

”

E-fólio A | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Sistemas Operativos

CÓDIGO: 21111

DOCENTE: Paulo Shirley, Gracinda Carvalho, José Coelho

A preencher pelo estudante

NOME: Ivo Vieira Baptista

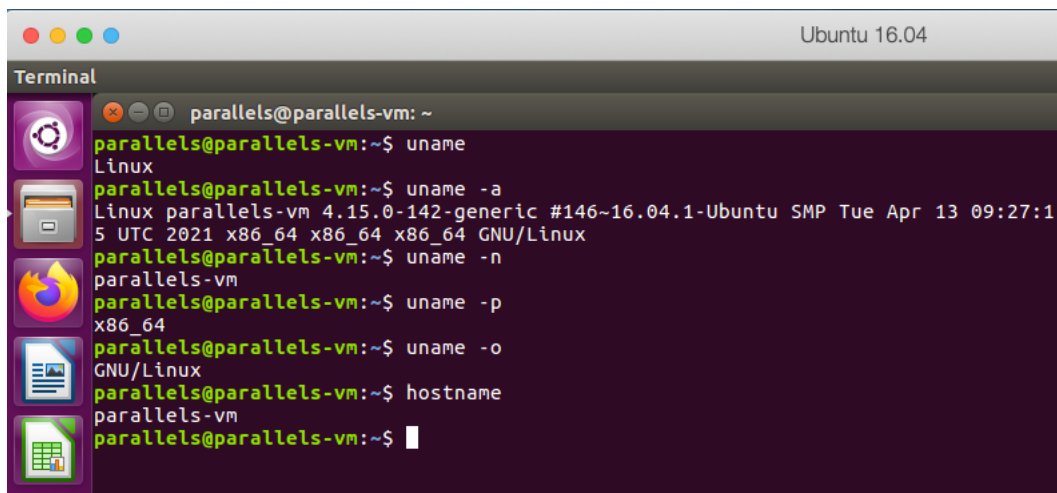
N.º DE ESTUDANTE: 2100927

CURSO: Licenciatura em Engenharia em Informática

DATA DE ENTREGA: 11 de Abril de 2022

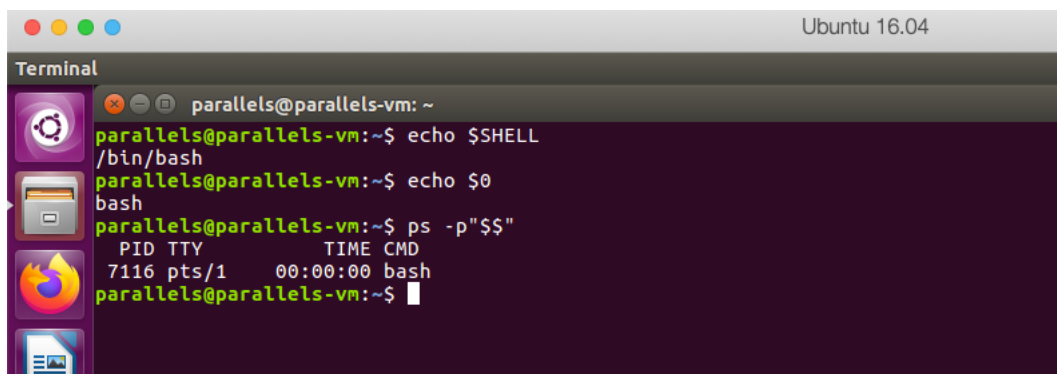
TRABALHO / RESOLUÇÃO:

Este trabalho foi desenvolvido com o Sistema Operativo GNU/Linux, Ubuntu 16.04, instalado numa maquina virtual com o Parallels-vm 4.15 como vemos a continuação, com os comandos `uname` e `hostname` verificamos o **kernel** , a maquina, o processador, e o sistema operacional que estamos a utilizar. No `hostname` conseguimos ver o host/network da rede neste exemplo na máquina virtual **parallels-vm**.

A terminal window titled 'Terminal' with a title bar 'Ubuntu 16.04'. The prompt is 'parallels@parallels-vm: ~'. The user enters several 'uname' commands: 'uname' returns 'Linux'; 'uname -a' returns 'Linux parallels-vm 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux'; 'uname -n' returns 'parallels-vm'; 'uname -p' returns 'x86_64'; 'uname -o' returns 'GNU/Linux'. Then the user enters 'hostname' which returns 'parallels-vm'.

```
parallels@parallels-vm: ~  
parallels@parallels-vm:~$ uname  
Linux  
parallels@parallels-vm:~$ uname -a  
Linux parallels-vm 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux  
parallels@parallels-vm:~$ uname -n  
parallels-vm  
parallels@parallels-vm:~$ uname -p  
x86_64  
parallels@parallels-vm:~$ uname -o  
GNU/Linux  
parallels@parallels-vm:~$ hostname  
parallels-vm  
parallels@parallels-vm:~$
```

Abaixo mostro a shell utilizada, o bash. Costumo utilizar o zsh no mac IOs, que é mais personalizavel, mas como os exemplos das actividades foram feitos com o bash, segue exemplo onde podemos ver o PID da instancia atual do shell com o comando `ps -p"$$_"` :

A terminal window titled 'Terminal' with a title bar 'Ubuntu 16.04'. The prompt is 'parallels@parallels-vm: ~'. The user enters 'echo \$SHELL' which returns '/bin/bash'. Then 'echo \$0' which returns 'bash'. Finally, 'ps -p"\$\$_"' which returns a table with columns 'PID', 'TTY', 'TIME', and 'CMD'. The table shows one entry: '7116 pts/1 00:00:00 bash'.

```
parallels@parallels-vm: ~  
parallels@parallels-vm:~$ echo $SHELL  
/bin/bash  
parallels@parallels-vm:~$ echo $0  
bash  
parallels@parallels-vm:~$ ps -p"$$_"  
  PID TTY          TIME CMD  
 7116 pts/1    00:00:00 bash  
parallels@parallels-vm:~$
```

Ferramentas utilizadas para o trabalho foi o editor **nano** e **vi** do terminal. Em vez de utilizar só comandos **ps**, costumo utilizar o **htop**, na instalação de servers, que recomendo para ver os processos em forma visual mais especifica. Segue-se alguns prints da ferramenta utilizada:

Ubuntu 16.04

Terminal

parallels@parallels-vm: ~

Tasks: 116, 247 thr: 1 running
Load average: 0.21 0.11 0.09
Uptime: 06:02:00

PID	USER	PRI	NI	VRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1120	root	20	0	361M	57352	9500	S	0.7	5.7	3:44.61	/usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/
4357	parallels	20	0	1383M	51204	15780	S	0.7	5.1	1:59.75	comptz
7106	parallels	20	0	644M	23776	16644	S	0.0	2.4	0:02.64	/usr/lib/gnome-terminal/gnome-terminal-server
10510	parallels	20	0	26288	4104	3144	R	1.3	0.4	0:00.37	htop
4096	parallels	20	0	511M	10280	5572	S	0.0	1.0	0:02.20	/usr/lib/x86_64-linux-gnu/ban/f/bamfdaemon
1350	root	20	0	361M	57352	9500	S	0.0	5.7	0:01.47	/usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/
4511	parallels	20	0	411M	2076	1776	S	0.7	0.2	2:08.09	/usr/bin/prlcc
4092	parallels	20	0	337M	2352	1280	S	0.0	0.2	0:01.46	/usr/bin/ibus-daemon --daemonize --xim --address unix:tmpdir=/tmp/ib
4453	parallels	20	0	411M	2076	1776	S	0.7	0.2	2:09.26	/usr/bin/prlcc
4074	parallels	20	0	337M	2352	1280	S	0.0	0.2	0:02.26	/usr/bin/ibus-daemon --daemonize --xim --address unix:tmpdir=/tmp/ib
6514	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:01.48	unity-control-center --overview
4278	parallels	20	0	631M	13640	7708	S	0.0	1.4	0:03.14	/usr/lib/x86_64-linux-gnu/unity/unity-panel-service
4136	parallels	20	0	183M	1772	1432	S	0.0	0.2	0:00.68	/usr/lib/ibus/ibus-engine-simple
4532	parallels	20	0	454M	2976	1720	S	0.0	0.3	0:09.95	/usr/bin/prlsga
6517	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:00.02	unity-control-center --overview
4494	parallels	20	0	454M	2976	1720	S	0.0	0.3	0:17.54	/usr/bin/prlsga
6164	root	20	0	94112	5716	3656	S	0.0	0.6	0:29.36	/usr/sbin/cupsd -l
4911	parallels	20	0	561M	14352	5884	S	0.0	1.4	0:01.39	/usr/bin/unity-scope-loader applications/applications.scope applicat
4854	parallels	20	0	821M	4408	728	S	0.0	0.4	0:00.14	/usr/lib/x86_64-linux-gnu/unity-scope-home/unity-scope-home
1	root	20	0	181M	3920	2544	S	0.0	0.4	0:04.68	/sbin/init splash
244	root	20	0	28456	2120	1836	S	0.0	0.2	0:01.16	/lib/systemd/systemd-journald
376	root	20	0	45248	1268	1156	S	0.0	0.1	0:00.32	/lib/systemd/systemd-udev

Ubuntu 16.04

Terminal

parallels@parallels-vm: ~

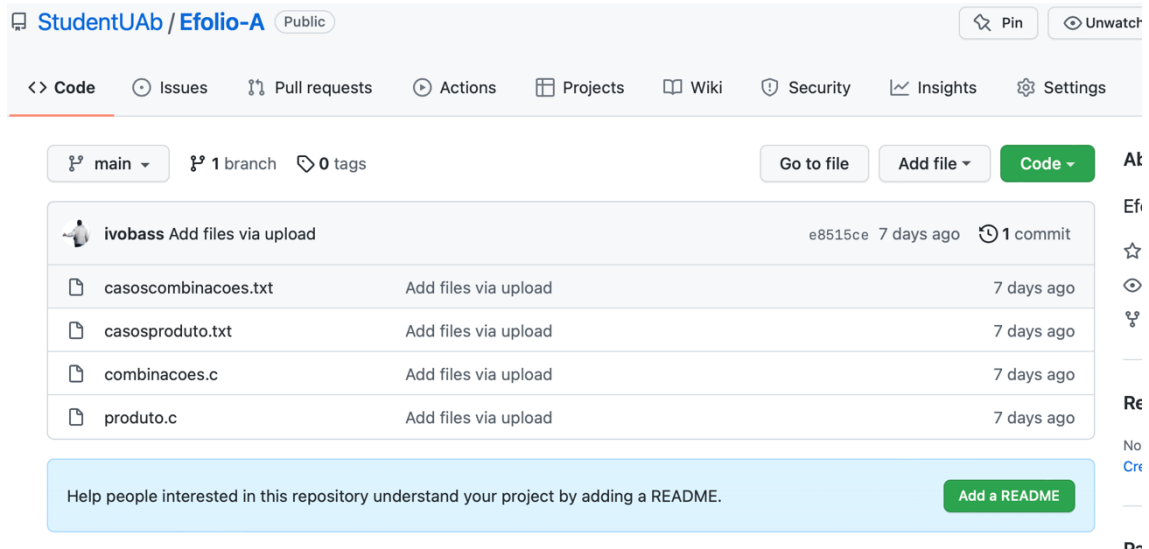
Tasks: 116, 245 thr: 1 running
Load average: 0.30 0.15 0.11
Uptime: 06:02:54

PID	USER	PRI	NI	VRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3874	parallels	20	0	46468	7592	1652	S	0.0	0.3	0:00.45	/sbin/upstart --user
28292	parallels	20	0	478M	16428	10460	S	0.0	1.6	0:01.14	/usr/lib/x86_64-linux-gnu/notify-osd
28295	parallels	20	0	478M	16428	10460	S	0.0	1.6	0:00.00	/usr/lib/x86_64-linux-gnu/notify-osd
28294	parallels	20	0	478M	16428	10460	S	0.0	1.6	0:00.00	/usr/lib/x86_64-linux-gnu/notify-osd
28293	parallels	20	0	478M	16428	10460	S	0.0	1.6	0:00.00	/usr/lib/x86_64-linux-gnu/notify-osd
7579	parallels	30	10	699M	88936	33736	S	0.0	8.9	0:09.25	/usr/bin/python3 /usr/bin/update-manager --no-update --n
7582	parallels	30	10	699M	88936	33736	S	0.0	8.9	0:00.00	/usr/bin/python3 /usr/bin/update-manager --no-update
7581	parallels	30	10	699M	88936	33736	S	0.0	8.9	0:00.00	/usr/bin/python3 /usr/bin/update-manager --no-update
7580	parallels	30	10	699M	88936	33736	S	0.0	8.9	0:00.00	/usr/bin/python3 /usr/bin/update-manager --no-update
7106	parallels	20	0	644M	24216	17028	S	0.7	2.4	0:03.32	/usr/lib/gnome-terminal/gnome-terminal-server
7116	parallels	20	0	22568	5032	3168	S	0.0	0.5	0:00.12	bash
10510	parallels	20	0	26288	4104	3144	R	1.3	0.4	0:00.96	htop
7109	parallels	20	0	644M	24216	17028	S	0.0	2.4	0:00.28	/usr/lib/gnome-terminal/gnome-terminal-server
7108	parallels	20	0	644M	24216	17028	S	0.0	2.4	0:00.00	/usr/lib/gnome-terminal/gnome-terminal-server
7107	parallels	20	0	644M	24216	17028	S	0.0	2.4	0:00.00	/usr/lib/gnome-terminal/gnome-terminal-server
6514	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:01.48	unity-control-center --overview
6519	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:00.00	unity-control-center --overview
6518	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:00.00	unity-control-center --overview
6517	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:00.02	unity-control-center --overview
6516	parallels	20	0	96G	27888	15520	S	0.0	2.8	0:00.00	unity-control-center --overview
5338	parallels	20	0	353M	2772	2028	S	0.0	0.3	0:00.03	/usr/lib/gvfs/gvfsd-dnssd --spawner :1.5 /org/gtk/gvfs/e
5340	parallels	20	0	353M	2772	2028	S	0.0	0.3	0:00.00	/usr/lib/gvfs/gvfsd-dnssd --spawner :1.5 /org/gtk/gvf
5339	parallels	20	0	353M	2772	2028	S	0.0	0.3	0:00.00	/usr/lib/gvfs/gvfsd-dnssd --spawner :1.5 /org/gtk/gvf
5292	parallels	20	0	416M	2412	1688	S	0.0	0.2	0:00.04	/usr/lib/gvfs/gvfsd-network --spawner :1.5 /org/gtk/gvf

Também utilizámos o **github** para guardar o código do efolio, podemos fazer download em qualquer lugar com o comando **git clone**:

Para Download o link:

<https://github.com/StudentUAb/Efolio-A>



Depois criamos o nosso ficheiro teste.c e executamos:

No programa teste.c começamos por colocar as bibliotecas:

`#include <stdio.h>` Manipulação de entradas e saídas

`#include <stdlib.h>` Diversas operações, incluindo conversão, geração de números pseudo-aleatórios, alocação de memória, controle de processo, sinais, busca e ordenação.

`#include <string.h>` para tratamentos de cadeia caracteres

`#include <sys/types.h>`. define muitos tipos usados em outros arquivos. Em sistemas mais antigos, era necessário incluí-lo *antes de* outros cabeçalhos do sistema.

`#include <unistd.h>` da acesso a ficheiros e pastas

Depois declaramos `#define MAXSTR 255`

Aqui o gcc diz ao compilador quando for linha a linha e encontrar MAXSTR substitui por 255

Tambem criamos uma função com o nome `compila` para chamar no `main()` como primeiro processo filho(child), neste código, `compila()` seria a função que chamaria o compilador de C para compilar o primeiro argumento (`argv[1]`)

```
void compila(char *arquivo)
{
    pid_t pid;
    if ((pid = fork()) < 0)
    {
        perror("Erro no fork da compilação");
        exit(1);
    }
    if (pid == 0)
        /* pid=0, código para o processo filho */
        {
            //primeiro processo filho seria a compilação
            printf("Processo PID = %5d  PPID = %5d [compilacao].\n", \
                (int) getpid(), (int) getppid());
            execlp("gcc", "gcc", "-o", "compilado", arquivo, NULL);
            exit(-1);
        }
    else
    {
        /* pid>0, código para o processo pai */
        wait(NULL);
    }
}
```

Aqui vemos no código que na função temos o `fork()`, validamos se é menor que 0 emite um erro, se for 0 passa a ser o processo filho(child) e se for 1 seria o processo pai(parent).

Criamos o `printf` tal como solicitado no enunciado, onde cria o primeiro filho que é a compilação do ficheiro, que se chama no nosso exemplo **compilado**, depois temos a função **main()** onde já declaramos o `argc` como inteiro onde nos vai indicar a quantidade de argumentos que serão passados ao executar o programa, também declaramos o `argv` com `char` que seria um ponteiro onde guarda o nome do programa que será chamado no prompt, também podíamos ter colocado `argv[]` em vez de `**argv`, nesta parte do programa é onde abrimos o ficheiro em modo de leitura utilizando o `fopen` e chamamos a função `compila`, e criamos um ciclo `while()` também experimentamos com `for` exemplo:

```
for(i = 0; i < 21; i++) //ciclo for para criação de processos filhos
{
    fgets(frase, MAXSTR, fln); //faz a leitura dos resultados esperados
    if ((pid = fork()) < 0)      //pid= fork() clona processo
    {
        perror("Erro no fork"); //se der erro no fork
        exit(1);                //termina o programa e devolve o erro
    }
    if (pid == 0)
        /* pid=0, código para o processo filho */
    {
        if (i == 0)              //primeira interação do ciclo for
```

continua....

Depois preferi utilizar o `while()` isto permite deixar de executar aquele `if(i == 0)` a cada iteração do loop:

```
compila();  
  
i = 0;  
  
while( fgets(frase, MAXSTR, fln) ) {  
    i++;  
  
    // código do programa  
}
```

Mas isto é uma questão de otimização. O código não está incorreto por isso.

Dentro do `while()` executamos os processos filhos(child), todos em que o nosso indicador i não seja igual a 0 (i!=0),

Maneira de declarar strings mais obscura, mas só coloquei para mostrar que existe outras formas de obter o mesmo resultado prático aqui na parte que o filho esta ser executado:

```
char ficheiro[40] = {""};  
  
snprintf(ficheiro, 40, "out%02d.txt", i-1);  
  
perror("");
```

`snprintf` guarda e formata no buffer do array os caracteres e valores, e com `perror` indica se produz uma mensagem standard de erro, também testei com o comando `strcat` para concatenar e armar os ficheiros out(i).txt, mas preferi esta forma. Seguindo com o programa visualizamos os PID e PPID dos processos filhos.

```
printf("Processo PID = %5d PPID = %5d [execucao caso %d].\n", (int) getpid(), (int) getppid(), i);
```

Depois como indicado no enunciado executamos a instrução `stdout` para redirecionar o output para um ficheiro

```
if( (stdout = freopen(ficheiro, "w", stdout)) == NULL )  
  
perror("Não consegui redirecionar stdout");
```

Seguimos com o comando `execlp` onde executamos o nosso programa `compilado` com os argumentos `compilado` que é o nosso `programa.c` e `casosprodutos.txt` que seria a variável `frase` declarada anteriormente.

Segue o código:

```
execlp("./compilado", "compilado", frase, NULL);
```

depois temos o `else` que é quando o PID é maior que 0 (`PID>0`)

Aqui começa o processo pai(parent) onde temos um `wait(NULL)`; finalizando com o `fclose()` que fecha o ficheiro que foi aberto anteriormente.

A função **wait** é implícita, Sem o **wait** o programa funciona na mesma, mas deixamos processos 'Zombi' no sistema, processos que já terminaram, mas não 'morreram', porque o pai destes não chamou o '**wait**'. Ou seja, os processos ficam ocupando espaço na tabela de processos. Se executarmos o programa muitas vezes, vai saturar a tabela de processos, e o sistema terá problemas para executar outros processos.

Concluimos a importância do **wait** que bloqueará o processo pai até que qualquer um de seus filhos termine. Se o filho terminar antes que o processo pai alcance `wait(NULL)`, o processo filho se transformará em um processo zombi, até que seu pai espere por ele e seja liberado da memória. Se o processo pai não esperar pelo filho e o pai terminar primeiro, o processo filho se tornará órfão.

Em outras palavras: o processo pai será bloqueado até que o processo filho retorne um status de saída para o sistema operacional, que é então retornado ao processo pai.

Para teste o nosso programa em um ambiente de produção com **Docker** que eu costumo utilizar, podemos executar uma imagem de **Ubuntu** e entrar no container e interagir com ele:

```
docker run -it ubuntu
```

aqui temos um Linux sem nada, onde é preciso instalar o **gcc** o **git** e depois clonamos o nosso repositório do **github** e podemos trabalhar e testar o nosso programa em C.

É desta forma que costumo utilizar para fazer estes tipo de programas, recomendo o **Docker**, porque uma maquina virtual ocupa mais espaço e fica lento o nosso computador, com **Docker** temos muitas vantagens e é muito mais pratico para trabalhar.