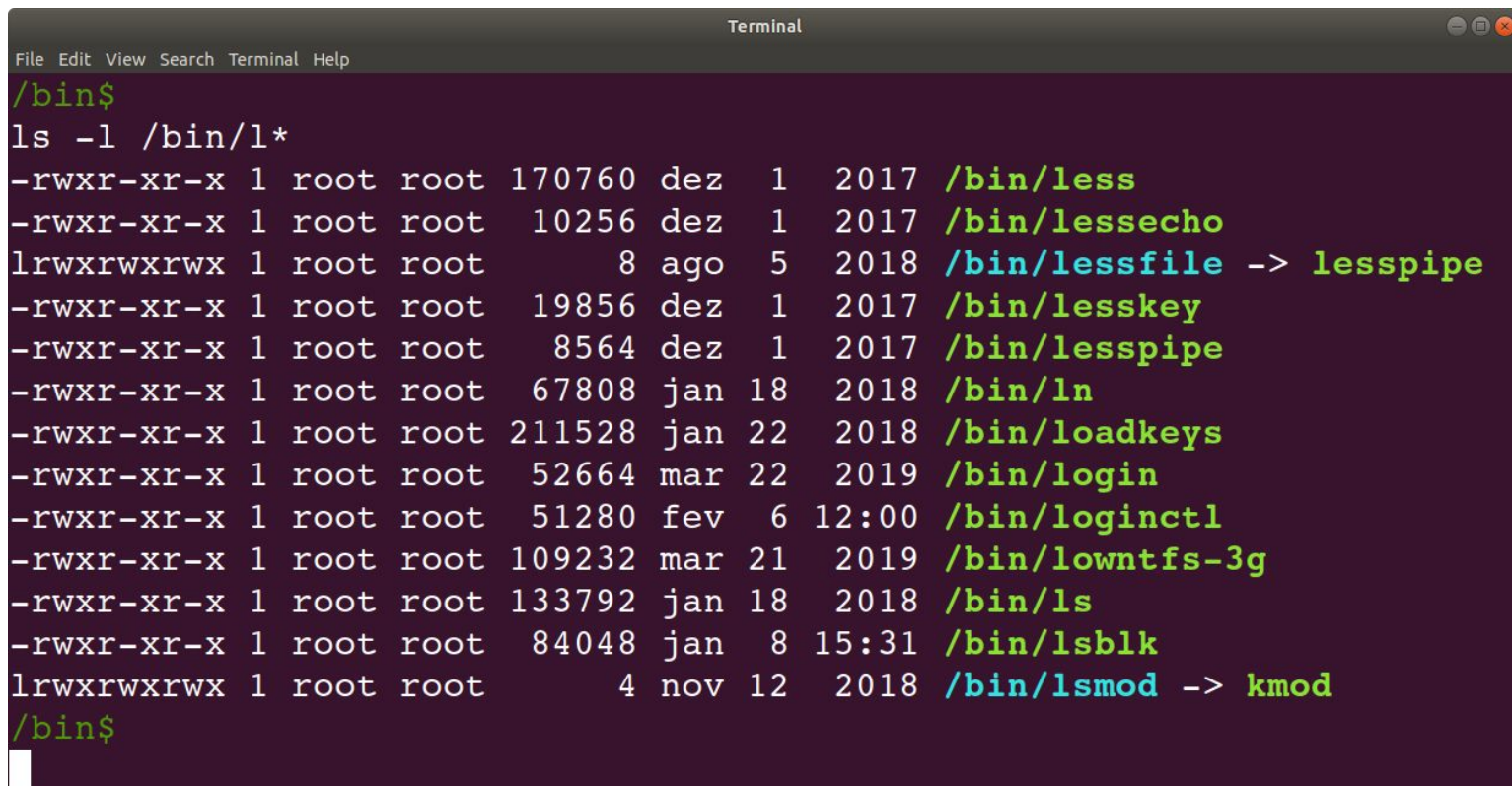


Sistemas Operacionais Embarcados

Processos

Processos

- **Definição:** um programa em execução
 - **Programa:** código em disco (passivo)
 - **Processo:** código sendo executado (ativo)

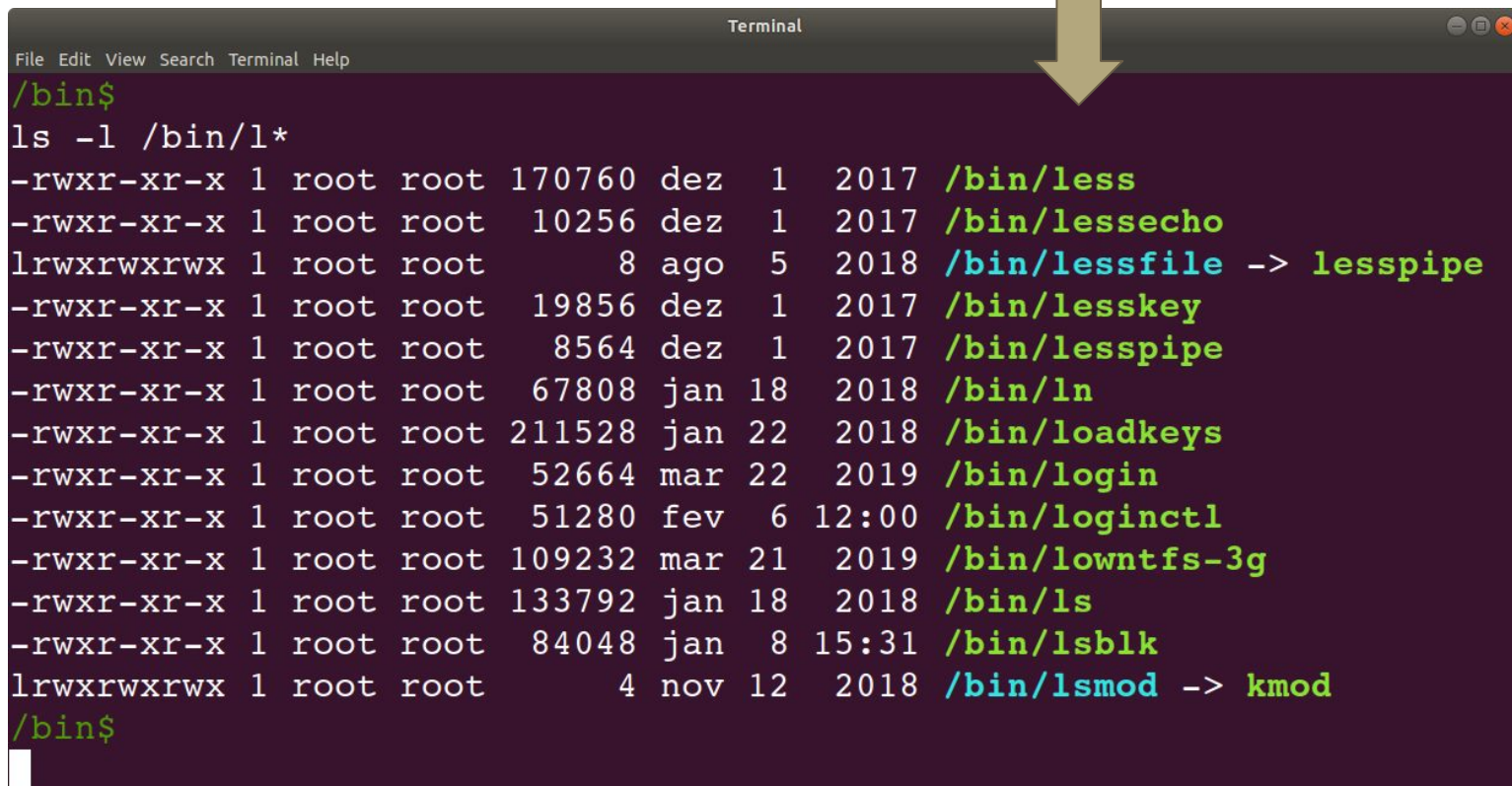


```
Terminal
File Edit View Search Terminal Help
/bin$
ls -l /bin/l*
-rwxr-xr-x 1 root root 170760 dez 1 2017 /bin/less
-rwxr-xr-x 1 root root 10256 dez 1 2017 /bin/lessecho
lrwxrwxrwx 1 root root 8 ago 5 2018 /bin/lessfile -> lesspipe
-rwxr-xr-x 1 root root 19856 dez 1 2017 /bin/lesskey
-rwxr-xr-x 1 root root 8564 dez 1 2017 /bin/lesspipe
-rwxr-xr-x 1 root root 67808 jan 18 2018 /bin/ln
-rwxr-xr-x 1 root root 211528 jan 22 2018 /bin/loadkeys
-rwxr-xr-x 1 root root 52664 mar 22 2019 /bin/login
-rwxr-xr-x 1 root root 51280 fev 6 12:00 /bin/loginctl
-rwxr-xr-x 1 root root 109232 mar 21 2019 /bin/lowntfs-3g
-rwxr-xr-x 1 root root 133792 jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 nov 12 2018 /bin/lsmode -> kmod
/bin$
```

Processos

- **Definição:** um programa em execução
 - **Programa:** código em disco (passivo)
 - **Processo:** código sendo executado

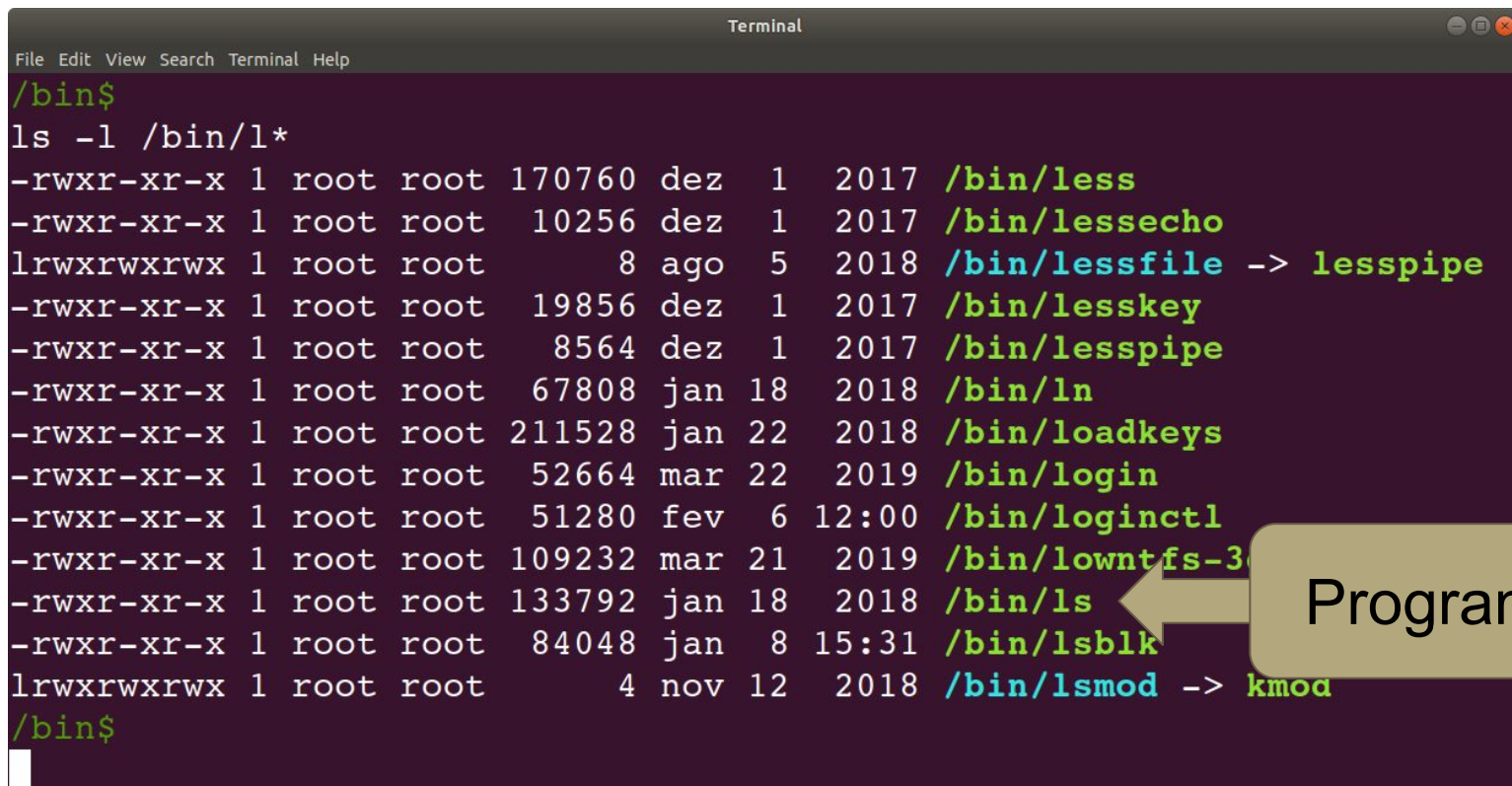
Alguns programas

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and standard window controls. The terminal shows the command `ls -l /bin/l*` and its output, which is a list of files in the /bin directory. The output includes file permissions, owner, group, size, date, time, and filename. Some files have additional information like `-> lesspipe` or `-> kmod`.

```
File Edit View Search Terminal Help
/bin$
ls -l /bin/l*
-rwxr-xr-x 1 root root 170760 dez 1 2017 /bin/less
-rwxr-xr-x 1 root root 10256 dez 1 2017 /bin/lessecho
lrwxrwxrwx 1 root root 8 ago 5 2018 /bin/lessfile -> lesspipe
-rwxr-xr-x 1 root root 19856 dez 1 2017 /bin/lesskey
-rwxr-xr-x 1 root root 8564 dez 1 2017 /bin/lesspipe
-rwxr-xr-x 1 root root 67808 jan 18 2018 /bin/ln
-rwxr-xr-x 1 root root 211528 jan 22 2018 /bin/loadkeys
-rwxr-xr-x 1 root root 52664 mar 22 2019 /bin/login
-rwxr-xr-x 1 root root 51280 fev 6 12:00 /bin/loginctl
-rwxr-xr-x 1 root root 109232 mar 21 2019 /bin/lowntfs-3g
-rwxr-xr-x 1 root root 133792 jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 nov 12 2018 /bin/lsmode -> kmod
/bin$
```

Processos

- **Definição:** um programa em execução
 - **Programa:** código em disco (passivo)
 - **Processo:** código sendo executado (ativo)

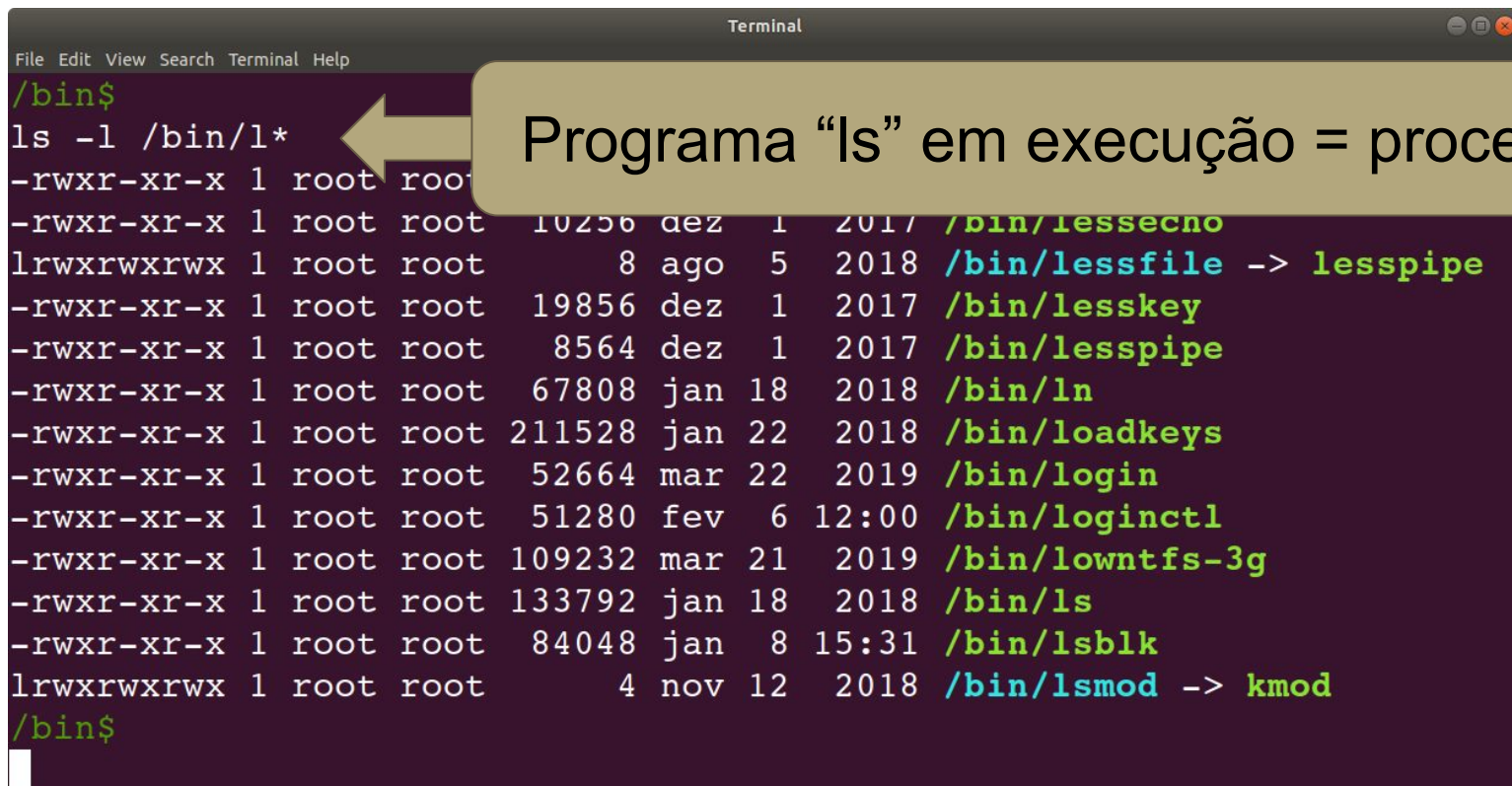


```
Terminal
File Edit View Search Terminal Help
/bin$
ls -l /bin/l*
-rwxr-xr-x 1 root root 170760 dez 1 2017 /bin/less
-rwxr-xr-x 1 root root 10256 dez 1 2017 /bin/lessecho
lrwxrwxrwx 1 root root 8 ago 5 2018 /bin/lessfile -> lesspipe
-rwxr-xr-x 1 root root 19856 dez 1 2017 /bin/lesskey
-rwxr-xr-x 1 root root 8564 dez 1 2017 /bin/lesspipe
-rwxr-xr-x 1 root root 67808 jan 18 2018 /bin/ln
-rwxr-xr-x 1 root root 211528 jan 22 2018 /bin/loadkeys
-rwxr-xr-x 1 root root 52664 mar 22 2019 /bin/login
-rwxr-xr-x 1 root root 51280 fev 6 12:00 /bin/loginctl
-rwxr-xr-x 1 root root 109232 mar 21 2019 /bin/lowntfs-3
-rwxr-xr-x 1 root root 133792 jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 nov 12 2018 /bin/lsmode -> kmod
/bin$
```

Programa "ls"

Processos

- **Definição:** um programa em execução
 - **Programa:** código em disco (passivo)
 - **Processo:** código sendo executado (ativo)



```
Terminal
File Edit View Search Terminal Help
/bin$
ls -l /bin/l*
-rwxr-xr-x 1 root root 10256 dez 1 2017 /bin/lessecho
lrwxrwxrwx 1 root root 8 ago 5 2018 /bin/lessfile -> lesspipe
-rwxr-xr-x 1 root root 19856 dez 1 2017 /bin/lesskey
-rwxr-xr-x 1 root root 8564 dez 1 2017 /bin/lesspipe
-rwxr-xr-x 1 root root 67808 jan 18 2018 /bin/ln
-rwxr-xr-x 1 root root 211528 jan 22 2018 /bin/loadkeys
-rwxr-xr-x 1 root root 52664 mar 22 2019 /bin/login
-rwxr-xr-x 1 root root 51280 fev 6 12:00 /bin/loginctl
-rwxr-xr-x 1 root root 109232 mar 21 2019 /bin/lowntfs-3g
-rwxr-xr-x 1 root root 133792 jan 18 2018 /bin/ls
-rwxr-xr-x 1 root root 84048 jan 8 15:31 /bin/lsblk
lrwxrwxrwx 1 root root 4 nov 12 2018 /bin/lsmode -> kmod
/bin$
```

Programa "ls" em execução = processo

Processos

- Cada processo é identificado por um número, o PID (*process identifier*)

```
Terminal
File Edit View Search Terminal Help
/proc$
ls -l
total 0
dr-xr-xr-x  9 root          root          0 mar 23 10:15  1
dr-xr-xr-x  9 root          root          0 mar 23 10:15 10
dr-xr-xr-x  9 root          root          0 mar 23 10:15 100
dr-xr-xr-x  9 root          root          0 mar 23 10:15 1009
dr-xr-xr-x  9 root          root          0 mar 23 10:15 101
dr-xr-xr-x  9 root          root          0 mar 23 10:15 106
dr-xr-xr-x  9 rtkit         rtkit        0 mar 23 10:15 1079
dr-xr-xr-x  9 root          root          0 mar 23 10:15 1083
dr-xr-xr-x  9 root          root          0 mar 23 10:15 11
dr-xr-xr-x  9 root          root          0 mar 23 10:15 1145
dr-xr-xr-x  9 root          root          0 mar 23 10:15 1146
dr-xr-xr-x  9 root          root          0 mar 23 10:15 115
dr-xr-xr-x  9 root          root          0 mar 23 10:15 12
dr-xr-xr-x  9 root          diogo        0 mar 23 10:16 1261
dr-xr-xr-x  9 whoopsie     whoopsie     0 mar 23 10:16 1296
dr-xr-xr-x  9 root          root          0 mar 23 10:15 13
dr-xr-xr-x  9 kernoops     adm          0 mar 23 10:16 1307
dr-xr-xr-x  9 kernoops     adm          0 mar 23 10:16 1311
```

Processos

- Cada processo é identificado por um número (process identifier)

Na pasta “/proc”, são guardadas diversas pastas numeradas, uma para cada processo

O número corresponde ao PID

File Edit View Search Terminal Help

/proc\$

ls -l

total 0

dr-xr-xr-x	9	root	root	0	mar	23	10:15	1
dr-xr-xr-x	9	root	root	0	mar	23	10:15	10
dr-xr-xr-x	9	root	root	0	mar	23	10:15	100
dr-xr-xr-x	9	root	root	0	mar	23	10:15	1009
dr-xr-xr-x	9	root	root	0	mar	23	10:15	101
dr-xr-xr-x	9	root	root	0	mar	23	10:15	106
dr-xr-xr-x	9	rtkit	rtkit	0	mar	23	10:15	1079
dr-xr-xr-x	9	root	root	0	mar	23	10:15	1083
dr-xr-xr-x	9	root	root	0	mar	23	10:15	11
dr-xr-xr-x	9	root	root	0	mar	23	10:15	1145
dr-xr-xr-x	9	root	root	0	mar	23	10:15	1146
dr-xr-xr-x	9	root	root	0	mar	23	10:15	115
dr-xr-xr-x	9	root	root	0	mar	23	10:15	12
dr-xr-xr-x	9	root	diogo	0	mar	23	10:16	1261
dr-xr-xr-x	9	whoopsie	whoopsie	0	mar	23	10:16	1296
dr-xr-xr-x	9	root	root	0	mar	23	10:15	13
dr-xr-xr-x	9	kernoops	adm	0	mar	23	10:16	1307
dr-xr-xr-x	9	kernoops	adm	0	mar	23	10:16	1311

Processos

- Cada processo é identificado por um número, o PID (*process identifier*)

```
Terminal
File Edit View Search Terminal Help
/proc$
ls -l /proc/1
ls: cannot read symbolic link '/proc/1/cwd': Permission denied
ls: cannot read symbolic link '/proc/1/root': Permission denied
ls: cannot read symbolic link '/proc/1/exe': Permission denied
total 0
dr-xr-xr-x 2 root root 0 mar 23 10:15 attr
-rw-r--r-- 1 root root 0 mar 23 18:46 autogroup
-r----- 1 root root 0 mar 23 18:46 auxv
-r--r--r-- 1 root root 0 mar 23 10:15 cgroup
--w----- 1 root root 0 mar 23 18:46 clear_refs
-r--r--r-- 1 root root 0 mar 23 10:15 cmdline
-rw-r--r-- 1 root root 0 mar 23 10:15 comm
-rw-r--r-- 1 root root 0 mar 23 18:46 coredump_filter
-r--r--r-- 1 root root 0 mar 23 18:46 cpuset
lrwxrwxrwx 1 root root 0 mar 23 10:17 cwd
-r----- 1 root root 0 mar 23 10:15 environ
lrwxrwxrwx 1 root root 0 mar 23 10:15 exe
dr-x----- 2 root root 0 mar 23 10:15 fd
dr-x----- 2 root root 0 mar 23 10:15 fdinfo
-rw-r--r-- 1 root root 0 mar 23 10:15 gid_map
```


Processos

- Cada processo *identificador*

Cada pasta contém diversas informações importantes para o processo correspondente

ro, o PID (*process*

```
File Edit View Search Terminal Help
/proc$
ls -l /proc/1
ls: cannot read symbolic link '/proc/1/c': Permission denied
ls: cannot read symbolic link '/proc/1/root': Permission denied
ls: cannot read symbolic link '/proc/1/environ': Permission denied
total 0
dr-xr-xr-x 2 root root 0 mar 23 10:15 attr
-rw-r--r-- 1 root root 0 mar 23 18:46 autogroup
-r----- 1 root root 0 mar 23 18:46 auxv
-r--r--r-- 1 root root 0 mar 23 10:15 cgroup
--w----- 1 root root 0 mar 23 18:46 clear_refs
-r--r--r-- 1 root root 0 mar 23 10:15 cmdline
-rw-r--r-- 1 root root 0 mar 23 10:15 comm
-rw-r--r-- 1 root root 0 mar 23 18:46 coredump_filter
-r--r--r-- 1 root root 0 mar 23 18:46 cpuset
lrwxrwxrwx 1 root root 0 mar 23 10:17 cwd
-r----- 1 root root 0 mar 23 10:15 environ
lrwxrwxrwx 1 root root 0 mar 23 10:15 exe
dr-x----- 2 root root 0 mar 23 10:15 fd
dr-x----- 2 root root 0 mar 23 10:15 fdinfo
-rw-r--r-- 1 root root 0 mar 23 10:15 gid_map
```

Processos


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Programas em execução:\n");
    system("ps");
    printf ("Identificador do processo (PID) é: %d\n",
            getpid ());
    printf ("Identificador do processo-pai (PPID): %d\n",
            getppid ());
    return 0;
}
```

Processos

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Programas e\n");
    system("ps");
    printf ("Identificador do processo (PID): %d\n",
            getpid ());
    printf ("Identificador do processo-pai (PPID): %d\n",
            getppid ());
    return 0;
}
```



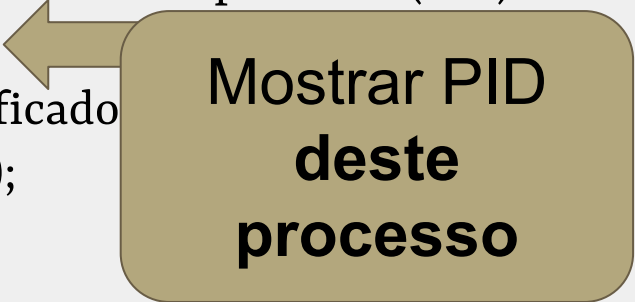
A diagram consisting of a light brown rounded rectangle with the text "Listar processos atuais" inside. A grey arrow points from the left side of this box to the `system("ps");` line in the code block above.

Code/06_Processos/Ex1.c

Processos

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Programas em execução:\n");
    system("ps");
    printf ("Identificador do processo (PID) é: %d\n",
           getpid( ));
    printf ("Identificado pelo pai (PPID) é: %d\n",
           getppid( ));
    return 0;
}
```

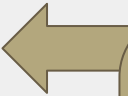


Mostrar PID deste processo

Processos

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

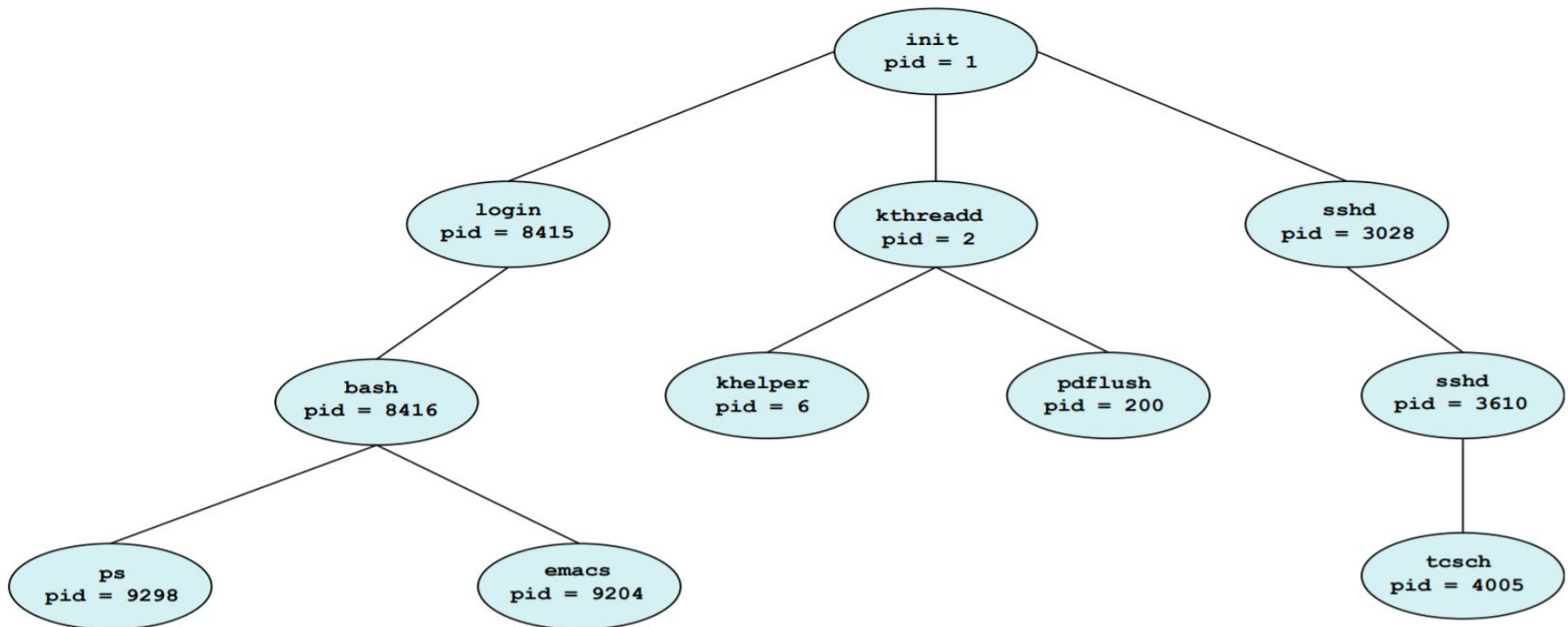
int main(void)
{
    printf("Programas em execução:\n");
    system("ps");
    printf ("Identificador do processo (PID) é: %d\n",
           getpid( ));
    printf ("Identificador do processo-pai (PPID): %d\n",
           getppid( ));
    return 0;
}
```



**Mostrar PID do
processo que
deu origem a
este processo**

Gerenciamento de Processos

- Processos são geralmente iniciados por outros processos, usando mecanismos do sistema operacional (mais à frente)



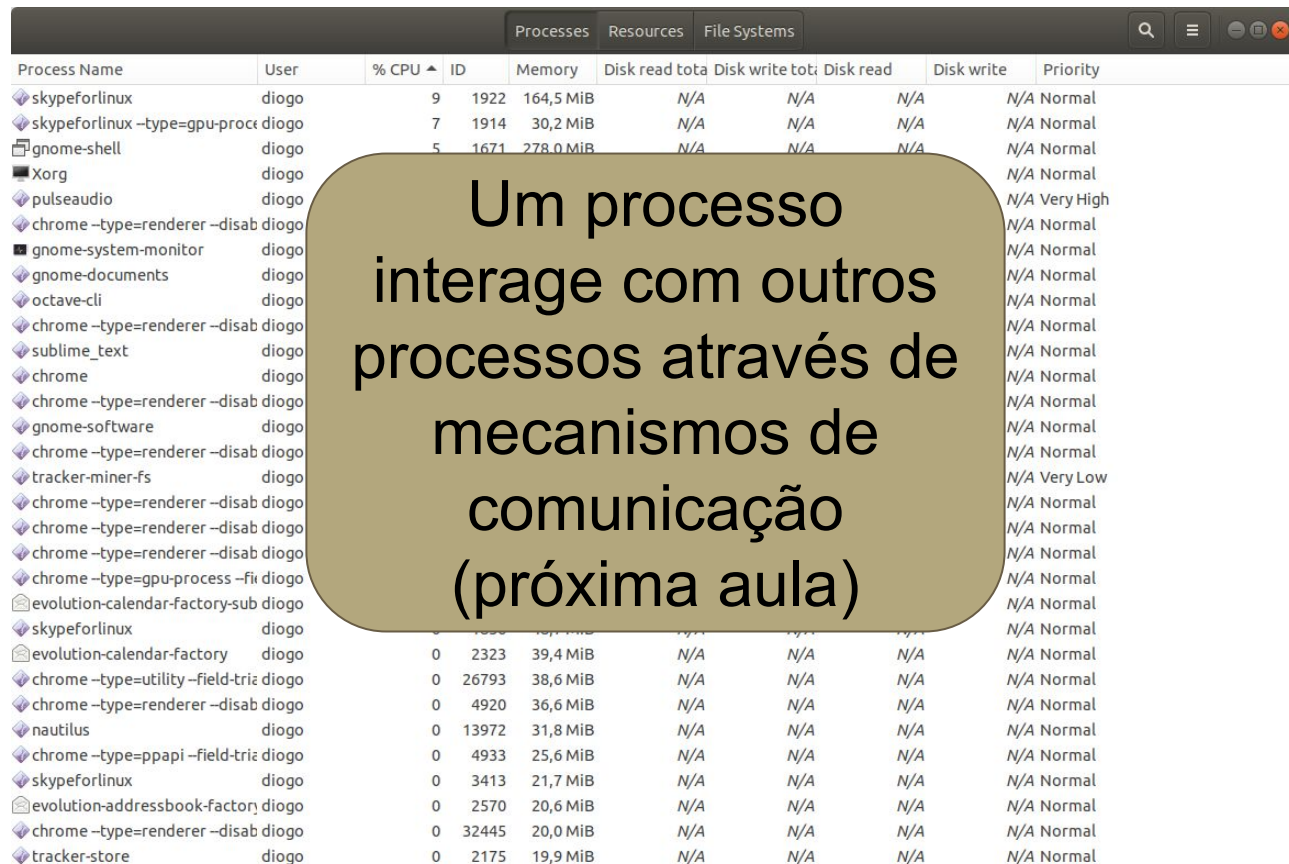
Processos

- Durante a execução, o processo compartilha o processador com outros processos em execução (escalonamento de processador).

Processes Resources File Systems									
Process Name	User	% CPU	ID	Memory	Disk read tota	Disk write tota	Disk read	Disk write	Priority
skypeforlinux	diogo	9	1922	164,5 MiB	N/A	N/A	N/A	N/A	Normal
skypeforlinux -type=gpu-proc	diogo	7	1914	30,2 MiB	N/A	N/A	N/A	N/A	Normal
gnome-shell	diogo	5	1671	278,0 MiB	N/A	N/A	N/A	N/A	Normal
Xorg	diogo	4	1442	22,7 MiB	N/A	N/A	N/A	N/A	Normal
pulseaudio	diogo	2	1694	3,8 MiB	N/A	N/A	N/A	N/A	Very High
chrome -type=renderer -disab	diogo	2	7206	122,2 MiB	N/A	N/A	N/A	N/A	Normal
gnome-system-monitor	diogo	2	7528	13,8 MiB	23,5 MiB	92,0 KiB	N/A	N/A	Normal
gnome-documents	diogo	0	1473	51,6 MiB	N/A	N/A	N/A	N/A	Normal
octave-cli	diogo	0	22796	619,6 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	4791	260,8 MiB	N/A	N/A	N/A	N/A	Normal
sublime_text	diogo	0	3534	220,4 MiB	N/A	N/A	N/A	N/A	Normal
chrome	diogo	0	26755	188,1 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	28065	171,5 MiB	N/A	N/A	N/A	N/A	Normal
gnome-software	diogo	0	3286	151,4 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	3747	141,8 MiB	N/A	N/A	N/A	N/A	Normal
tracker-miner-fs	diogo	0	2170	120,4 MiB	N/A	N/A	N/A	N/A	Very Low
chrome -type=renderer -disab	diogo	0	3949	93,3 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	28034	90,3 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	2533	81,7 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=gpu-process -fi	diogo	0	26791	78,1 MiB	N/A	N/A	N/A	N/A	Normal
evolution-calendar-factory-sub	diogo	0	2444	56,5 MiB	N/A	N/A	N/A	N/A	Normal
skypeforlinux	diogo	0	1856	48,7 MiB	N/A	N/A	N/A	N/A	Normal
evolution-calendar-factory	diogo	0	2323	39,4 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=utility -field-tri	diogo	0	26793	38,6 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	4920	36,6 MiB	N/A	N/A	N/A	N/A	Normal
nautilus	diogo	0	13972	31,8 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=ppapi -field-tri	diogo	0	4933	25,6 MiB	N/A	N/A	N/A	N/A	Normal
skypeforlinux	diogo	0	3413	21,7 MiB	N/A	N/A	N/A	N/A	Normal
evolution-addressbook-facton	diogo	0	2570	20,6 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	32445	20,0 MiB	N/A	N/A	N/A	N/A	Normal
tracker-store	diogo	0	2175	19,9 MiB	N/A	N/A	N/A	N/A	Normal

Processos

- Durante a execução, o processo compartilha o processador com outros processos em execução (escalonamento de processador).



Process Name	User	% CPU	ID	Memory	Disk read tota	Disk write tota	Disk read	Disk write	Priority
skypeforlinux	diogo	9	1922	164,5 MiB	N/A	N/A	N/A	N/A	Normal
skypeforlinux -type=gpu-proc	diogo	7	1914	30,2 MiB	N/A	N/A	N/A	N/A	Normal
gnome-shell	diogo	5	1671	278,0 MiB	N/A	N/A	N/A	N/A	Normal
Xorg	diogo								Normal
pulseaudio	diogo								Very High
chrome -type=renderer -disab	diogo								Normal
gnome-system-monitor	diogo								Normal
gnome-documents	diogo								Normal
octave-cli	diogo								Normal
chrome -type=renderer -disab	diogo								Normal
sublime_text	diogo								Normal
chrome	diogo								Normal
chrome -type=renderer -disab	diogo								Normal
gnome-software	diogo								Normal
chrome -type=renderer -disab	diogo								Normal
tracker-miner-fs	diogo								Very Low
chrome -type=renderer -disab	diogo								Normal
chrome -type=renderer -disab	diogo								Normal
chrome -type=renderer -disab	diogo								Normal
chrome -type=gpu-process -fi	diogo								Normal
evolution-calendar-factory-sub	diogo								Normal
skypeforlinux	diogo								Normal
evolution-calendar-factory	diogo	0	2323	39,4 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=utility -field-tri	diogo	0	26793	38,6 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	4920	36,6 MiB	N/A	N/A	N/A	N/A	Normal
nautilus	diogo	0	13972	31,8 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=ppapi -field-tri	diogo	0	4933	25,6 MiB	N/A	N/A	N/A	N/A	Normal
skypeforlinux	diogo	0	3413	21,7 MiB	N/A	N/A	N/A	N/A	Normal
evolution-addressbook-facton	diogo	0	2570	20,6 MiB	N/A	N/A	N/A	N/A	Normal
chrome -type=renderer -disab	diogo	0	32445	20,0 MiB	N/A	N/A	N/A	N/A	Normal
tracker-store	diogo	0	2175	19,9 MiB	N/A	N/A	N/A	N/A	Normal

Processos

Ambiente	<ul style="list-style-type: none">- Espaço de endereçamento- Processo pai/filho- Proprietário- Arquivos abertos (descritores de arquivo)- Parâmetros de chamadas em andamento- Sinais
Execução	<ul style="list-style-type: none">- Identificador do processo (PID)- Contador de programa (PC - program counter)- Apontador de pilha (SP - stack pointer)- Registradores- Estado de execução- Momento de início do processo- Estatísticas de uso (tempo de processador utilizado etc.)

Processos

Cada processo
roda
independentemente dos
demais

- Espaço de endereçamento
 - Processo pai/filho
 - Proprietário
 - Arquivos abertos (descritores de arquivo)
 - Parâmetros de chamadas em andamento
 - Sinais
-
- Identificador do processo (**PID**)
 - Contador de programa (**PC - program counter**)
 - Apontador de pilha (**SP - stack pointer**)
 - Registradores
 - Estado de execução
 - Momento de início do processo
 - Estatísticas de uso (tempo de processador utilizado etc.)

Processos

O processador
é chaveado
entre os
diversos
processos

- Espaço de endereçamento
 - Processo pai/filho
 - Proprietário
 - Arquivos abertos (descritores de arquivo)
 - Parâmetros de chamadas em andamento
 - Sinais
-
- Identificador do processo (**PID**)
 - Contador de programa (**PC - program counter**)
 - Apontador de pilha (**SP - stack pointer**)
 - Registradores
 - Estado de execução
 - Momento de início do processo
 - Estatísticas de uso (tempo de processador utilizado etc.)

Processos

O processador
é chaveado
entre os
diversos
processos

- E
-
-
-
-
-
- S

Não é possível prever o tempo de
execução de um processo
(depende da carga do sistema)

- Identificador do processo (**PID**)
- Contador de programa (**PC - program counter**)
- Apontador de pilha (**SP - stack pointer**)
- Registradores
- Estado de execução
- Momento de início do processo
- Estatísticas de uso (tempo de processador utilizado etc.)

Processos

Processo P1

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x, y;
```

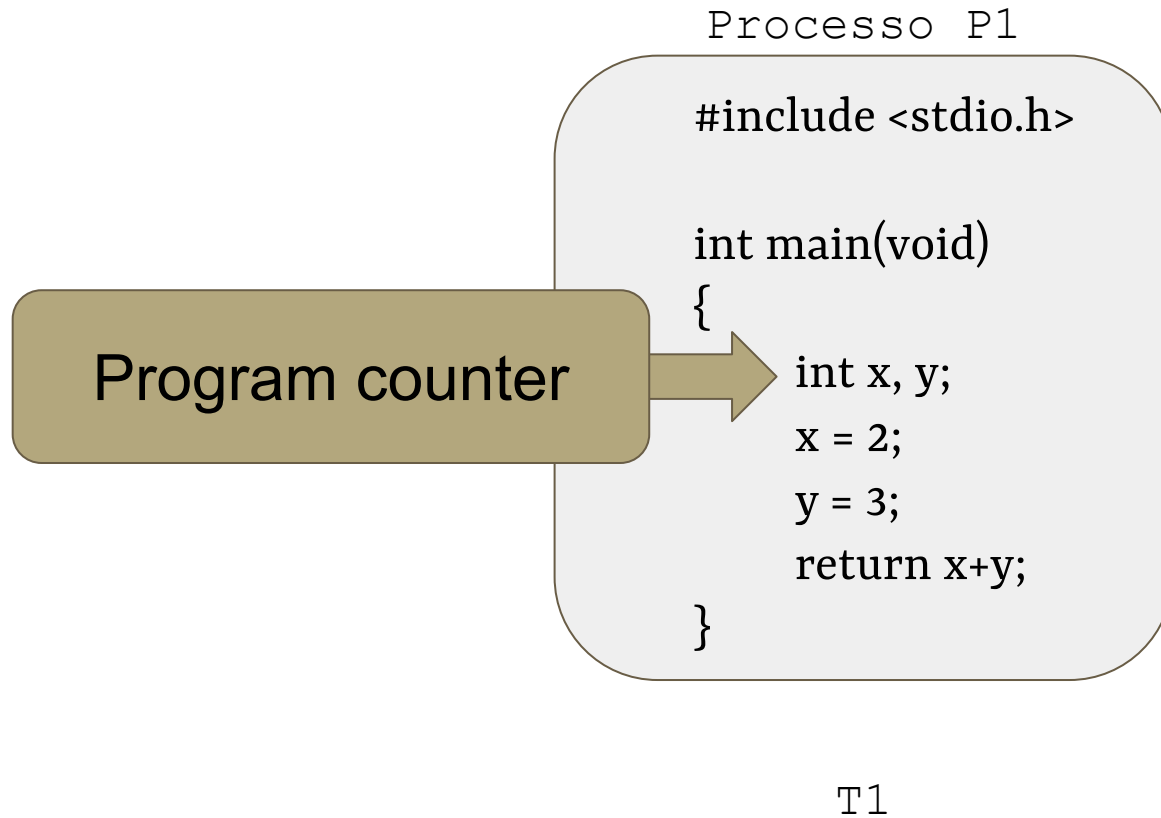
```
    x = 2;
```

```
    y = 3;
```

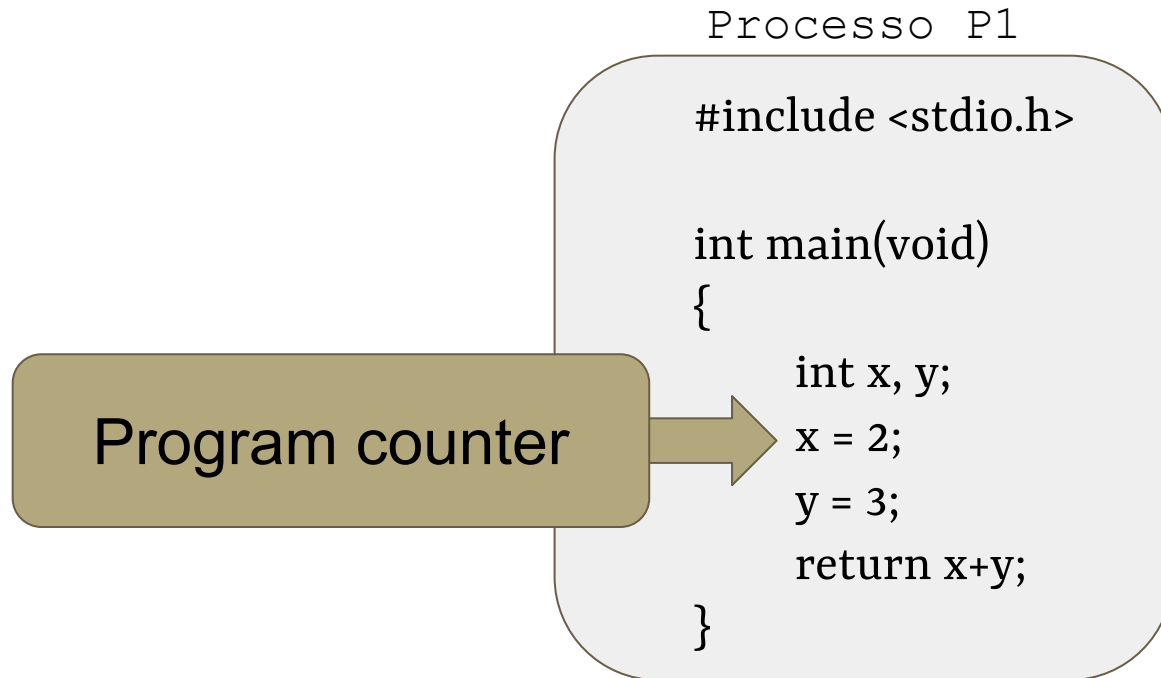
```
    return x+y;
```

```
}
```

Processos



Processos



T2

Processos

Processo P1

```
#include <stdio.h>
```

```
int main(void)  
{
```

```
    int x, y;
```

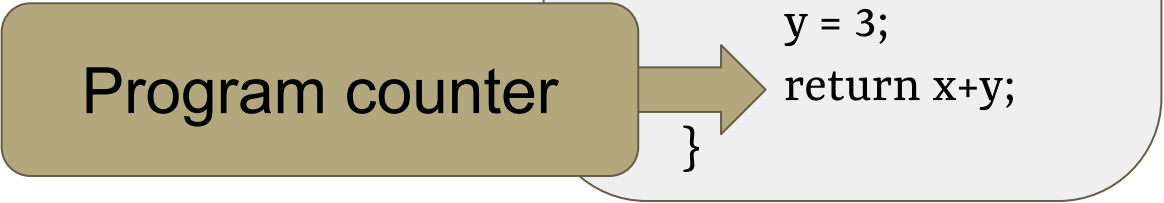
```
    x = 2;
```

```
    y = 3;
```

```
    return x+y;
```

```
}
```

Program counter



T3

Processos

Processo P1

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

PC

```
    int x, y;
```

```
    x = 2;
```

```
    y = 3;
```

```
    return x+y;
```

```
}
```

Processo P2

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    float a, b;
```

```
    a = 21.0;
```

```
    b = 30.0;
```

```
    return x+y;
```

```
}
```

Processo P3

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x, y;
```

```
    x = 2;
```

```
    y = 3;
```

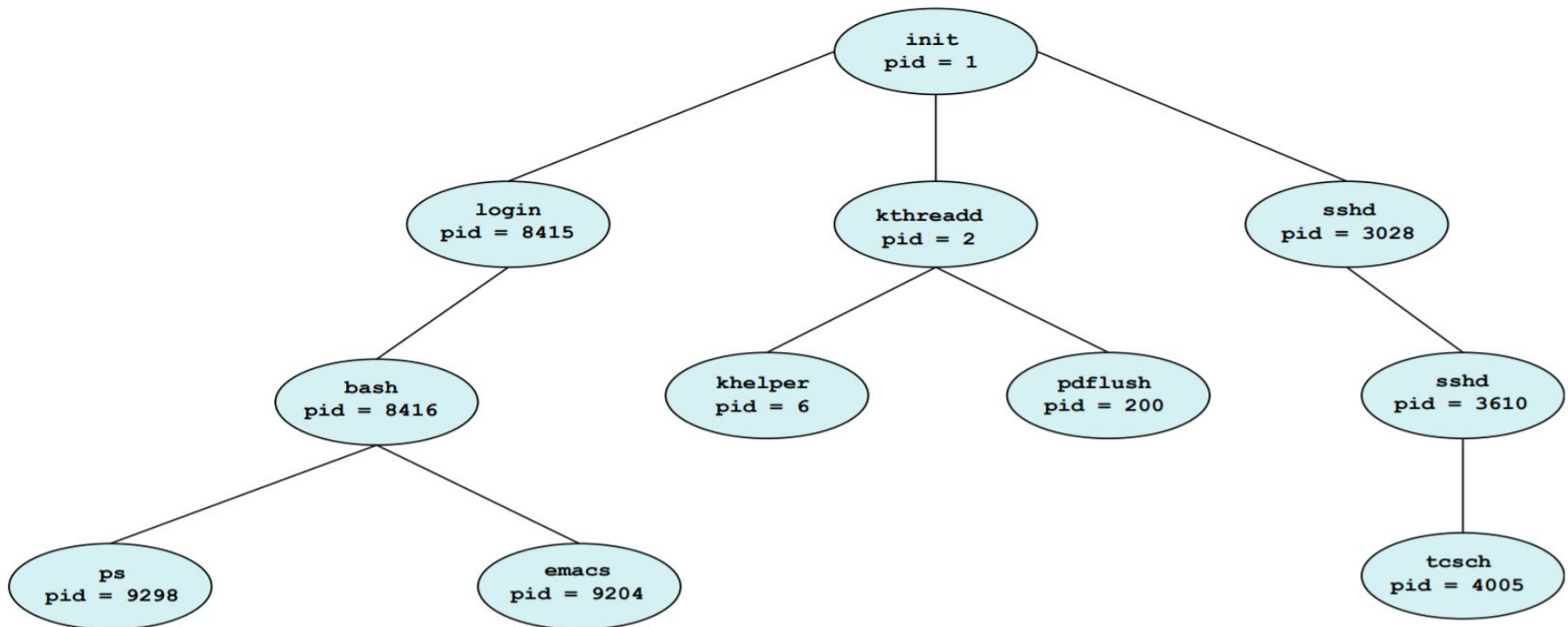
```
    return x+y;
```

```
}
```

T1

Criação de Processos

- Processos em Linux podem ser criados usando:
 - “system()”
 - “fork()” e “exec()”



```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a;
    printf("Comando: \"ls\"\\n");
    a = system("ls");
    printf("Retorno: %d\\n\\n", a);

    printf("Comando: \"ls 12345.txt\"\\n");
    a = system("ls 12345.txt");
    printf("Retorno: %d\\n\\n", a);

    printf("Comando: \"abcde\"\\n");
    a = system("abcde");
    printf("Retorno de system(\"abcde\"): %d\\n\\n", a);
    return 0;
}
```

Code/06_Processos/Ex2.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
```


```
    int a;
    printf("Comando: \"ls\"\\n");
    a = system("ls");
    printf("Retorno: %d\\n\\n", a);
```

```
    printf("Comando: \"ls 12345.txt\"\\n");
    a = system("ls 12345.txt");
    printf("Retorno: %d\\n\\n", a);
```

```
    printf("Comando: \"abcde\"\\n");
    a = system("abcde");
    printf("Retorno de system(\"abcde\"): %d\\n\\n", a);
    return 0;
```

```
}
```

Executa o programa
"ls", e apresenta na
tela o valor
retornado pelo
programa



```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
```

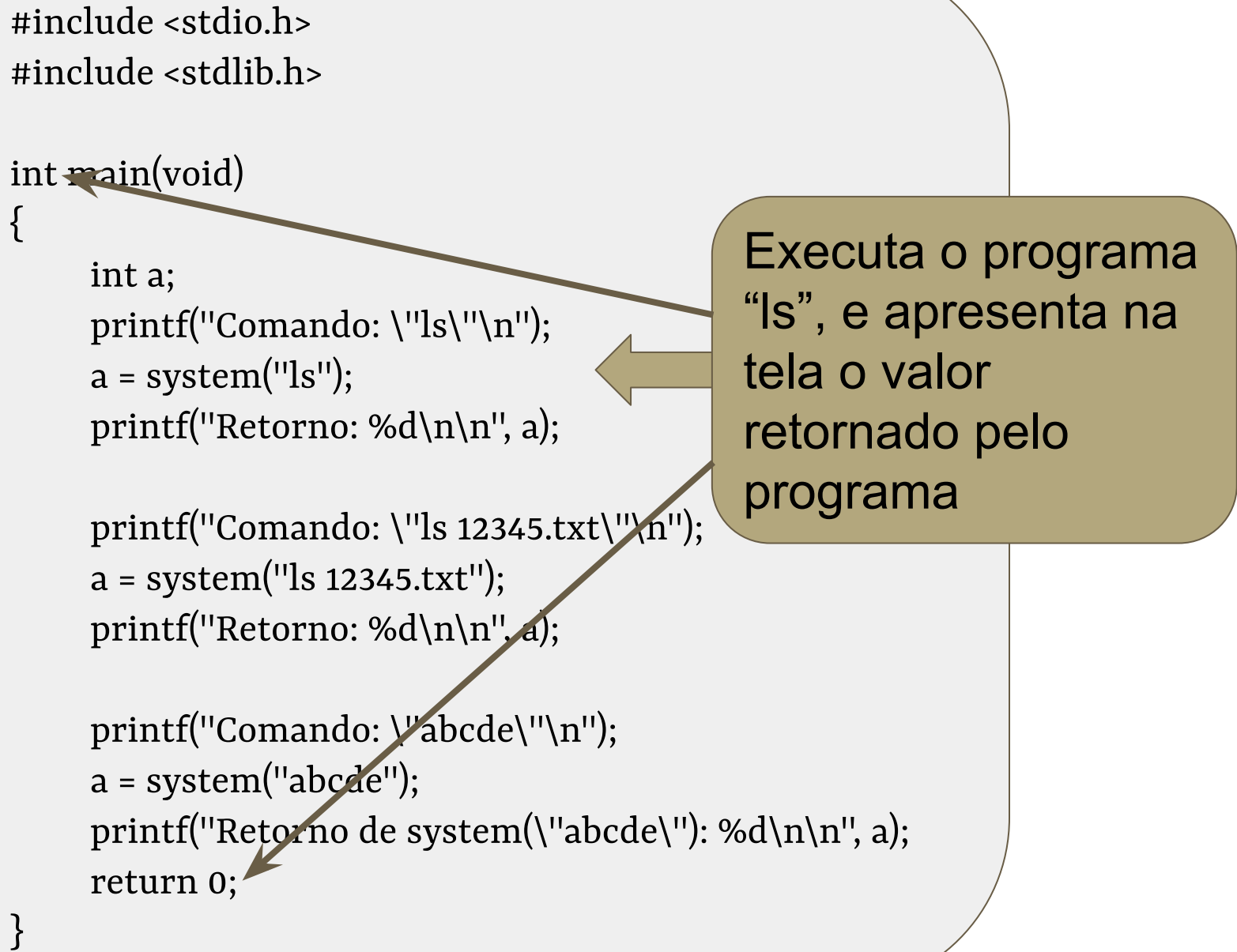
```
    int a;
    printf("Comando: \"ls\"\\n");
    a = system("ls");
    printf("Retorno: %d\\n\\n", a);
```

```
    printf("Comando: \"ls 12345.txt\"\\n");
    a = system("ls 12345.txt");
    printf("Retorno: %d\\n\\n", a);
```

```
    printf("Comando: \"abcde\"\\n");
    a = system("abcde");
    printf("Retorno de system(\"abcde\"): %d\\n\\n", a);
    return 0;
```

```
}
```

Executa o programa
"ls", e apresenta na
tela o valor
retornado pelo
programa




```
#include <stdio.h>
#include <stdlib.h>
```

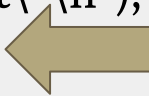
```
int main(void)
{
```

```
    int a;
    printf("Comando: \"ls\"\n");
    a = system("ls");
    printf("Retorno: %d\n\n", a);
```

```
    printf("Comando: \"ls 12345.txt\"\n");
    a = system("ls 12345.txt");
    printf("Retorno: %d\n\n", a);
```

```
    printf("Comando: \"abcde\"\n");
    a = system("abcde");
    printf("Retorno de system(\"abcde\"): %d\n\n", a);
    return 0;
}
```

Executa o programa
“ls” para buscar um
arquivo inexistente
na pasta. Repare no
valor retornado pelo
programa



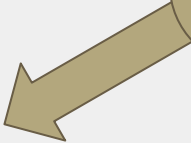
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a;
    printf("Comando: \"ls\"\\n");
    a = system("ls");
    printf("Retorno: %d\\n\\n", a);

    printf("Comando: \"ls 12345.txt\"\\n");
    a = system("ls 12345.txt");
    printf("Retorno: %d\\n\\n", a);

    printf("Comando: \"abcde\"\\n");
    a = system("abcde");
    printf("Retorno de system(\"abcde\"): %d\\n\\n", a);
    return 0;
}
```

Tenta executar um programa inexistente. Repare no valor retornado pelo programa



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
    int a;
    printf("Insira um valor inteiro: ");
    scanf("%d", &a);
    return 2*a;
}
```

Code/06_Processos/Ex3a.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n",
        a);
    printf("Dobro do valor inserido"
        " = %d\n",
        a/256);
    return 0;
}
```

Code/06_Processos/Ex3b.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Insira um valor inteiro: ");
    scanf("%d", &a);
    return 2*a;
}
```

Code/06_Processos/Ex3a.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n",
        a);
    printf("Dobro do valor inserido"
        " = %d\n",
        a/256);
    return 0;
}
```

Code/06_Processos/Ex3b.c

**Usuário insere
um valor inteiro,
e o programa
retorna o dobro
do valor**

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Insira um valor inteiro: ");
    scanf("%d", &a);
    return 2*a;
}
```

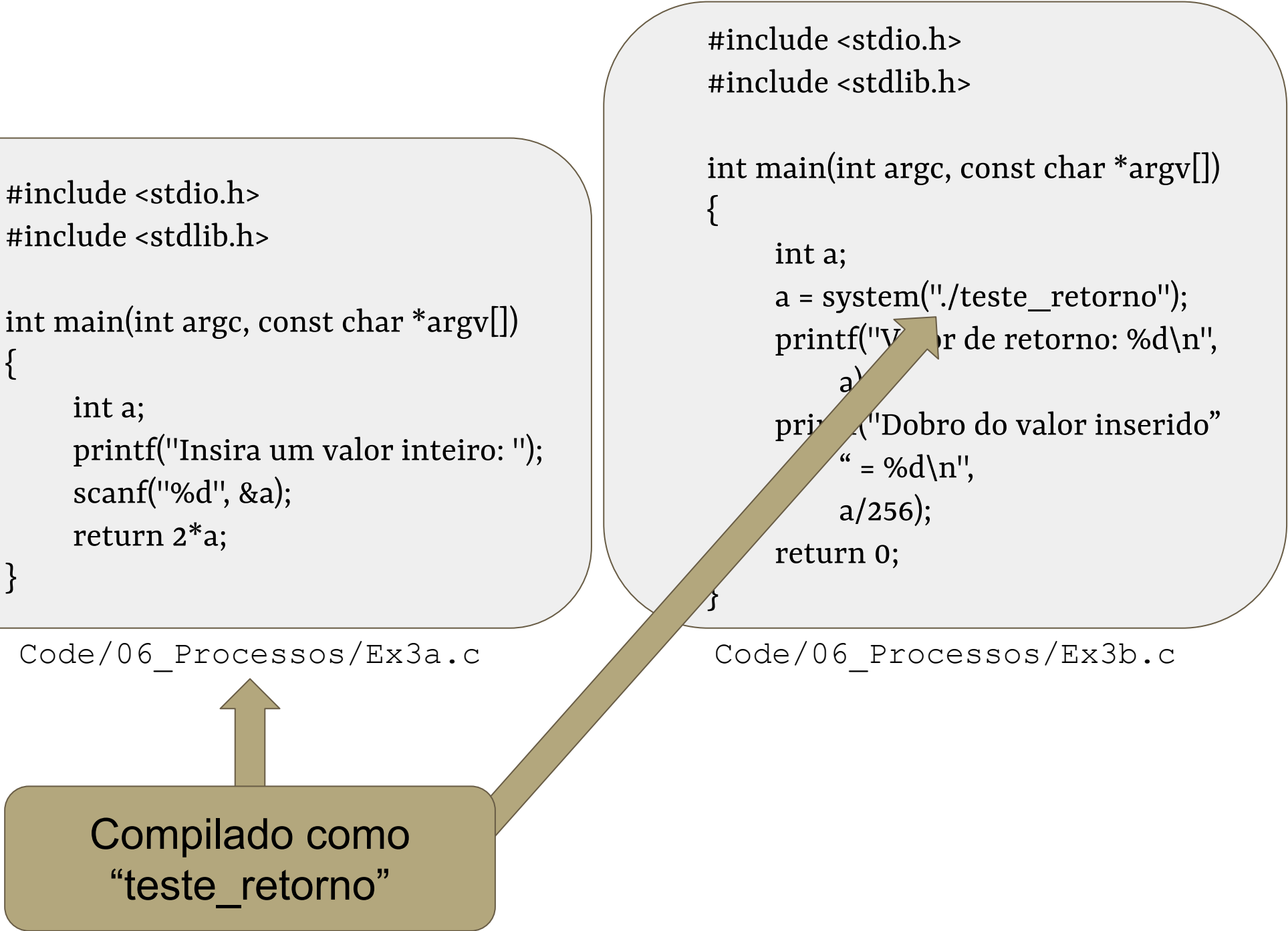
Code/06_Processos/Ex3a.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n",
           a);
    printf("Dobro do valor inserido"
           " = %d\n",
           a/256);
    return 0;
}
```

Code/06_Processos/Ex3b.c

Compilado como
"teste_retorno"



```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Insira um valor inteiro: ");
    scanf("%d", &a);
    return 2*a;
}
```

Code/06_Processos/Ex3a.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n",
        a);
    printf("Dobro do valor inserido"
        " = %d\n",
        a/256);
    return 0;
}
```

Code/06_Processos/Ex3b.c

Valor retornado



```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Informe um valor: ");
    scanf("%d", &a);
    return 2*a;
}
```

Code/06_Processos/Ex3a.c

A função "system()" insere 8 bits, por onde ela indica possíveis erros

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n", a);
    printf("Dobro do valor inserido"
           " = %d\n", a/256);
    return 0;
}
```

Code/06_Processos/Ex3b.c


```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Informe um valor: ");
    scanf("%d", &a);
    return 2*a;
}
```

A função "system()" insere 8 bits, por onde ela indica possíveis erros

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n", a);
    printf("Dobro do valor inserido"
           " = %d\n", a/256);
    return 0;
}
```

Code/06_Proc

essos/Ex3b.c

A função "system()" é simples, porém perigosa, pois só indicamos um comando através de uma string

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
    int a;
    printf("Informe um valor: ");
    scanf("%d", &a);
    return 2*a;
}
```

A função "system()" insere 8 bits, por onde ela indica possíveis erros

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n", a);
    printf("Dobro do valor inserido"
           " = %d\n", a/256);
    return 0;
}
```

Code/06_Proc

essos/Ex3b.c

1. O comando na string é executado com os mesmos privilégios do processo atual

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    printf("Informe um valor: ");
    scanf("%d", &a);
    return 2*a;
}
```

A função "system()" insere 8 bits, por onde ela indica possíveis erros

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, const char *argv[])
{
    int a;
    a = system("./teste_retorno");
    printf("Valor de retorno: %d\n", a);
    printf("Dobro do valor inserido"
           " = %d\n", a/256);
    return 0;
}
```

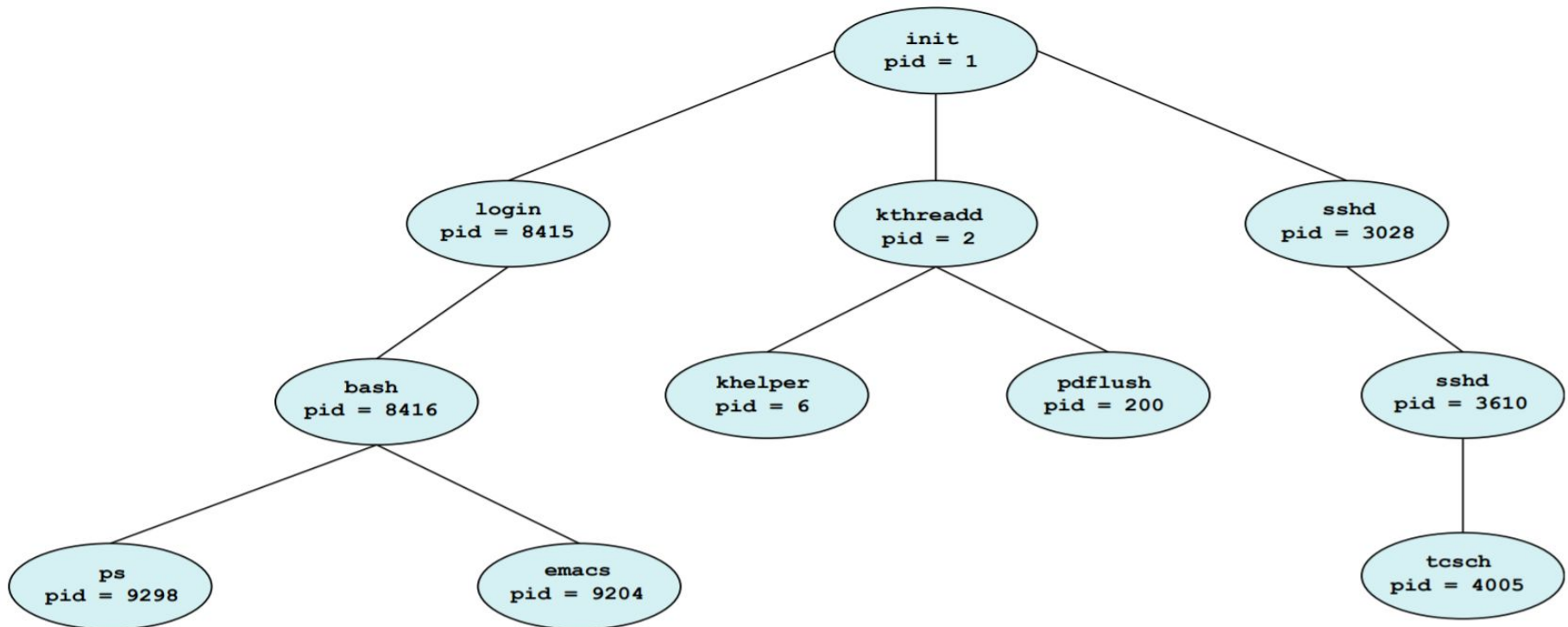
Code/06_Proc

essos/Ex3b.c

2. A função "system()" não confere o comando na string

Criação de Processos

- “fork()” e “exec()”
 - No UNIX, um processo não executa outro processo
 - Precisamos **copiar o processo atual** (processo-filho) e executar outro processo



```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void func(int x, int y, int z)
{
    printf("PID = %d, "
           "x=%d, y=%d e z=%d\n",
           getpid( ), x, y, z);
}

```

Code/06_Processos/Ex4.c

```

int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}

```

(Continuação)

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

```
void func(int x, int y, int z)
{
    printf("PID = %d, "
           "x=%d, y=%d e z=%d\n",
           getpid( ), x, y, z);
}
```

Code/06_Processos/Ex4.c

```
int main(void)
{
    int a = 1, b = 2, c = 3;
```

A função “func()”
escreve na tela o PID
do processo e o valor
de 3 variáveis

```
    c=9;
    func(a, b, c);
}
else
{
    sleep(1);
    func(a, b, c);
}
return 0;
}
```

(Continuação)

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

```
void func(int x, int y, int z)
{
    printf("PID = %d,
           "x=%d, y=%d\n",
           getpid(), x, y);
}
```

Code/06_Processos/Ex4.c

A função "func()" é
chamada com valores

x = a = 1
y = b = 2
z = c = 3

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

(Continuação)


```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
void func(void)
```

```
{
```

```
    printf("Hello World!\n");
```

```
}
```

Code/01

A função “fork()” cria uma cópia idêntica do processo atual, replicando o **program counter**, o **stack pointer**, o uso dos registradores, os descritores de arquivos abertos, a memória alocada etc.

```
int main(void)
```

```
{
```

```
    int a = 1, b = 2, c = 3;
```

```
    pid_t pid_filho = 0;
```

```
    func(a, b, c);
```

```
    pid_filho = fork();
```

```
    if(pid_filho == 0)
```

```
{
```

```
        a=7;
```

```
        b=8;
```

```
        c=9;
```

```
        func(a, b, c);
```

```
}
```

```
    else
```

```
{
```

```
        sleep(1);
```

```
        func(a, b, c);
```

```
}
```

```
    return 0;
```

```
}
```

(Continuação)

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

```
void func(int a, int b, int c)
{
    printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```

**Essa cópia é
chamada de
processo-filho, e ela
recebe um PID
diferente do
processo-pai**



```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

Code/06_Processos/Ex4.c

(Continuação)

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
void func(int a, int b, int c)
```

```
{
```

```
    printf("a=%d, b=%d, c=%d\n", a, b, c);
```

```
}
```

Considerando que o
PID do processo atual
é 1000, e que o
processo-filho recebe
o PID 1001...

Code/06_Processos/Ex4.c

```
int main(void)
```

```
{
```

```
    int a = 1, b = 2, c = 3;
```

```
    pid_t pid_filho = 0;
```

```
    func(a, b, c);
```

```
    pid_filho = fork();
```

```
    if(pid_filho == 0)
```

```
    {
```

```
        a=7;
```

```
        b=8;
```

```
        c=9;
```

```
        func(a, b, c);
```

```
    }
```

```
    else
```

```
    {
```

```
        sleep(1);
```

```
        func(a, b, c);
```

```
    }
```

```
    return 0;
```

```
}
```

(Continuação)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/
#include <uni
```

Program counter

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

```
%d, "
= %d e z = %d\n",
x, y, z);
```

processos/Ex4.c

```
~/Code/06_Processos $ ./Ex4.out
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

(Continuação)

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

```
void func(int x, y, z)
{
```

Program counter

```
PID = 1000
a = 1
b = 2
c = 3
pid_filho = 1001
etc.
```

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
```

```
    printf("x=%d e z=%d\n",
           x, y, z);
```

```
Processos/Ex4.c
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

(Continuação)

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

```
void func(int
{
```

Program counter

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

PID = 1001

a = 1

b = 2

c = 3

pid_filho = 0

etc.

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

PID = 1001

a = 1

b = 2

c = 3

pid_filho = 0

etc.

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
```

```
int main(void)
```

```
{
```

```
    int a = 1, b = 2, c = 3;
```

```
    pid_t pid_filho = 0;
```

```
    func(a, b, c);
```

```
    pid_filho = fork();
```

```
    if(pid_filho == 0)
```

Agora temos 2 processos
ao invés de 1

Repare que
“pid_filho = 1001”
para o processo-pai e
“pid_filho = 0”
para o processo-filho

```
    return 0;
```

```
}
```

(Continuação)


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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

PID = 1001

a = 1

b = 2

c = 3

pid_filho = 0

etc.

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
```

```
int main(void)
```

```
{
```

```
    int a = 1, b = 2, c = 3;
```

```
    pid_t pid_filho = 0;
```

```
    func(a, b, c);
```

```
    pid_filho = fork();
```

```
    if(pid_filho == 0)
```

```
    {
```

```
        a=7;
```

```
        b=8;
```

```
        c=9;
```

```
        func(a, b, c);
```

```
    }
```

```
    else
```

```
    {
```

```
        sleep(1);
```

```
        func(a, b, c);
```

```
    }
```

```
    return 0;
```

```
}
```

PC(filho)

PC(pai)

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

PID = 1001

a = 7

b = 8

c = 9

pid_filho = 0

etc.

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

PC(filho)

PC(pai)

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

PID = 1001

a = 7

b = 8

c = 9

pid_filho = 0

etc.

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
```

```
int main(void)
```

```
{
```

```
    int a = 1, b = 2, c = 3;
```

```
    pid_t pid_filho = 0;
```

```
    func(a, b, c);
```

```
    pid_filho = fork();
```

```
    if(pid_filho == 0)
```

```
    {
```

```
        a=7;
```

```
        b=8;
```

```
        c=9;
```

```
        func(a, b, c);
```

```
    }
```

```
    else
```

```
    {
```

```
        sleep(1);
```

```
        func(a, b, c);
```

```
    }
```

```
    return 0;
```

```
}
```

PC(pai)

PC(filho)

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

```
    printf("x=%d y=%d e z=%d\n",
           x, y, z);
```

Processos/Ex4.c

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

PC(pai)

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

```
    printf("x=%d y=%d e z=%d\n",
           x, y, z);
```

Processos/Ex4.c

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

PC(pai)

(Continuação)

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

```
void func(int x, int y, int z)
{
```

PID = 1000

a = 1

b = 2

c = 3

pid_filho = 1001

etc.

```
    printf("PID = %d, "
           "x=%d e y=%d e z=%d\n",
           pid, x, y, z);
```

Processos/Ex4.c

```
~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
PID = 1000, x = 1, y = 2, z = 3
```

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
```

PC(pai)

(Continuação)

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void func(int x, int y, int z)
{
    printf("PID = %d, "
           "x=%d, y=%d e z=%d\n",
           getpid( ), x, y, z);
}

```

Code/06_Processos/Ex4.c

```

~/Code/06_Processos $ ./Ex4.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
PID = 1000, x = 1, y = 2, z = 3
~/Code/06_Processos $

```

```

int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}

```

(Continuação)

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

```
void func(int x, int y, int z)
{
    printf("PID = %d, x=%d, y=%d, z=%d\n",
           getpid(), x, y, z);
}
```

Code/06_Processos

```
~/Code/06_Processos $ ./a.out
PID = 1000, x = 1, y = 2, z = 3
PID = 1001, x = 7, y = 8, z = 9
PID = 1000, x = 1, y = 2, z = 3
~/Code/06_Processos $
```

A função "sleep(n)"
suspende a
execução do
código durante "n"
segundos

Ela foi usada para
evitar que
processo-pai e
processo-filho
escrevessem no
terminal
simultaneamente

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    pid_t pid_filho = 0;
    func(a, b, c);
    pid_filho = fork();
    if(pid_filho == 0)
    {
        a=7;
        b=8;
        c=9;
        func(a, b, c);
    }
    else
    {
        sleep(1);
        func(a, b, c);
    }
    return 0;
}
```

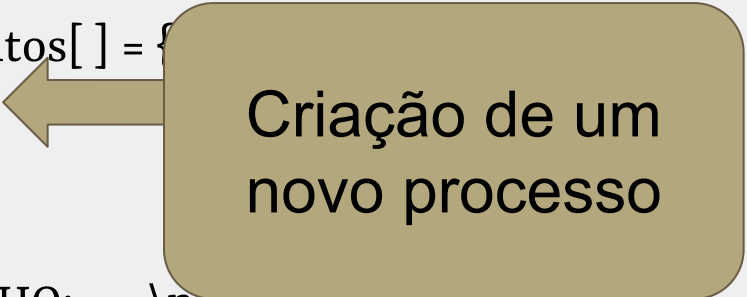
(Continuação)


```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = {
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n" );
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
            " erro de execucao em execvp()\n");
    }
    return 0;
}
```



Criação de um novo processo

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: Mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```

**Processo-filho
somente escreve
na tela e termina
a execução**

Processo-
pai
executa o
comando
“ls” usando
a função
“execvp()”

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char* lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <unistd.h>
int execvp(const char *file, char *const argv[]);
```

“execvp()” substitui o processo sendo executado atualmente por um novo

- “file” é o nome do arquivo (ou programa) a ser executado
- “argv” é um vetor de strings com os parâmetros para o arquivo (ou programa)
- Se o arquivo (ou comando) executar corretamente, “execvp()” não retorna, e o processo é terminado
- Se houver algum erro, “execvp()” o processo original é executado depois da chamada a “execvp()”, que retorna o valor -1

```
return 0;
```

```
}
```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * l
    pid_t p
    if(pid_
    {
        pri
    }
    else
    {
        pri
        exe
        printf( "Processo PAI: mensagem escrita se houver"
                " erro de execucao em execvp()\n");
    }
    return 0;
}

```

“execvp()” é uma das várias funções da família “exec()”:

- “execl()”
- “execlp()”
- “execle()”
- “execv()”
- “execvp()”
- “execvpe()”
- etc.


```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char* lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");

        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}
```

A lista de
argumentos é
um vetor de
strings

NULL indica o
fim do vetor

Se tudo der certo, o comando "ls" é executado corretamente, e esta mensagem não será executada



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

(void)

* lista_de_argumentos[ ] = { "ls", NULL};

pid_filho = fork( );

_filho == 0)

printf("Processo-FILHO: ---\n");

printf("Processo-PAI: ---\n");
execvp(lista_de_argumentos[0], lista_de_argumentos);
printf("Processo-PAI: mensagem escrita se houver"
      " erro de execucao em execvp()\n");
}
return 0;
}
```



```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", "-l", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", "-l", NULL};
    pid_t pid_filho = fork( );
    if ( pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");

        sleep(1);

        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```

Idêntico ao exemplo anterior, exceto pela lista de argumentos, que executa “ls -l” ao invés de “ls”

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "abcde", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "abcde", NULL};
    pid_t pid_filho = fork( );
    if ( pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");

        printf("Processo-PAI: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    return 0;
}

```


Idêntico ao exemplo anterior, exceto pela lista de argumentos, que executa “abcde” ao invés de “ls -l”

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "abcde", NULL};
    pid_t filho = fork( );
    if (filho == 0)
        printf("Processo-FILHO: ---\n");
    else
        printf("Processo-PAI: ---\n");
    execvp(lista_de_argumentos[0], lista_de_argumentos);
    printf("Processo-PAI: mensagem escrita se houver"
           " erro de execucao em execvp()\n");
}
return 0;
}

```

Como o comando
“abcde” não existe,
haverá um erro em
“execvp()”



Com o erro, o código continuará sendo executado após a chamada a “execvp()”, e esta mensagem será executada

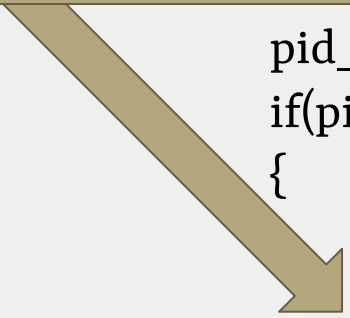
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *lista_de_argumentos[ ] = { "abcde", NULL};
    pid_t pid_filho = fork( );
    if (pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
    }
    printf("Processo-PAI: ---\n");
    execvp(lista_de_argumentos[0], lista_de_argumentos);
    printf("Processo-PAI: mensagem escrita se houver"
           " erro de execucao em execvp()\n");
}
return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0)
    {
        printf("Processo-FILHO: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    else
    {
        printf("Processo-PAI: ---\n");
    }
    return 0;
}
```

```
#include <stdio.h>
```

Neste exemplo, quem
chama a função “execvp()”
é o processo-filho, ao invés
do processo-pai



```
    lista_de_argumentos[ ] = { "ls", NULL};  
    pid_t pid_filho = fork( );  
    if(pid_filho == 0)  
    {  
        printf("Processo-FILHO: ---\n");  
        execvp(lista_de_argumentos[0], lista_de_argumentos);  
        printf("Processo-PAI: mensagem escrita se houver"  
              " erro de execucao em execvp()\n");  
    }  
    else  
    {  
        printf("Processo-PAI: ---\n");  
    }  
    return 0;  
}
```



```
#include <stdio.h>
```

O processo-pai tem muito menos instruções para executar do que o processo-filho

```
    lista_de_argumentos[ ] = { "ls", NULL};  
    pid_t pid_filho = fork( );  
    if(pid_filho == 0)  
    {  
        printf("Processo-FILHO: ---\n");  
        execvp(lista_de_argumentos[0], lista_de_argumentos);  
        printf("Processo-PAI: mensagem escrita se houver"  
              " erro de execucao em execvp()\n");  
    }  
    else  
    {  
        printf("Processo-PAI: ---\n");  
    }  
    return 0;  
}
```

Quando o processo-pai acaba, o processo-filho vira um processo-zumbi, executando “sem supervisão paterna”

```
lista_de_argumentos[ ] = { "ls", NULL};  
pid_t pid_filho = fork( );  
if(pid_filho == 0)  
{  
    printf("Processo-FILHO: ---\n");  
    execvp(lista_de_argumentos[0], lista_de_argumentos);  
    printf("Processo-PAI: mensagem escrita se houver"  
        " erro de execucao em execvp()\n");  
}  
else  
{  
    printf("Processo-PAI: ---\n");  
}  
return 0;  
}
```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
    char * lista_de_argumentos[ ] = { "ls", NULL};
    pid_t pid_filho = fork( );
    if(pid_filho == 0) {
        printf("Processo-FILHO: ---\n");
        execvp(lista_de_argumentos[0], lista_de_argumentos);
        printf("Processo-PAI: mensagem escrita se houver"
               " erro de execucao em execvp()\n");
    }
    else {
        printf("Processo-PAI: ---\n");
        wait(NULL);
        printf("Fim do processo-PAI\n");
    }
    return 0;
}

```

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

Idêntico ao exemplo anterior, mas agora o processo-pai espera o processo-filho terminar de ser executado, evitando a criação de um processo zumbi

```
    argumentos[ ] = { "ls", NULL};
    fork( );

    //o-FILHO: ---\n");
    //e_argumentos[0], lista_de_argumentos);
    printf("Processo-PAI: mensagem escrita se houver"
           " erro de execucao em execvp()\n");
    }
    else {
        printf("Processo-PAI: ---\n");
        wait(NULL);
        printf("Fim do processo-PAI\n");
    }
    return 0;
}
```