



UNIVERSIDADE FEDERAL DO CEARÁ – CAMPUS SOBRAL

CURSO DE ENGENHARIA DA COMPUTAÇÃO

DISCIPLINA: Sistemas Operacionais

PROFESSOR: Joniel Bastos

LISTA Nº 01

Manipulando Processos no Linux e Processos em C

ALUNO	MATRÍCULA
--------------	------------------

Jonas Carvalho Fortes	494513
------------------------------	---------------

Sobral – CE

2022

SUMÁRIO

1. OBJETIVOS.....	3
2. MATERIAL UTILIZADO	3
3. PROCEDIMENTO EXPERIMENTAL	4
4. REFERÊNCIAS BIBLIOGRÁFICAS	18

1. OBJETIVOS DA PRÁTICA

- Manipular os processos no Linux.
- Gerenciar processos utilizando a linguagem de programação C.

2. MATERIAL UTILIZADO

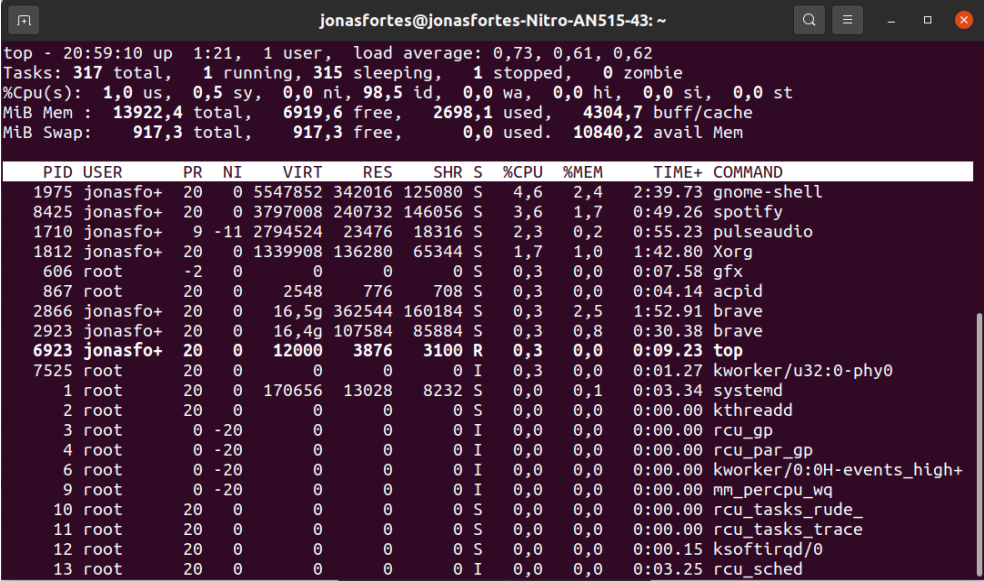
- Dual-boot do Sistema Operacional Linux. Distribuição Ubuntu 20.04.1.
- Visual Studio Code v1.67.0.
- Gcc v9.4.0.

3. PROCEDIMENTO EXPERIMENTAL

Este relatório divide-se em duas partes. A primeira trata dos procedimentos referentes à manipulação de processos utilizando o sistema operacional Linux e a segunda, dos processos na linguagem de programação C.

Na primeira parte, iniciou-se verificando o funcionamento de um comando cujo função é exibir uma lista dos processos do sistema e suas respectivas estatísticas. Trata-se do comando “top” e a partir dele é possível obter informações como o ID do processo, prioridade do processo e quantidade de memória virtual utilizada, entre outras informações. Ao executar o comando no terminal, tem-se a seguinte visualização mostrada na figura 01.

Figura 01 – Execução do comando “top”



```
jonasfortes@jonasfortes-Nitro-AN515-43: ~
top - 20:59:10 up 1:21, 1 user, load average: 0,73, 0,61, 0,62
Tasks: 317 total, 1 running, 315 sleeping, 1 stopped, 0 zombie
%Cpu(s): 1,0 us, 0,5 sy, 0,0 ni, 98,5 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
MiB Mem : 13922,4 total, 6919,6 free, 2698,1 used, 4304,7 buff/cache
MiB Swap: 917,3 total, 917,3 free, 0,0 used, 10840,2 avail Mem
```

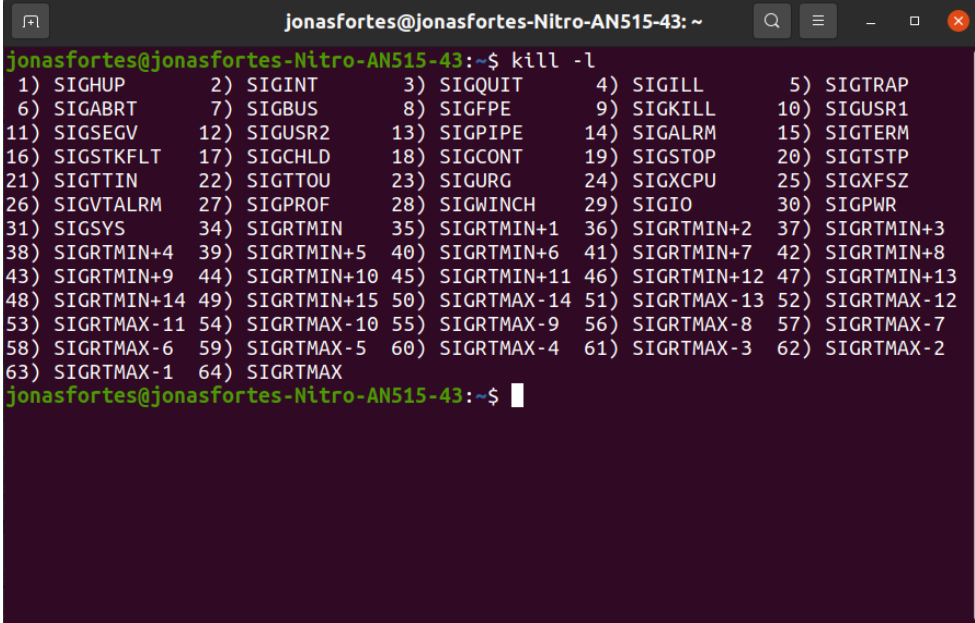
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1975	jonasfo+	20	0	5547852	342016	125080	S	4,6	2,4	2:39.73	gnome-shell
8425	jonasfo+	20	0	3797008	240732	146056	S	3,6	1,7	0:49.26	spotify
1710	jonasfo+	9	-11	2794524	23476	18316	S	2,3	0,2	0:55.23	pulseaudio
1812	jonasfo+	20	0	1339908	136280	65344	S	1,7	1,0	1:42.80	Xorg
606	root	-2	0	0	0	0	S	0,3	0,0	0:07.58	gfx
867	root	20	0	2548	776	708	S	0,3	0,0	0:04.14	acpid
2866	jonasfo+	20	0	16,5g	362544	160184	S	0,3	2,5	1:52.91	brave
2923	jonasfo+	20	0	16,4g	107584	85884	S	0,3	0,8	0:30.38	brave
6923	jonasfo+	20	0	12000	3876	3100	R	0,3	0,0	0:09.23	top
7525	root	20	0	0	0	0	I	0,3	0,0	0:01.27	kworker/u32:0-phy0
1	root	20	0	170656	13028	8232	S	0,0	0,1	0:03.34	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:0H-events_high+
9	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_wq
10	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_rude_
11	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_trace
12	root	20	0	0	0	0	S	0,0	0,0	0:00.15	ksoftirqd/0
13	root	20	0	0	0	0	I	0,0	0,0	0:03.25	rcu_sched

Fonte: Próprio autor

Ao observar a figura 01, nota-se que as informações referentes aos processos estão organizadas em colunas. Na última coluna encontram-se os nomes dos comandos referentes aos processos; e para saber a prioridade do processo basta observar a coluna três (PR). Por exemplo, o comando “spotify” tem prioridade de processamento igual a 20.

Em seguida, o comando “kill” foi analisado. Tal comando serve para enviar sinais para os processos, de modo que o processo seja pausado, interrompido ou encerrado a depender do sinal enviado ao processo. Para visualizar a lista de sinais que podem ser enviados, é usado o comando “kill -l”. A figura 02 mostra a execução deste comando.

Figura 02 – Execução do comando “kill -l”



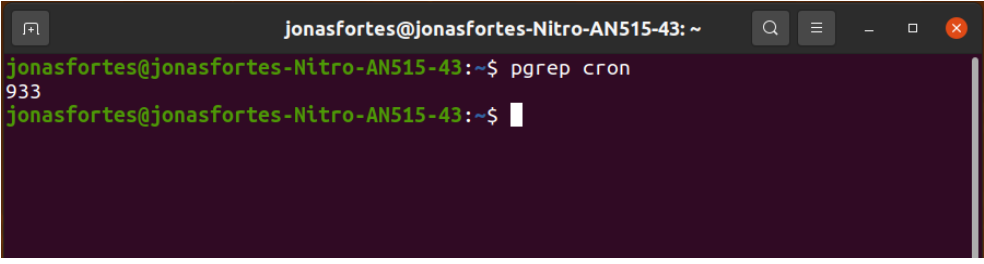
```
jonasfortes@jonasfortes-Nitro-AN515-43: ~  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ kill -l  
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP  
6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1  
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM     15) SIGTERM  
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT     19) SIGSTOP     20) SIGTSTP  
21) SIGTTIN    22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ  
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH    29) SIGIO        30) SIGPWR  
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13  
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12  
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  
63) SIGRTMAX-1 64) SIGRTMAX  
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Na figura 02 nota-se vários sinais, numerados de 1 a 64. Ao executar o comando “kill”, pode-se escolher um desses sinais para ser enviado ao processo. Vale ressaltar que caso não seja especificado o sinal no comando, o sinal padrão enviado será o SIGTERM.

Caso deseje-se finalizar o processo correspondente ao agendador de tarefas (cron), usa-se o comando “kill” juntamente ao PID do processo, ou seja, seu identificador. Para tal, utiliza-se o comando “pgrep cron” para consultar o PID deste processo em específico, como mostra a figura 03.

Figura 03 – Execução do comando “pgrep cron”

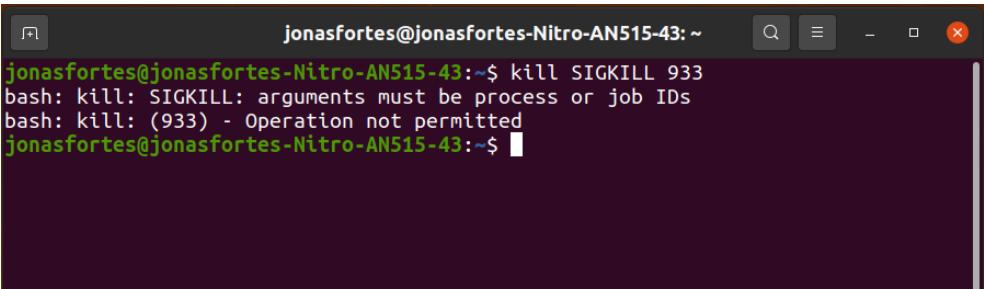
A terminal window with a dark purple background. The title bar shows 'jonasfortes@jonasfortes-Nitro-AN515-43: ~'. The prompt is 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$'. The command 'pgrep cron' has been entered, and the output '933' is displayed on the next line. The prompt is now 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$' with a cursor.

```
jonasfortes@jonasfortes-Nitro-AN515-43:~$ pgrep cron
933
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Após obter o PID do processo “cron”, basta enviar, por exemplo, um sinal de eliminação (SIGKILL) junto ao PID do processo, utilizando o seguinte comando: “kill SIGKILL 933”. Dessa forma, o processo “cron” será finalizado. A figura 04 mostra a execução deste comando.

Figura 04 – Execução do comando “kill SIGKILL 933”

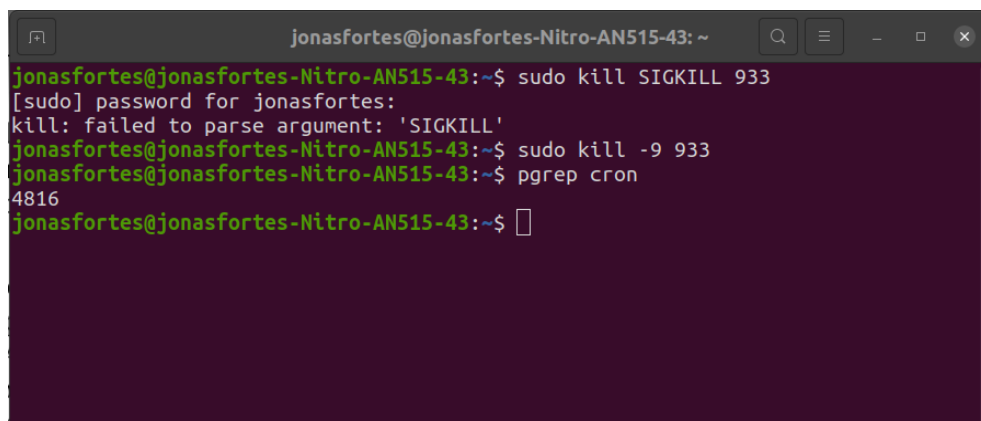
A terminal window with a dark purple background. The title bar shows 'jonasfortes@jonasfortes-Nitro-AN515-43: ~'. The prompt is 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$'. The command 'kill SIGKILL 933' has been entered. The output shows two error messages: 'bash: kill: SIGKILL: arguments must be process or job IDs' and 'bash: kill: (933) - Operation not permitted'. The prompt is now 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$' with a cursor.

```
jonasfortes@jonasfortes-Nitro-AN515-43:~$ kill SIGKILL 933
bash: kill: SIGKILL: arguments must be process or job IDs
bash: kill: (933) - Operation not permitted
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Percebe-se, no entanto, que a operação não foi permitida para o processo “cron”, pois o comando não foi executado com a permissão de superusuário. Para resolver, basta acrescentar o comando “sudo” antes dos demais comandos e referenciar o sinal pelo seu número, no caso 9. Dessa forma, “sudo kill -9 933” é executado, como mostra a figura 05.

Figura 05 – Execução do comando “sudo kill -9 933”



```
jonasfortes@jonasfortes-Nitro-AN515-43: ~  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ sudo kill SIGKILL 933  
[sudo] password for jonasfortes:  
kill: failed to parse argument: 'SIGKILL'  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ sudo kill -9 933  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ pgrep cron  
4816  
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Observa-se, na figura 05, que houve uma falha ao passar o sinal SIGKILL, mas não ocorreu problemas ao substituí-lo pelo seu número referente da lista de sinais. Também percebe-se que o processo “cron” foi iniciado automaticamente após ser encerrado, pois ao consultar seu PID, ele reaparece com um identificador diferente.

Após isso, outro comando foi verificado. É o comando “bg”, o qual coloca um processo que está sendo executado em primeiro plano, em segundo plano. Para demonstrá-lo, o comando “top” foi usado como exemplo. Dessa forma, se for necessário iniciar o comando “top” em segundo plano, pode-se usar “top &” como é mostrado na figura 06.

Figura 06 – Execução do comando “top &”



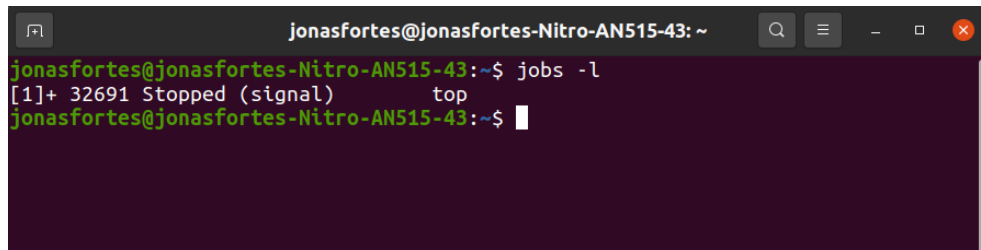
```
jonasfortes@jonasfortes-Nitro-AN515-43: ~  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ top &  
[1] 32691  
jonasfortes@jonasfortes-Nitro-AN515-43:~$ jobs  
[1]+  Stopped                  top  
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Na figura 06, observa-se que após iniciar o comando “top” em segundo plano surge o número do processo na lista de trabalhos junto ao seu PID.

Depois, pode-se observar o processo iniciado em segundo plano ao chamar o comando “jobs -l”, como mostra a figura 07.

Figura 07 – Execução do comando “jobs -l”

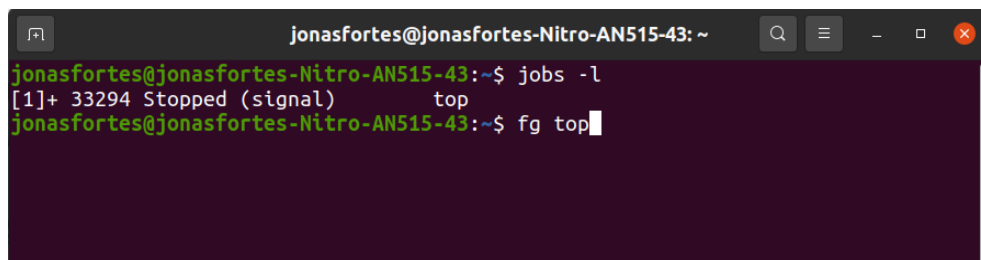
A terminal window with a dark purple background. The title bar shows 'jonasfortes@jonasfortes-Nitro-AN515-43: ~'. The prompt is 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$'. The command 'jobs -l' has been entered and executed. The output is '[1]+ 32691 Stopped (signal) top'. The prompt is now 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$' with a cursor.

```
jonasfortes@jonasfortes-Nitro-AN515-43:~$ jobs -l
[1]+ 32691 Stopped (signal) top
jonasfortes@jonasfortes-Nitro-AN515-43:~$
```

Fonte: Próprio autor

Percebe-se que o processo “top” está parado em segundo plano. E ainda, é possível colocar o processo em primeiro plano utilizando o comando “fg”, como mostra a figura 08 e 09.

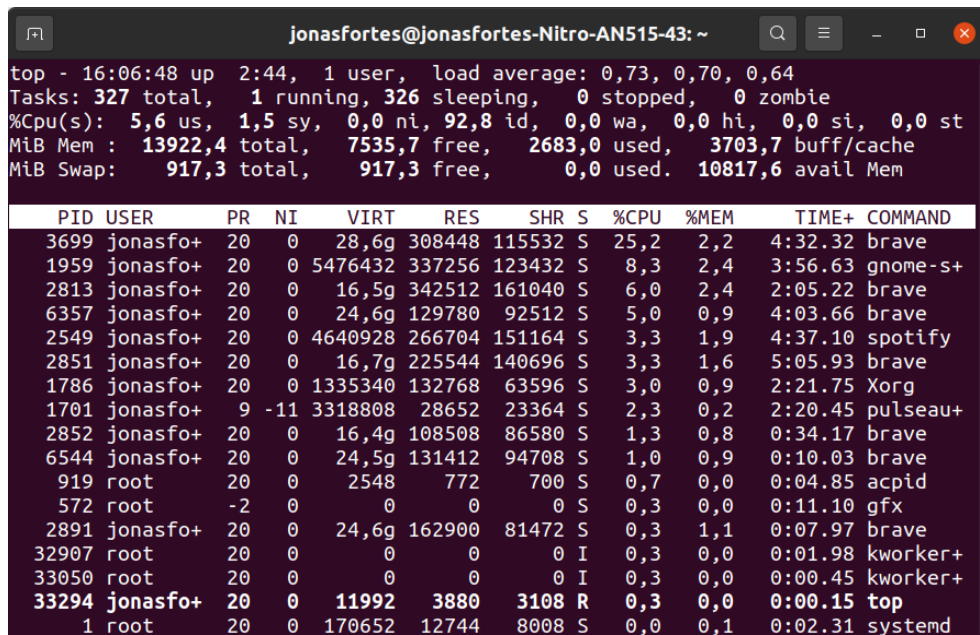
Figura 08 – Execução do comando “fg top”

A terminal window with a dark purple background. The title bar shows 'jonasfortes@jonasfortes-Nitro-AN515-43: ~'. The prompt is 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$'. The command 'jobs -l' has been entered and executed, showing '[1]+ 33294 Stopped (signal) top'. The prompt is now 'jonasfortes@jonasfortes-Nitro-AN515-43:~\$'. The command 'fg top' has been entered and is being executed, indicated by a cursor.

```
jonasfortes@jonasfortes-Nitro-AN515-43:~$ jobs -l
[1]+ 33294 Stopped (signal) top
jonasfortes@jonasfortes-Nitro-AN515-43:~$ fg top
```

Fonte: Próprio autor

Figura 09 – Execução do comando “fg top”



```

top - 16:06:48 up 2:44, 1 user, load average: 0,73, 0,70, 0,64
Tasks: 327 total, 1 running, 326 sleeping, 0 stopped, 0 zombie
%Cpu(s): 5,6 us, 1,5 sy, 0,0 ni, 92,8 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
MiB Mem : 13922,4 total, 7535,7 free, 2683,0 used, 3703,7 buff/cache
MiB Swap: 917,3 total, 917,3 free, 0,0 used. 10817,6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 3699 jonasfo+  20   0   28,6g  308448 115532 S   25,2   2,2   4:32.32 brave
 1959 jonasfo+  20   0 5476432 337256 123432 S    8,3   2,4   3:56.63 gnome-s+
 2813 jonasfo+  20   0   16,5g  342512 161040 S    6,0   2,4   2:05.22 brave
 6357 jonasfo+  20   0   24,6g  129780  92512 S    5,0   0,9   4:03.66 brave
 2549 jonasfo+  20   0 4640928 266704 151164 S    3,3   1,9   4:37.10 spotify
 2851 jonasfo+  20   0   16,7g  225544 140696 S    3,3   1,6   5:05.93 brave
 1786 jonasfo+  20   0 1335340 132768  63596 S    3,0   0,9   2:21.75 Xorg
 1701 jonasfo+   9 -11 3318808  28652  23364 S    2,3   0,2   2:20.45 pulseau+
 2852 jonasfo+  20   0   16,4g  108508  86580 S    1,3   0,8   0:34.17 brave
 6544 jonasfo+  20   0   24,5g  131412  94708 S    1,0   0,9   0:10.03 brave
   919 root      20   0    2548    772    700 S    0,7   0,0   0:04.85 acpid
   572 root      -2   0         0         0     0 S    0,3   0,0   0:11.10 gfx
 2891 jonasfo+  20   0   24,6g  162900  81472 S    0,3   1,1   0:07.97 brave
 32907 root      20   0         0         0     0 I    0,3   0,0   0:01.98 kworker+
 33050 root      20   0         0         0     0 I    0,3   0,0   0:00.45 kworker+
33294 jonasfo+  20   0   11992   3880   3108 R    0,3   0,0   0:00.15 top
     1 root      20   0  170652  12744  8008 S    0,0   0,1   0:02.31 systemd
  
```

Fonte: Próprio autor

Dessa forma, mostradas nas figuras 08 e 09, percebe-se que o comando “top” retorna ao primeiro plano.

A segunda parte deste relatório trata dos processos na linguagem C. Inicialmente, foi criado um programa que recebe parâmetros na chamada principal e exibe metade dos parâmetros pelo processo pai e outra metade pelo processo filho. O código do programa está exposto a seguir.

Código 01 – Programa em C - item 2-1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

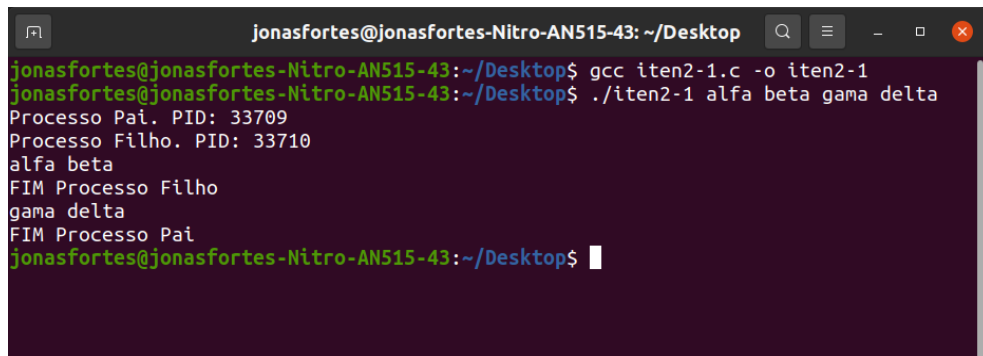
int main(int argc, char *argv[]){
    pid_t pid;
    int i;

    if((pid = fork()) < 0){
        perror("Erro!");
        exit(1);
    }
    else if (pid == 0){
        //Execução primeira metade pelo processo filho
        printf("Processo Filho. PID: %d\n", getpid());
        for(i=1; i <= argc/2; i++){
            printf("%s ", argv[i]);
        }
        printf("\nFIM Processo Filho\n");
    }else{
        //Execução segunda metade pelo processo pai
        printf("Processo Pai. PID: %d\n", getpid());
        wait(0); // espera o Filho terminar
        for(i=(argc/2)+1; i < argc; i++){
            printf("%s ", argv[i]);
        }
        printf("\nFIM Processo Pai\n");
    }
    return 0;
}
```

Fonte: Próprio autor

No código 01, nota-se que a chamada do comando “fork” está retornando um valor para a variável ‘pid’, a qual é usada para verificar qual processo está sendo executado, o pai ou o filho. Dessa forma, parte dos argumentos armazenados no vetor ‘argv’ são executados no processo filho, e depois a outra parte é executada no processo pai. É possível verificar a execução do programa no terminal, como mostra a figura 10.

Figura 10 – Execução do programa iten2-1.c



```
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ gcc iten2-1.c -o iten2-1
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ ./iten2-1 alfa beta gama delta
Processo Pai. PID: 33709
Processo Filho. PID: 33710
alfa beta
FIM Processo Filho
gama delta
FIM Processo Pai
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$
```

Fonte: Próprio autor

A figura 10 mostra que primeiramente o programa foi compilado. Após isso, o programa é chamado juntamente com os parâmetros ‘*alfa*’, ‘*beta*’, ‘*gama*’ e ‘*delta*’. Ou seja, o processo pai é criado e logo depois cria o processo filho, este por sua vez exibe os dois primeiros parâmetros e finaliza; depois o processo pai exibe os dois últimos parâmetros e finaliza também. Portanto, o programa de fato dividiu a exibição dos parâmetros entre os dois processos, como o esperado.

Depois, outro programa foi feito para receber os comandos a serem executados, por parâmetro. Isto é, o programa deve executar todos os comandos e esperar que terminem antes de finalizar o processo. O código do programa é exibido a seguir.

Código 02 – Programa em C - item2-2.c

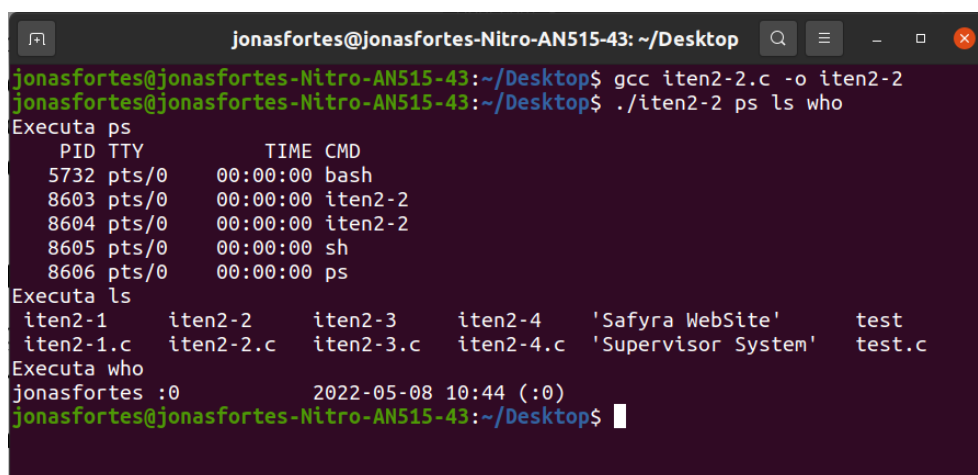
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    pid_t pid;
    if((pid = fork()) < 0){
        perror("Erro!");
        exit(1);
    }
    else if (pid == 0){
        //Execução pelo processo filho
        for(int i=1; i < argc; i++){
            printf("Executa %s \n", argv[i]);
            system(argv[i]);
        }
    }else{
        //Execução pelo processo pai
        wait(0); //espera o processo filho, ou seja, o término dos
comandos
    }
    return 0;
}
```

Fonte: Próprio autor

No código 02, percebe-se que há a criação de um processo filho no qual todos os comandos serão executados, e dessa forma, o processo pai só irá finalizar quando o filho terminar as execuções dos comandos. Pode-se observar o comportamento do programa na figura 11.

Figura 11 – Execução do programa iten2-2.c



```
jonasfortes@jonasfortes-Nitro-AN515-43: ~/Desktop
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ gcc iten2-2.c -o iten2-2
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ ./iten2-2 ps ls who
Executa ps
  PID TTY          TIME CMD
  5732 pts/0    00:00:00 bash
  8603 pts/0    00:00:00 iten2-2
  8604 pts/0    00:00:00 iten2-2
  8605 pts/0    00:00:00 sh
  8606 pts/0    00:00:00 ps
Executa ls
iten2-1   iten2-2   iten2-3   iten2-4   'Safyra WebSite'  test
iten2-1.c iten2-2.c iten2-3.c iten2-4.c 'Supervisor System' test.c
Executa who
jonasfortes :0                2022-05-08 10:44 (:0)
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$
```

Fonte: Próprio autor

Observa-se, na figura 11, que o programa é chamado e passa os comandos ‘ps’, ‘ls’ e ‘who’ para serem executados. Assim, as execuções dos comandos são feitas em ordem, antes do processo finalizar.

O próximo programa a ser analisado gera um processo filho e através deste inicia um aplicativo qualquer do sistema operacional. Para este exemplo, utilizou-se o Spotify para ser executado pelo programa. O código está exposto a seguir.

Código 03 – Programa em C - item2-3.c

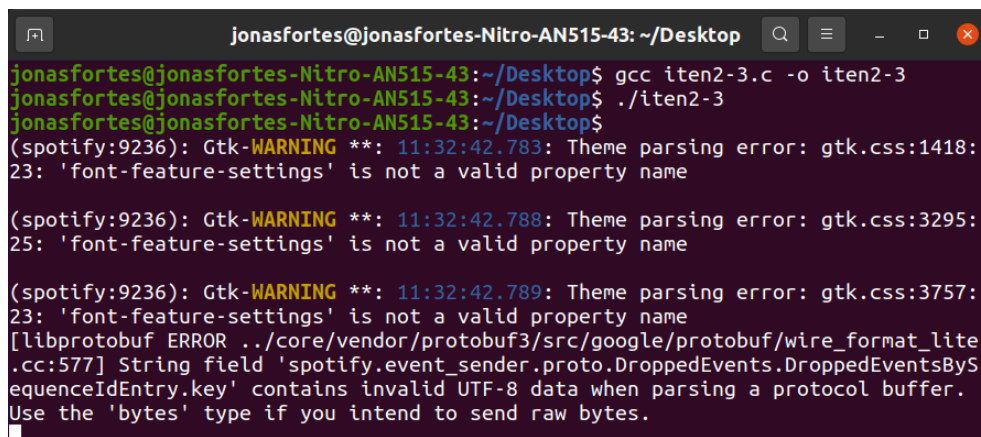
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    pid_t pid;
    if((pid = fork()) < 0){
        perror("Erro!");
        exit(1);
    }
    else if (pid == 0){
        //Execução processo filho
        system("spotify");
    }
    return 0;
}
```

Fonte: Próprio autor

No código 03, um processo filho é criado e o único comando nele faz a execução do Spotify, enquanto o processo pai apenas cria o processo filho e finaliza. O programa pode ser testado no terminal também, como mostra a figura 12.

Figura 12 – Execução do programa item2-3.c



```
jonasfortes@jonasfortes-Nitro-AN515-43: ~/Desktop
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ gcc item2-3.c -o item2-3
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ ./item2-3
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$
(spotify:9236): Gtk-WARNING **: 11:32:42.783: Theme parsing error: gtk.css:1418:
23: 'font-feature-settings' is not a valid property name

(spotify:9236): Gtk-WARNING **: 11:32:42.788: Theme parsing error: gtk.css:3295:
25: 'font-feature-settings' is not a valid property name

(spotify:9236): Gtk-WARNING **: 11:32:42.789: Theme parsing error: gtk.css:3757:
23: 'font-feature-settings' is not a valid property name
[libprotobuf ERROR ../core/vendor/protobuf3/src/google/protobuf/wire_format_lite
.cc:577] String field 'spotify.event_sender.proto.DroppedEvents.DroppedEventsByS
equenceIdEntry.key' contains invalid UTF-8 data when parsing a protocol buffer.
Use the 'bytes' type if you intend to send raw bytes.
```

Fonte: Próprio autor

Na figura 11, há algumas mensagens referentes à execução do Spotify, e após isto, o programa permanece aberto e pronto para o uso.

O último programa foi feito para demonstrar uma hierarquia de execução entre os processos pai, filho e filho1. Ou seja, o processo pai cria um processo filho, e este por sua vez cria um processo filho1, de modo que a execução de um espere o término do outro: primeiro o filho1 termina sua execução para que o filho finalize a sua, e assim o processo pai possa finalmente terminar. O programa é mostrado a seguir.

Código 04 – Programa em C - item2-4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    pid_t pid;
    pid_t pid2;

    if((pid = fork()) < 0){

        perror("Erro!");
        exit(1);
    }
    else if (pid == 0){
        if((pid2 = fork()) < 0){
            perror("Erro!");
            exit(1);
        }
        else if (pid2 == 0){
            //Execução pelo processo filho1

            printf("\nExecução Processo Filho1 ->");
        }else{
            //Execução pelo processo filho
        }
    }
}
```

```

        wait(0); //filho espera filho1
        printf("Execução Processo Filho ->");
    }
} else {

    //Execução processo pai
    wait(0); //pai espera filho
    printf("Execução Processo Pai\n");
}
return 0;
}

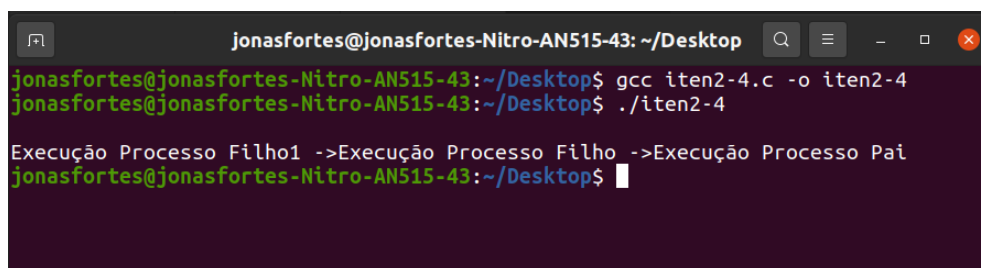
```

Fonte: Próprio autor

Ao analisar o código 04, percebe-se que há duas chamadas do comando ‘fork’, uma para criar o processo filho e outra para criar o filho1. Dessa forma, foi preciso usar a função ‘wait’ para fazer com que o processo pai esperasse o término do filho e também para que o filho esperasse o término do processo filho1. Dessa forma, foi garantido a hierarquia de execução entre os processos.

A execução do programa é mostrada na figura 13 a seguir.

Figura 13 – Execução do programa iten2-4.c



```

jonasfortes@jonasfortes-Nitro-AN515-43: ~/Desktop
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ gcc iten2-4.c -o iten2-4
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$ ./iten2-4

Execução Processo Filho1 ->Execução Processo Filho ->Execução Processo Pai
jonasfortes@jonasfortes-Nitro-AN515-43:~/Desktop$

```

Fonte: Próprio autor

Na figura 13, nota-se que as execuções seguiram a hierarquia definida, pois cada processo espera o término da execução de seu respectivo filho.

Por fim, basta definir as diferenças entre o comando 'fork' e o 'clone', do linux, uma vez que ambos têm funções semelhantes, mas não iguais. O comando 'fork' cria uma cópia idêntica do processo em que ele é chamado, com seus próprios espaços de endereçamento na memória. Já o comando 'clone' é uma chamada de sistema para criar um novo processo filho que compartilha os espaços de memória, estados de processos e PID com o processo pai. A principal diferença entre o 'fork' e o 'clone' é a estrutura de dados que é compartilhada ou não.

4. REFERÊNCIA BIBLIOGRÁFICA

PORQUE USAR PERROR?. [S. l.], 2013. Disponível em: <https://www.vivaolinux.com.br/topico/C-C++/Como-usar-perror>. Acesso em: 8 maio 2022.

SISTEMAS-OPERACIONAIS. [S. l.], 2021. Disponível em: <https://github.com/isaias0rt0n/sistemas-operacionais/blob/main/README.md>. Acesso em: 8 maio 2022.

PROCESS Identification (pid_t) data type in C language. [S. l.], 2018. Disponível em: https://www.includehelp.com/c/process-identification-pid_t-data-type.aspx. Acesso em: 8 maio 2022.

THE DIFFERENCE between fork(), vfork(), exec() and clone(). [S. l.], 2020. Disponível em: [https://stackoverflow.com/questions/4856255/the-difference-between-fork-vfork-exec-and-clone#:~:text=clone\(\)%20is%20the%20syscall,etc\)%20are%20shared%20or%20not](https://stackoverflow.com/questions/4856255/the-difference-between-fork-vfork-exec-and-clone#:~:text=clone()%20is%20the%20syscall,etc)%20are%20shared%20or%20not). Acesso em: 8 maio 2022.