



Eng. Informática

Sistemas Operativos

- Ano Letivo 2020/2021 -

Relatório do Trabalho Prático Nº 1

Ricardo Vieira a41726 [a41726@alunos.ipb.pt]

Henrique Araújo a41751 [a41751@alunos.ipb.pt]

1º Fase

1.a) Introdução

Esta primeira fase do trabalho foca-se principalmente em saber reconfigurar, recompilar e instalar um novo kernel do Linux e explorar as diferentes opções presentes no menu, para isso criamos com recurso ao VMware uma máquina virtual com Ubuntu MATE na versão 20.04.1 de 64 bits, 2 GB de RAM, 40 GB de espaço em disco e 2 cores.

Devido às restrições das nossas máquinas pessoais, tentamos explorar diferentes configurações de compilação de forma a minimizar o espaço em disco e os tempos de compilação/recompilação, bem como fazer um uso mais eficiente do CPU e RAM, para isto decidimos explorar alguns algoritmos de compressão de kernel, diferentes do pré configurado LZ4 em que o tamanho do kernel é 18% maior que o antigo e bastante usado gzip.

Também tentamos reduzir a quantidade de módulos e drivers restringindo nos apenas ao nosso tipo de processador e arquitetura, no caso um processador Intel x86 64 bits. O tamanho do kernel poderia ser minimizado ainda mais ao fazermos uma build minimalista apenas com os módulos e drivers absolutamente necessários, mas optamos para não o fazer para não perdermos o recurso a certas funcionalidades e drivers que mesmo não utilizadas no decorrer do trabalho nos poderiam ser úteis.

1.b) Compilação do kernel

Começamos por visitar o site oficial (kernel.org) de onde podemos obter o código fonte(source) da última versão estável do kernel, há quando a realização deste trabalho a última versão estável utilizada foi a 5.10.1.

Fazendo o download do source e extraindo o código para uma pasta criada para o propósito da realização do trabalho (KernelSO), decidimos antes de começar a construir o nosso novo kernel copiar a config atual na qual constituem os módulos e drivers necessários para o nosso sistema utilizando o seguinte comando:

```
a41726@ubuntu:~$ cd KernelSO/linux-5.10.1/
a41726@ubuntu:~/KernelSO/linux-5.10.1$ sudo cp -v /boot/config-$(uname -r) .config
'/boot/config-5.4.0-58-generic' -> '.config'
```

De seguida criamos uma config baseada na config atual e módulos carregados(lsmmod), desabilitando assim qualquer opção de modulo que não é necessária para os módulos carregados. Para isso utilizamos o seguinte comando:

```
a41726@ubuntu:~/KernelSO/linux-5.10.1$ make localmodconfig
```

De seguida foi necessário instalar as ferramentas e dependências necessárias para compilar o kernel utilizando o seguinte comando:

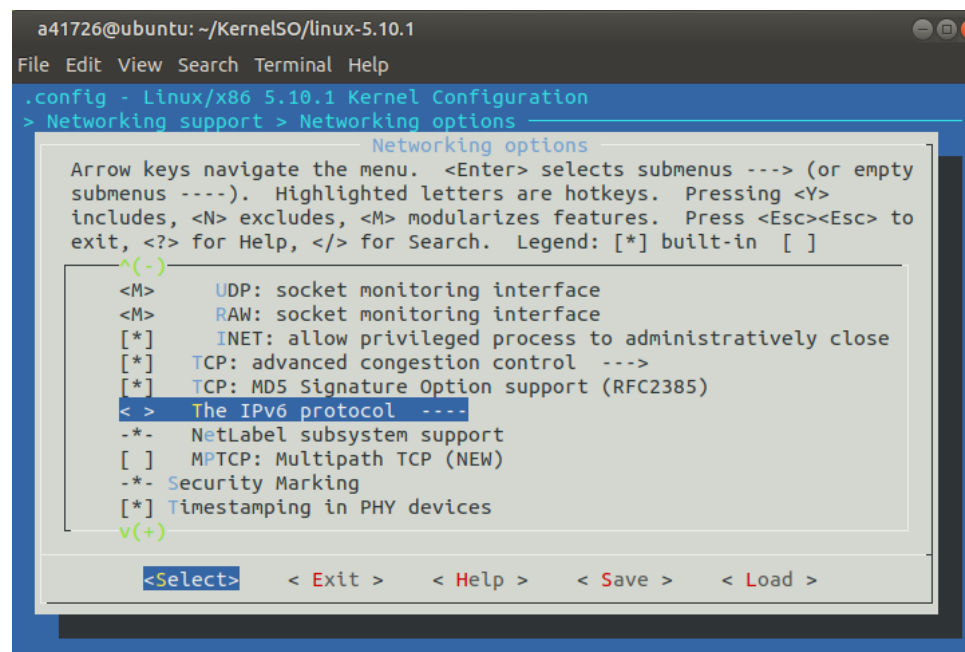
```
a41726@ubuntu:~$ sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

Tendo a atual config guardada e as ferramentas necessárias para a compilação do kernel iniciamos então a configuração do novo kernel utilizando o comando:

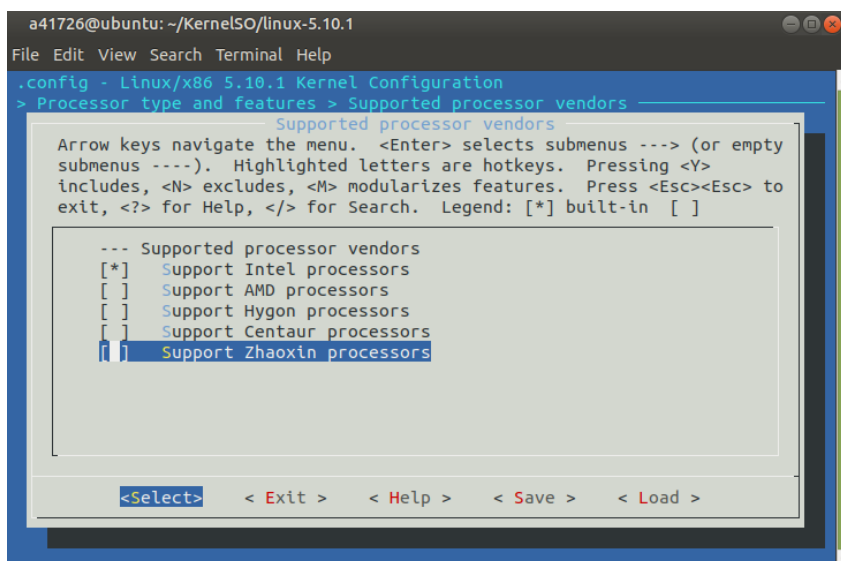
```
a41726@ubuntu:~$ cd KernelSO/linux-5.10.1/  
a41726@ubuntu:~/KernelSO/linux-5.10.1$ make menuconfig
```

Acendendo ao menu procedemos as seguintes alterações:

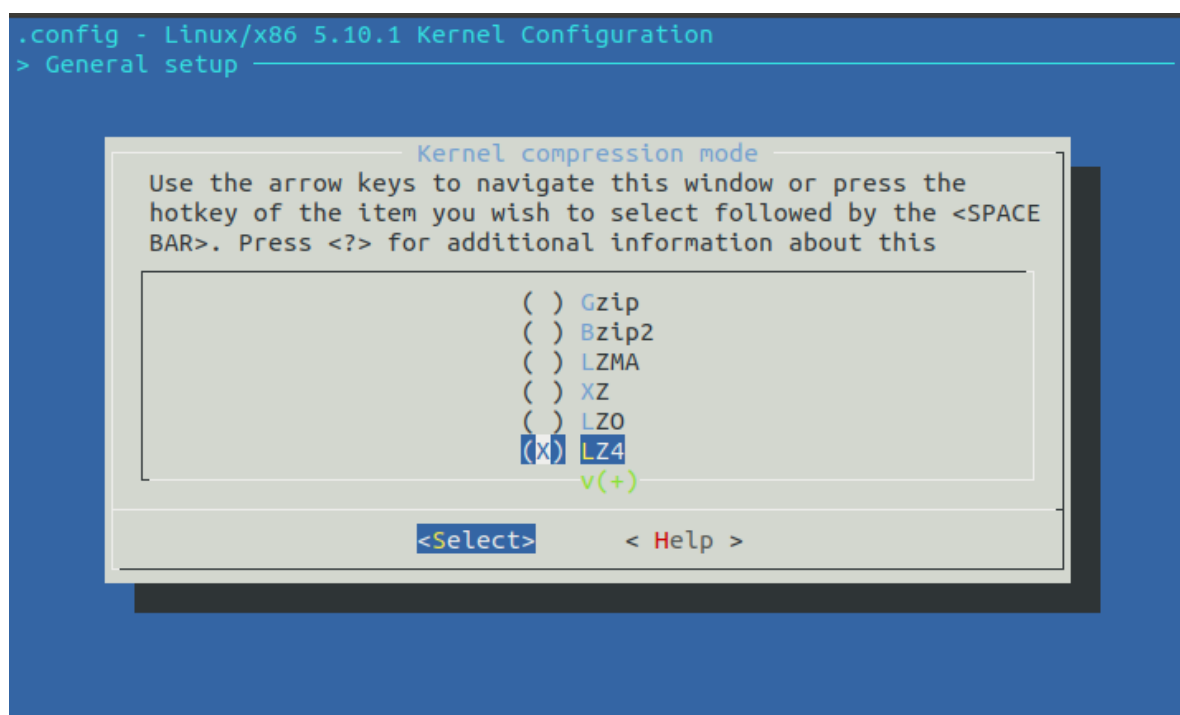
Desabilitamos o suporte para IPv6



Deixamos apenas habilitado as funções definidas para o nosso tipo de processador



Nesta primeira build deixamos o algoritmo de compressão do kernel pré-defenido, o LZ4.



Guardando a configuração e saindo do menu, demos início ao processo de compilação em paralelo para minimizar o tempo de compilação, e sendo o nosso processador de 2 cores decidimos usar 3 jobs. Para isto utilizamos o seguinte comando:

```
a41726@ubuntu:~$ make -j3
```

A compilação é bastante exigente e demorada como podemos ver abaixo, mesmo usando os 2 cores a 100% demorou entre 20 a 35 minutos para terminar a compilação

The image shows two terminal windows. The left window displays the output of the 'make' command, listing various object files being compiled, such as 'arch/x86/lib/iomem.o' and 'drivers/gpu/drm/amd/amdgpu/./andkfd/kfd_cratt.o'. The right window shows the output of the 'top' command, providing system statistics: 'Tasks: 110, 210 thr; 2 running', 'Load average: 3.28 3.31 2.91', and 'Uptime: 01:34:59'. Below the statistics is a table of running processes, with 'make -f' visible in the list.

Após terminada a compilação do kernel instalamos os módulos utilizando o seguinte comando:

```
a41726@ubuntu:~$ sudo make modules_install
```

Até aqui compilamos o kernel e instalamos os módulos, de seguida instalamos o kernel utilizando o seguinte comando:

```
a41726@ubuntu:~$ sudo make install
```

Instalado o kernel demos update ao multi-carregador do nosso sistema operativo, o grub2, para isso utilizamos o seguinte comando:

```
a41726@ubuntu:~$ sudo update-grub2
```

Para efeitos de melhor identificação do nosso kernel mudamos a identificação do extraversion no nosso makefile.

```
1 # SPDX-License-Identifier: GPL-2.0
2 VERSION = 5
3 PATCHLEVEL = 10
4 SUBLEVEL = 1
5 EXTRAVERSION = -a41726
6 NAME = Kleptomaniac Octopus
7
8 # *DOCUMENTATION*
```

Reiniciando a máquina realizamos os testes da nova configuração:

```
a41726@ubuntu: ~
File Edit View Search Terminal Help
a41726@ubuntu:~$ uname -a
Linux ubuntu 5.10.1-a41726 #2 SMP Mon Dec 21 07:16:04 PST 2020 x86_64 x86_64
x86_64 GNU/Linux
a41726@ubuntu:~$ cat /proc/net/if_inet6
cat: /proc/net/if_inet6: No such file or directory
a41726@ubuntu:~$ du -sh KernelSO/linux-5.10.1/
4.5G    KernelSO/linux-5.10.1/
a41726@ubuntu:~$ ls -lha /boot | grep 5.10.1-a41726
-rw-r--r-- 1 root root 152K Dec 21 07:17 config-5.10.1-a41726
-rw-r--r-- 1 root root 47M Dec 21 07:17 initrd.img-5.10.1-a41726
-rw-r--r-- 1 root root 5.2M Dec 21 07:17 System.map-5.10.1-a41726
lrwxrwxrwx 1 root root 21 Dec 21 07:17 vmlinuz -> vmlinuz-5.10.1-a41726
-rw-r--r-- 1 root root 11M Dec 21 07:17 vmlinuz-5.10.1-a41726
a41726@ubuntu:~$
```

Como podemos verificar, o modulo de IPv6 não foi compilado na nossa nova versão do kernel. Além disso o tamanho do nosso kernel é de 4.5 GB e temos uma imagem de kernel de 11MB, para além disso são carregados 47MB para a memória RAM a quando a inicialização do Linux.

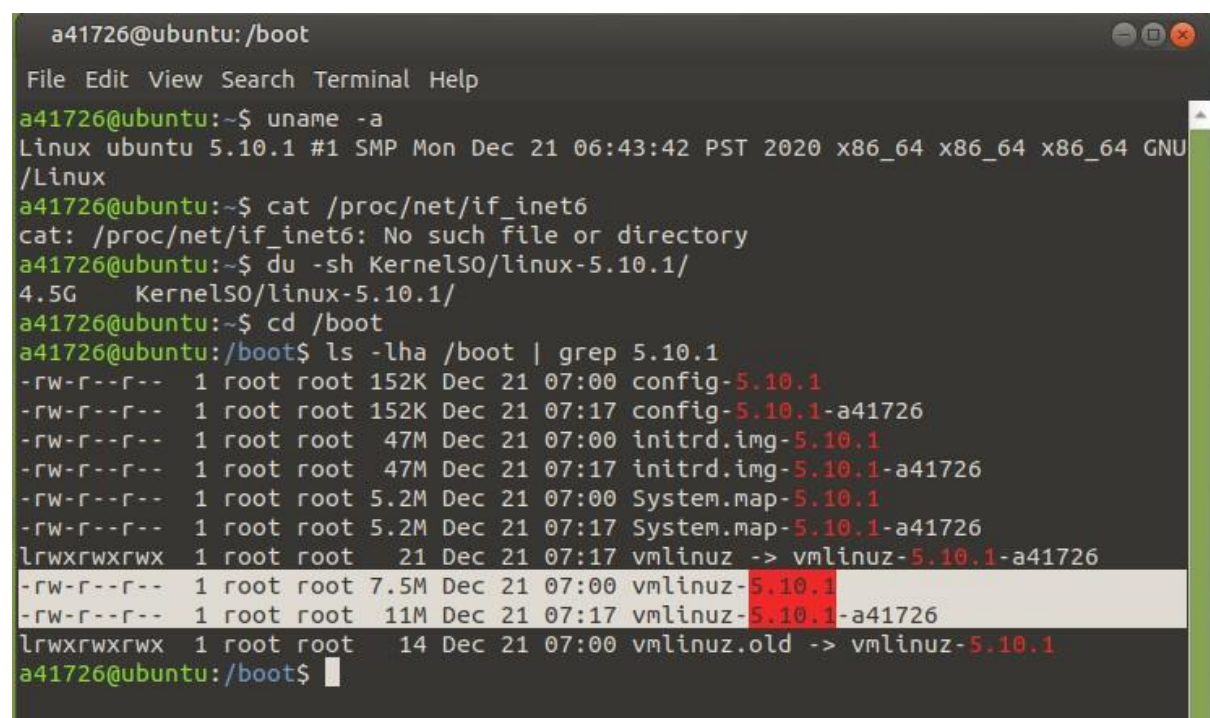
1.c) Kernel alternativo

Mediante as informações e tamanhos citados acima, decidimos recompilar o kernel e testar um algoritmo de compressão diferente para buscar uma melhor otimização do espaço e da RAM, pra isso usamos o recente e recém introduzido ao kernel, zstd, que por sua vez tem um rácio de compressão intermediário no entanto conta com tempos de descompressão mais rápidos que o lz0 mas menores que o lz4 além de contar com melhor compressão que o gzip. Esperamos assim obter um kernel igualmente rápido e com uma imagem mais pequena.

O processo de re-compilação é semelhante ao produzido acima com a exceção de precisarmos do utilitário zstd e termos que mudar o algoritmo para zstd. Como estamos a recompilar o processo de compilação demorou cerca de 2 minutos. Para instalar o utilitário zstd utilizamos o seguinte comando:

```
a41726@ubuntu:~$ sudo apt install zstd
```

Para facilitar a comparação deixamos o extraversion da nossa versão vazia.

A terminal window titled 'a41726@ubuntu: /boot' with a menu bar (File, Edit, View, Search, Terminal, Help). The user runs several commands: 'uname -a' showing 'Linux ubuntu 5.10.1 #1 SMP Mon Dec 21 06:43:42 PST 2020 x86_64 x86_64 x86_64 GNU/Linux'; 'cat /proc/net/if_inet6' resulting in 'No such file or directory'; 'du -sh KernelSO/linux-5.10.1/' showing '4.5G KernelSO/linux-5.10.1/'; 'cd /boot'; and 'ls -lha /boot | grep 5.10.1'. The last command lists files in /boot related to kernel 5.10.1. The files and their sizes are: config-5.10.1 (152K), config-5.10.1-a41726 (152K), initrd.img-5.10.1 (47M), initrd.img-5.10.1-a41726 (47M), System.map-5.10.1 (5.2M), System.map-5.10.1-a41726 (5.2M), vmlinuz -> vmlinuz-5.10.1-a41726 (21), vmlinuz-5.10.1 (7.5M), vmlinuz-5.10.1-a41726 (11M), and vmlinuz.old -> vmlinuz-5.10.1 (14M). The last three lines are highlighted in grey.

```
a41726@ubuntu: /boot
File Edit View Search Terminal Help
a41726@ubuntu:~$ uname -a
Linux ubuntu 5.10.1 #1 SMP Mon Dec 21 06:43:42 PST 2020 x86_64 x86_64 x86_64 GNU/Linux
a41726@ubuntu:~$ cat /proc/net/if_inet6
cat: /proc/net/if_inet6: No such file or directory
a41726@ubuntu:~$ du -sh KernelSO/linux-5.10.1/
4.5G    KernelSO/linux-5.10.1/
a41726@ubuntu:~$ cd /boot
a41726@ubuntu:/boot$ ls -lha /boot | grep 5.10.1
-rw-r--r-- 1 root root 152K Dec 21 07:00 config-5.10.1
-rw-r--r-- 1 root root 152K Dec 21 07:17 config-5.10.1-a41726
-rw-r--r-- 1 root root 47M Dec 21 07:00 initrd.img-5.10.1
-rw-r--r-- 1 root root 47M Dec 21 07:17 initrd.img-5.10.1-a41726
-rw-r--r-- 1 root root 5.2M Dec 21 07:00 System.map-5.10.1
-rw-r--r-- 1 root root 5.2M Dec 21 07:17 System.map-5.10.1-a41726
lrwxrwxrwx 1 root root 21 Dec 21 07:17 vmlinuz -> vmlinuz-5.10.1-a41726
-rw-r--r-- 1 root root 7.5M Dec 21 07:00 vmlinuz-5.10.1
-rw-r--r-- 1 root root 11M Dec 21 07:17 vmlinuz-5.10.1-a41726
lrwxrwxrwx 1 root root 14 Dec 21 07:00 vmlinuz.old -> vmlinuz-5.10.1
a41726@ubuntu:/boot$
```

Comparando os valores registados acima, utilizando o algoritmo zstd conseguimos uma imagem do kernel 68% mais pequena.

2º Fase

2.a) Introdução

Nesta segunda fase serão feitas as alterações necessárias ao kernel produzido na primeira fase a fim de suportar duas novas primitivas.

- `long getnchildren (int debug)`: retorna o número de processos filhos que o processo invocador tem, quando a primitiva é invocada;
- `long getcpid (int order, int debug)`: devolve o PID do filho do processo invocador, que ocupa a posição `order` na lista dos seus filhos, por exemplo, se o invocador tiver 4 filhos, de PIDs 87, 94, 111 e 150, e essa for também a ordem pela qual estão organizados na lista de filhos, então `getcpid(0,...)` deve retornar 87, `getcpid(1,...)` deve retornar 94, etc.; se `order` for demasiado grande, tendo em conta o número atual de filhos, a primitiva deve retornar -1.

O parâmetro `debug` irá controlar a produção de mensagens que são acrescentadas ao ficheiro `/var/log/kern.log` permitindo acompanhar a execução das primitivas.

2.b) Primitiva `getnchildren`

Usamos como fonte para desenvolver o código desta primitiva a obra “Linux Kernel Development”, Robert Love, Addison-Wesley, 2010, em especial do capítulo 3 (Process Management, The Process Family Tree), onde nos é dito que existe uma hierarquia direta entre todos os processos em Linux, onde todos os processos possuem um só pai, e onde todos os filhos desse processo são irmãos (siblings).

A relação entre processos está guardada dentro do descritor do processo. Onde cada `task_struct` tem um apontador para o `task_struct` do pai, chamado de *parent*, e a lista dos filhos chamado de *children*. Sendo assim, dado o processo atual é nos possível obter o seu descritor utilizando o seguinte código:

```
Struct task_struct *my_parent = current->parent;
```


De forma similar é possível iterar sobre os filhos utilizando o seguinte código:

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
    task = list_entry(list, struct task_struct, sibling);
    // a task agora aponta para o filho atual
}
```

Ou seja, poderíamos iterar sobre os filhos usando a função “list_for_each” que atua como uma lista ligada (LinkedList), onde a mesma recebe dois parâmetros:

- 1) Posição (neste caso “list”) que atua como cursor no loop, ao percorrer os filhos do processo pai.
- 2) Cabeça da lista ligada (neste caso “¤t->children”) onde “current” é o processo que invocou a chamada do sistema, e onde ¤t->children compõe um filho desse processo. É uma variável global do tipo “struct task_struct”.

Sendo assim, falta só armazenar toda a informação do filho iterado na variável “task”, dando uso à função “list_entry” que recebe como parâmetros 3 variáveis:

- 1) Um “struct list_head” da lista iterada que, no nosso caso, é “list”.
- 2) O tipo de estrutura embebida (como estamos a armazenar uma task_struct, este parâmetro especifica isso)
- 3) O membro da lista dada no primeiro parâmetro, que no nosso caso é “sibling”, que é um campo da estrutura “current” que nos dá um dos irmãos (processos com o mesmo pai) iterados.

Sendo assim o nosso código usou estas estruturas, onde `list_for_each` itera por cada filho do processo pai e incrementa o contador, guardando assim a quantidade de filhos numa variável.

Segue então o código:

```
1 #include <linux/kernel.h>
2 #include <linux/syscalls.h>
3
4 SYSCALL_DEFINE1(getnchildren, int, debug) {
5
6     struct task_struct *task;
7     struct list_head *list;
8
9     int childCount = 0;
10
11     list_for_each(list, &current->children) {
12         task = list_entry(list, struct task_struct, sibling);
13         childCount++;
14
15         if(debug==1) {
16             printk("Filhos: %d \n", childCount);
17         }
18     }
19
20     return childCount;
21 }
22
```

2.c) Primitiva getpid

Nesta primitiva fizemos uso da estrutura do getnchildren, mas desta vez guardamos o PID dos processos filho num array, devolvendo assim o PID do filho do processo invocador, que ocupa a posição order na lista dos seus filhos.

Segue então o código:

```
1 #include <linux/kernel.h>
2 #include <linux/syscalls.h>
3
4 SYSCALL_DEFINE2(getcpid, int, order, int, debug) {
5
6     struct task_struct *task;
7     struct list_head *list;
8
9     int arr[99];
10    int index = 0;
11
12    list_for_each(list, &current->children){
13
14        task = list_entry(list, struct task_struct, sibling);
15        arr[index] = task->pid;
16        index++;
17    }
18
19    if (debug == 1)
20        printk("Filho de order %d com pid %d \n", order, arr[order]);
21
22    if (order>index)
23        return -1;
24
25    else
26        return arr[order];
27 }
```

2.d) Implementação das systemcalls no kernel

Para implementar as primitivas no kernel, iremos trabalhar no diretório source do nosso kernel

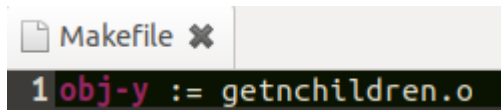
```
a41726@ubuntu:~$ cd /linux-5.10.1/
```

Começamos por criar um diretório para as nossas primitivas, para isso utilizamos o seguinte comando:

```
a41726@ubuntu:~$ mkdir getnchildren
a41726@ubuntu:~$ mkdir getpid
```

Nos nossos diretórios das primitivas iremos adicionar o nosso código e o Makefile, usando os seguintes comandos. No caso utilizamos o vscode como editor de texto.

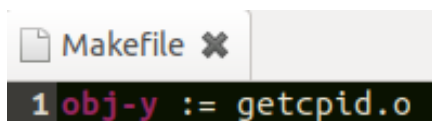
```
a41726@ubuntu:~$ cd getnchildren
a41726@ubuntu:~$ code getnchildren.c
a41726@ubuntu:~$ code Makefile
```



```
Makefile ✕
1 obj-y := getnchildren.o
```

E de forma similar para o getcpid:

```
a41726@ubuntu:~$ cd getcpid
a41726@ubuntu:~$ code getcpid.c
a41726@ubuntu:~$ code Makefile
```

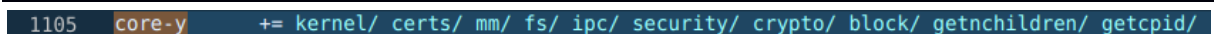


```
Makefile ✕
1 obj-y := getcpid.o
```

No nosso Makefile apenas escrevemos uma linha de código, onde y representa built in, isto é, representa um sim na config do kernel.

De seguida adicionamos os diretórios das nossas primitivas ao makefile do nosso kernel para que estas sejam compiladas há quando a recompilação do kernel. Utilizamos o seguinte comando:

```
a41726@ubuntu:~$ code Makefile
```



```
1105 core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ getnchildren/ getcpid/
```

De seguida implementamos os protótipos das nossas primitivas no header das systemcalls. Para isso utilizamos o seguinte comando:

```
a41726@ubuntu:~$ code include/linux/syscalls.h
```

E adicionamos os protótipos das nossas primitivas.

```
1351  asmlinkage long sys_getnchildren(int);
1352  asmlinkage long sys_getcpid(int, int);
1353
1354  #endif
```

De seguida adicionamos as nossas primitivas á tabela de system calls do kernel, respeitando a nossa arquitetura (x86_64 bits), utilizando o seguinte comando:

```
a41726@ubuntu:~$ code arch/x86/entry/syscalls/syscall_64.tbl
```

```
365  441 common  getnchildren  sys_getnchildren
366  442 common  getcpid      sys_getcpid
367  #
```

De seguida criamos o header das nossas primitivas, no qual conterà o código necessário para a implementação da componente userland das duas system calls, onde basicamente esta irá invocar a componente no kernel, fazendo de referência os códigos de system calls que acabamos de adicionar á tabela de system calls do nosso kernel.

Há quando a utilização das nossas primitivas irá ser necessário apenas fazer o include do nosso header, sendo este, `#include <mysyscalls.h>`.

Segue então abaixo o código do nosso header.

```
*mysyscalls.h ✕
1 #include <sys/syscall.h>
2 #include <sys/types.h>
3
4 long getnchildren(int debug) {
5     return syscall(441, debug);
6 }
7
8 long getcpid(int order, int debug) {
9     return syscall(442, order, debug);
10 }
```

2.e) Implementação dos headers

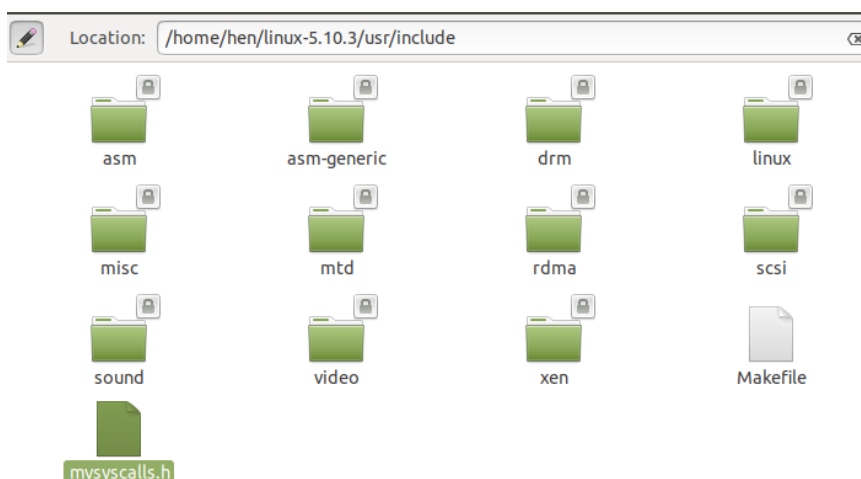
Para que o GCC consiga encontrar o nosso ficheiro header, primeiro temos que saber como é feita a procura de ficheiros headers definidos entre < e > (como em `#include <mysyscalls.h>`).

O GCC é composto por muitos algoritmos, mas o algoritmo que nos interessa para os ficheiros headers é o CPP (C Pre-Processor). Ao executar o comando “`cpp -v`”, este retorna os diretórios onde começa a procura pelos headers definidos entre < ... >.

```
#include ".." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/9/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
```

Através do output, verificamos que um dos diretórios de pesquisa é o `/usr/include`. Então, adicionamos o nosso `mysyscalls.h` ao diretório correspondente no kernel.

Na próxima etapa verificamos como é que o compilador do kernel adiciona, depois, o header ao diretório verificado no CPP. Adicionamos assim o nosso header `mysyscalls.h` ao diretório.



2.d) Recompilação do Kernel & Teste das Primitivas

Por fim recompilamos o kernel com 3 jobs de forma similar á segunda fase e reiniciamos a máquina com o novo kernel, utilizando os seguintes comandos:

```
a41726@ubuntu:~$ sudo make -j3
a41726@ubuntu:~$ sudo make modules_install -j3
a41726@ubuntu:~$ sudo make headers_install -j3
a41726@ubuntu:~$ sudo make install -j3
a41726@ubuntu:~$ sudo update-grub2
a41726@ubuntu:~$ sudo reboot
```

Para terminar testamos as nossas novas primitivas utilizando o seguinte código que nos foi fornecido.

```
1 // mysyscalls-test.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <mysyscalls.h>
7
8 #define NCHILDREN 10
9 int main()
10 {
11     int pids[NCHILDREN], i;
12     int original_parent = getpid();
13
14     printf("before fork: current number of children: %ld\n", getnchildren(1));
15     for (i = 0; i < NCHILDREN; i++)
16     {
17         pids[i] = fork();
18         if (pids[i] == 0)
19         {
20             while (getppid() == original_parent);
21             exit(0);
22         }
23     }
24     printf("after fork: current number of children: %ld\n", getnchildren(1));
25     for (i = 0; i < NCHILDREN; i++)
26         printf("pids[%d]:%d\tgetcpid(%d,1):%ld\n", i, pids[i], i, getcpid(i, 1))
27     return 0;
28 }
```

E realizamos os testes das primitivas:

```
fish /home/a41726
File Edit View Search Terminal Help
a41726@ubuntu ~> gcc mysyscall-test.c -o mysyscall-test
a41726@ubuntu ~> ./mysyscall-test
before fork: current number of children: 0
after fork: current number of children: 10
pids[0]:8721      getcpid(0,1):8721
pids[1]:8722      getcpid(1,1):8722
pids[2]:8723      getcpid(2,1):8723
pids[3]:8724      getcpid(3,1):8724
pids[4]:8725      getcpid(4,1):8725
pids[5]:8726      getcpid(5,1):8726
pids[6]:8727      getcpid(6,1):8727
pids[7]:8728      getcpid(7,1):8728
pids[8]:8729      getcpid(8,1):8729
pids[9]:8730      getcpid(9,1):8730
a41726@ubuntu ~>

sudo /
File Edit View Search Terminal Help
Dec 28 10:18:14 ubuntu kernel: [ 1130.527092] Filhos: 1
Dec 28 10:18:14 ubuntu kernel: [ 1130.527095] Filhos: 2
Dec 28 10:18:14 ubuntu kernel: [ 1130.527096] Filhos: 3
Dec 28 10:18:14 ubuntu kernel: [ 1130.527097] Filhos: 4
Dec 28 10:18:14 ubuntu kernel: [ 1130.527098] Filhos: 5
Dec 28 10:18:14 ubuntu kernel: [ 1130.527098] Filhos: 6
Dec 28 10:18:14 ubuntu kernel: [ 1130.527099] Filhos: 7
Dec 28 10:18:14 ubuntu kernel: [ 1130.527100] Filhos: 8
Dec 28 10:18:14 ubuntu kernel: [ 1130.527101] Filhos: 9
Dec 28 10:18:14 ubuntu kernel: [ 1130.527102] Filhos: 10
Dec 28 10:18:14 ubuntu kernel: [ 1130.527152] Filho de order 0 com pid 8721
Dec 28 10:18:14 ubuntu kernel: [ 1130.527171] Filho de order 1 com pid 8722
Dec 28 10:18:14 ubuntu kernel: [ 1130.527187] Filho de order 2 com pid 8723
Dec 28 10:18:14 ubuntu kernel: [ 1130.527202] Filho de order 3 com pid 8724
Dec 28 10:18:14 ubuntu kernel: [ 1130.527217] Filho de order 4 com pid 8725
Dec 28 10:18:14 ubuntu kernel: [ 1130.527232] Filho de order 5 com pid 8726
Dec 28 10:18:14 ubuntu kernel: [ 1130.527248] Filho de order 6 com pid 8727
Dec 28 10:18:14 ubuntu kernel: [ 1130.527263] Filho de order 7 com pid 8728
Dec 28 10:18:14 ubuntu kernel: [ 1130.527278] Filho de order 8 com pid 8729
Dec 28 10:18:14 ubuntu kernel: [ 1130.527293] Filho de order 9 com pid 8730
```


Referencias

List_for_each e list_entry: <https://elixir.bootlin.com/linux/v5.10.3/source/include/linux/list.h>

Váriável “current” no kernel: <https://stackoverflow.com/questions/12434651/what-is-the-current-in-linux-kernel-source>

Siblings: <https://stackoverflow.com/questions/34704761/why-sibling-list-is-used-to-get-the-task-struct-while-fetching-the-children-of-a>

In-kernel memory compression: <https://lwn.net/Articles/545244/>

“Linux Kernel Development”, Robert Love, Addison-Wesley, 2010