)

{

fflush(stdout);

printf("Processo P2: PID=%d\n", getpid());

fflush(stdout);

}else if(p1 > 0)

{

p2 = fork();

if(p2 == 0)

{

fflush(stdout);

printf("Processo P3: PID=%d\n", getpid());

fflush(stdout);

}else if(p2 > 0)

{

p3 = fork();

if(p3 == 0)

{

fflush(stdout);

printf("Processo P4: PID=%d\n", getpid());

fflush(stdout);

}else if(p3 > 0)

{

p4 = fork();

if(p4 == 0)

{

fflush(stdout);

printf("Processo P5: PID=%d\n", getpid());

fflush(stdout);

}else if(p4 > 0)

{

p5 = fork();

if(p5 == 0)

{

fflush(stdout);

printf("Processo P6: PID=%d\n", getpid());

fflush(stdout);

}

}

}

}

}

getchar();

}

**5.3.3. “t.c”**

#include <stdio.h>

#include <stdlib.h>

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <linux/unistd.h>

#include <linux/types.h>

/\*TAMANHO DO BUFFER ENVIADO PARA A SYSCALL PARA RECEBER INFORMACOES DOS

PROCESSOS FILHOS E QUANTIDADE DE FILHOS DE UM PROCESSO;\*/

#define BUFF\_SIZE 4096

void main()

{

int x, i;

long int status;

int \*buf;

/\*LENDO UM PID PARA ENVIAR COMO PARAMETRO DA SYSCALL SYS\_TCD\_G4;\*/

fflush(stdout);

printf("Digite o pid do processo a ser procurado:\n");

fflush(stdout);

fflush(stdin);

scanf("%d",&x);

/\*ALOCANDO MEMORIA PARA O VETOR DE INTEIROS(BUFFER) DE TAMANHO

BUFF\_SIZE;

\*O VETOR TERA BUFF\_SIZE DIVIDIDO PELO SIZEOF(INT);

\*VERIFICA SE A ALOCACAO DO BUFFER FOI BEM SUCEDIDA, CASO NAO

SEJA, FINALIZA O PROCESSO;

\*ATRIBUI 0 PARA O VETOR NA POSICAO 0, QUE INDICA A QUANTIDADE

DE FILHOS DO PROCESSO;\*/

buf = (int\*) malloc(BUFF\_SIZE);

if(!buf)

{

fflush(stdout);

printf("\*\*\*ERRO NA ALOCACAO\*\*\*\n");

fflush(stdout);

sleep(5);

return;

}

buf[0] = 0;

/\*PROTOTIPO DA SYSCALL SYS\_TCD\_G4:

asmlinkage long sys\_tcd\_g4(int \_\_user pid, int \_\_user \*buf, int \_\_user size);

\*CHAMANDO A SYSCALL PASSANDO COMO ARGUMENTO O NUMERO DELA,

IMPLEMENTADO NO ARQUIVO SYSCALL\_X64.TBL ENCONTRADO NA PASTA

~LINUX-4.13.12/ARCH/X86/ENTRY/SYSCALLS, O PID DO PROCESSO A

SER PROCURADO, O VETOR DE INTEIROS(BUFFER) E O TAMANHO DESSE

VETOR EM BYTES, RESPECTIVAMENTE;\*/

status = syscall(334, x, buf, BUFF\_SIZE);

/\*VERIFICA O RETORNO DA SYSCALL;\*/

if(status == 0)

{

fflush(stdout);

printf("System call deu certo(%ld)\n",status);

fflush(stdout);

}

else if(status == 1)

{

fflush(stdout);

printf("System call nao foi completada por falta de tamanho no buffer(%ld)\n",status);

fflush(stdout);

}

else if(status == -1)

{

if(buf[0] == -1)

{

fflush(stdout);

printf("System call informou que o processo nao existe(%ld)\n", status);

fflush(stdout);

}

else

{

fflush(stdout);

printf("System call nao executou como esperado(%ld)\n", status);

fflush(stdout);

}

sleep(3);

return;

}

else

{

fflush(stdout);

printf("System call retornou valor inesperado(%ld)\n",status);

fflush(stdout);

}

/\*IMPRIME PARA O USUARIO AS INFORMACOES COLETADAS\*/

fflush(stdout);

printf("A quantidade de filhos do processo com pid %d eh %d\n",x,buf[0]);

fflush(stdout);

i = 0;

while(i < buf[0])

{

if( ((i\*3)+3) > (BUFF\_SIZE/sizeof(int)) )

{

fflush(stdout);

printf("System call nao retornou as informacoes dos processos filhos seguintes\n");

fflush(stdout);

break;

}

fflush(stdout);

printf("\nProcesso Filho(%d):\n-Pid : %d\n-PPid : %d\n-Uid : %d\n",(i+1),buf[((i\*3)+1)],buf[((i\*3)+2)],buf[((i\*3)+3)]);

fflush(stdout);

i++;

}

/\*FINAL DO PROCESSO QUE UTILIZA A SYSCALL;\*/

fflush(stdout);

printf("Fim\n");

fflush(stdout);

}

**6. Conclusão**

O trabalho realizado teve grande valor para o enriquecimento intelectual dos alunos porque nos apresentou aspectos do desenvolvimento do kernel, como, suas estruturas internas, a programação em modo kernel, definição e aspectos de uma system call, dentre outros aspectos até então desconhecidos por grande parte dos alunos e também dos programadores de forma geral. Isso nos abre um leque de oportunidades incrível, não só como alunos para trabalhar em pesquisas relacionadas, mas como profissionais capazes de melhorar suas condições de trabalho, pois um grande profissional deve estar habituado no seu ambiente de trabalho, como o sistema operacional usado.

Além disso, nos trouxe uma familiaridade ainda maior com o Sistema Operacional Linux, nas distribuições Ubuntu e Debian, e entender de maneira geral como funcionam os kernels de todos os Sistemas Operacionais que operem de maneira semelhante.

A imersão no conteúdo e a habilidade de procurar conhecimento com as próprias mãos com certeza foram os diferenciais desse trabalho, já que esse era o desafio proposto, pois nem sempre teremos algum tutor ou profissional para auxiliarmos nas nossas jornadas para tornamos profissionais, assim aprendemos com os nossos erros e treinamos para encontrarmos soluções rápidas para solucionarmos nossos problemas de forma geral.

**7. Bibliografia**

WIKIPEDIA. Bloco de controle de processo. Disponível em:

<https://pt.wikipedia.org/wiki/Bloco\_de\_controle\_de\_processo>.

Acesso em: 12/12/2017.

WIKIPEDIA. Chamada de sistema. Disponível em:

<https://pt.wikipedia.org/wiki/Chamada\_de\_sistema>.

Acesso em: 12/12/2017.

JÚNIOR, Rivalino Matias. Unidade II – Modelo de Processos\Threads. Disponível em:

<http://www.facom.ufu.br/~rivalino/gbc045/%5b1%5d%20Slides%20(Lecture%20Notes)/UNIDADE%20II/GBC045-Unidade%20II.pdf>.

Acesso em: 12/12/2017.

PROGRESSIVA, Programação. O que são ChamadasDisponível em:

<http://www.programacaoprogressiva.net/2014/09/O-que-sao-Chamadas-de-Sistema-para-que-servem-Exemplos-de-System-Calss.html>.

Acesso em: 12/12/2017.

JÚNIOR, Rivalino Matias. Tutorial - Compilação de um novo Kernel Linux. Disponível em:

<http://www.facom.ufu.br/~rivalino/gbc045/%5b2%5d%20Papers%20&%20TRs/tutorial-kernel-generico.pdf>.

Acesso em: 12/12/2017.

JÚNIOR, Rivalino Matias. System Call. Disponível em:

<http://www.facom.ufu.br/~rivalino/gbc045/%5b2%5d%20Papers%20&%20TRs/System%20Call.pdf>.

Acesso em: 12/12/2017.

JÚNIOR, Rivalino Matias. Tutorial - Adicionando novas chamadas de sistema ao kernel Linux. Disponível em:

<http://www.facom.ufu.br/~rivalino/gbc045/%5b2%5d%20Papers%20&%20TRs/tutorial-syscall-linux.pdf>.

Acesso em: 12/12/2017.

STACKOVERFLOWS. Traversing task\_struct->children in Linux kernel. Disponível em:

<https://stackoverflow.com/questions/1446239/traversing-task-struct-children-in-linux-kernel>.

Acesso em: 12/12/2017.

FREEELECTRONS. Sched.h#L519. Disponível em:

<http://elixir.free-electrons.com/linux/v4.13.12/source/include/linux/sched.h#L519>.

Acesso em: 12/12/2017.

FREEELECTRONS. Cred.h#L110. Disponível em:

<http://elixir.free-electrons.com/linux/v4.13.12/source/include/linux/cred.h#L110>.

Acesso em: 12/12/2017.

STACKOVERFLOW. What’s the Deal with All the Different UIDs a Processs Can Have?. Disponível em:

<https://stackoverflow.com/questions/205070/whats-the-deal-with-all-the-different-uids-a-process-can-have>.

Acesso em: 12/12/2017.

WIKIPEDIA. Printk. Disponível em:

<https://en.wikipedia.org/wiki/Printk>.

Acesso em: 12/12/2017.

FREEELECTRONS. Pid.h#L118. Disponível em:

<https://elixir.free-electrons.com/linux/v3.8/source/include/linux/pid.h#L118>.

Acesso em: 12/12/2017.

FREEELECTRONS. Pid.h#L83. Disponível em:

<https://elixir.free-electrons.com/linux/v3.4/source/include/linux/pid.h#L83>.

Acesso em: 12/12/2017.

PSEUDOMUTO. Implementing a Generic Linked List in C. Disponível em:

<http://pseudomuto.com/development/2013/05/02/implementing-a-generic-linked-list-in-c/>.

Acesso em: 12/12/2017.

STACKOVERFLOW. List\_entry in Linux. Disponível em:

<https://stackoverflow.com/questions/5550404/list-entry-in-linux>.

Acesso em: 12/12/2017.

FREEELECTRONS. List.h#L254. Disponível em:

<http://elixir.free-electrons.com/linux/v4.13.12/source/drivers/gpu/drm/nouveau/include/nvif/list.h#L254>.

Acesso em: 12/12/2017.

FREEELECTRONS. Uaccess.h#L151. Disponível em:

<http://elixir.free-electrons.com/linux/v4.13.12/source/include/linux/uaccess.h#L151>.

Acesso em: 12/12/2017.

QUORA. Linux Kernel: How does copy\_to\_user work?. Disponível em:

<https://www.quora.com/Linux-Kernel-How-does-copy\_to\_user-work>.

Acesso em: 12/12/2017.

TANENBAUM, A. S. and Woodhull, A. S. Sistemas Operacionais - Projeto e Implementacao. Bookman, 2000.