

# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

*Multiprogramação*  
Protecção dos recursos

# Processo: Relembrando conceitos...

- Processo: programa em execução. Se esta progride de forma sequencial, podemos dizer que é um processo sequential; mas se há múltiplas actividades a acontecer “ao mesmo tempo” no interior do processo, mencionamos o termo concorrência – por ex., *threads* (a ver mais tarde ☺)
- Um processo inclui: **Zona de código**, **Zona de Stack**, **Zona de dados**, etc.
- Um processo não pode: violar o EE de outros processos ou do SO, manipular periféricos críticos (como o disco) etc., i.e.,
  - **Não pode manipular recursos partilhados com outros!**

# *Instruções privilegiadas...*

## □ Como impedir estas operações?

- **Estudou em AC!** Um CPU tem
  - um registo de **flags**, e...
  - uma dessas flags, **chamemos-lhe S/U**, indica se o CPU está
    - em modo **Supervisor** ou **Utilizador**
- **Modo Supervisor**
  - O CPU pode executar qualquer instrução do seu repertório (instruction set)
- **Modo Utilizador**
  - Há instruções, ditas privilegiadas, que não podem ser executadas – se o forem, o CPU interrompe a execução e gera um *trap* (software interrupt)

# *Instruções privilegiadas...*

## □ Quais e porquê?

- **Estudou em AC!** ☺

- CLI: **CLear Interrupts**. Porquê? Porque um programa que executasse essa instrução fazia com que o CPU deixasse de atender interrupções (a “inversa”, STI, é também privilegiada):
  - O timer que activa o escalonador deixava de conseguir interromper, logo o escalonador não era activado, logo o processo podia nunca “largar” o CPU
  - Os interrupts dos periféricos deixavam de ser atendidos, logo o SO não sabia e – por exemplo na placa de rede – perdiam-se os dados que chegavam...
- HLT: **Halt**. Porquê? Óbvio ☺ parava o computador...
- IN e OUT: Porquê? Óbvio ☺ acedem directamente aos periféricos...

# *Instruções privilegiadas...*

- Quais e porquê? (continuação)
  - Não estudou em AC ☺
    - Instruções que alterem o mapa de memória permitindo ao processo aceder a “zonas” de memória que não são suas...
      - Um simples MOV para o registo de controle CR3
    - Alterar a flag S/U: permitiria a um processo de utilizador “fazer-se passar” por SO e “espiolhar tudo”, ou “dar cabo de tudo” ☺
    - etc... mais algumas, não muitas...
- Notas finais:
  - a implementação de S/U em CPUs modernos já não é uma “simples” flag...
  - Não confundir S/U com root! Não tem nada a ver... ☺

# *Como invocar serviços do SO?*

## □ Ponto de situação:

- *Um processo não pode aceder fora do seu EE*
- *Então, como pode executar uma função como `fork()` ?*
  - O código de `fork()` não está no processo<sup>1</sup>, está no SO...

## □ Em que ficamos???

- É “simples”: a chamada de uma função do SO faz-se usando no código do programa (que corre num processo em modo utilizador) a instrução `int` (software interrupt, a.k.a. trap) para “largar” o EE do processo e “entrar” no (EE) do kernel

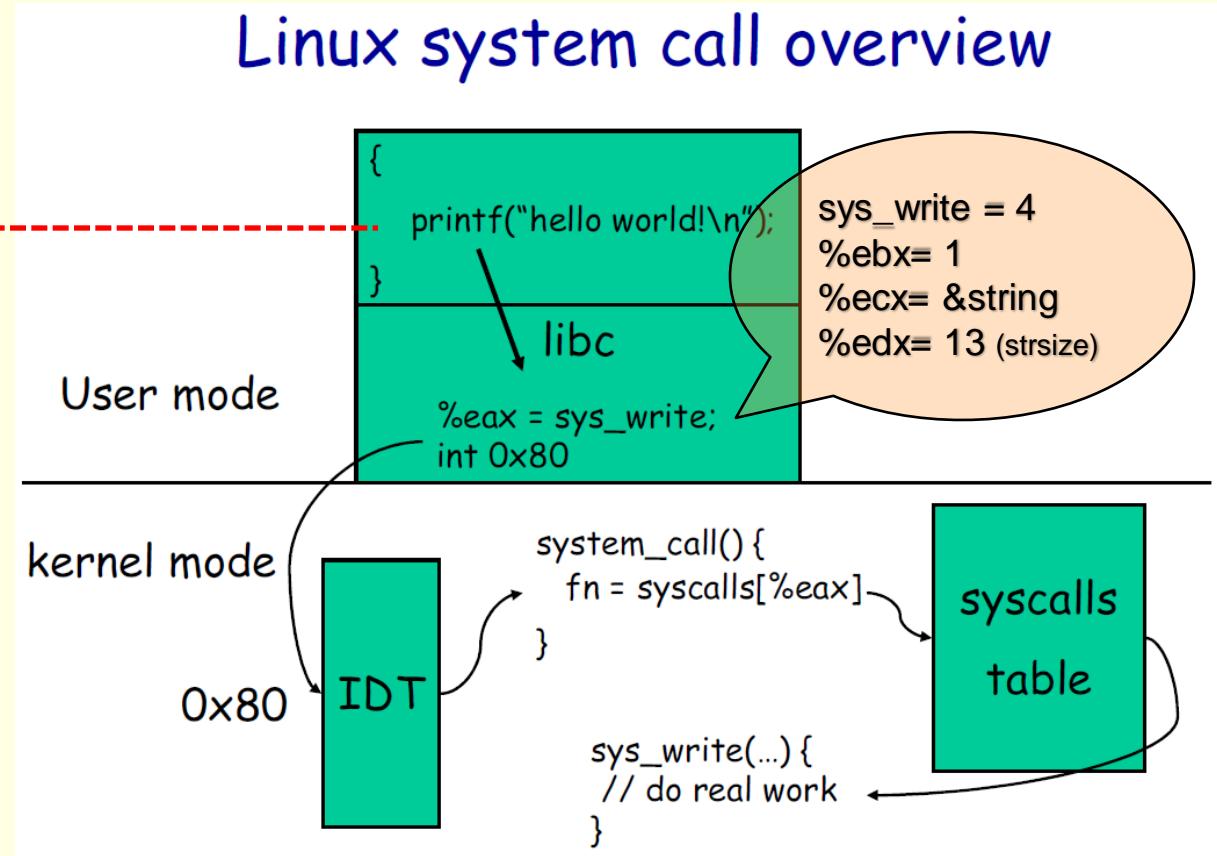
<sup>1</sup> Na verdade há um pouco do código do `fork` que está na biblioteca do C, e esta é “metida” no EE do processo, mas o “grosso” do código está no SO

# *Chamada de serviço SO*

- Quando o CPU executa a instrução **INT** (AC de novo ☺  
É mesmo preciso escrever isto? ☺)
  - Os registos “relevantes” (TPC) são guardados no stack
  - As flags são guardadas no stack (S/U está em U)
  - O modo do CPU é mudado para supervisor (flag em S)
  - ... o **INT** faz como que um **call** a uma função do kernel
- Para regressar ao processo...
  - ... a função do kernel termina com **IRET**
  - Pop das flags (**POPF**) e dos registos...
  - ... pop do IP e... estamos de novo no processo de utilizador!

# *Chamada de serviço Linux (formato “antigo”)*

*push &string  
call printf*



Adaptado de: CIS  
3207 - Operating  
Systems, Temple  
University

# *Chamada de serviço Linux*

## □ `syscall_table`

- Cada entrada contém um apontador (32 bits, i.e., 4 bytes, no Linux x86) para a função que realiza o serviço (existe em memória, no EE do kernel)

- $\text{write} = 4$ , logo  
 $\text{offset} = 16$

Offset	Symbol	sys_call_table	System call location
0	<code>__NR_restart_syscall</code>	<code>.long sys_restart_syscall</code>	<code>--&gt; ./linux/kernel/signal.c</code>
4	<code>__NR_exit</code>	<code>.long sys_exit</code>	<code>--&gt; ./linux/kernel/exit.c</code>
8	<code>__NR_fork</code>	<code>.long sys_fork</code>	<code>--&gt; ./linux/arch/386/kernel/process.c</code>
1272	<code>__NR_getcpu</code>	<code>.long sys_getcpu</code>	<code>--&gt; ./linux/kernel/sys.c</code>
1276	<code>__NR_epoll_pwait</code>	<code>.long sys_epoll_pwait</code>	<code>--&gt; ./linux/kernel/sys_ni.c</code>
<hr/>			
<code>__NR_syscalls</code>			
↑			
<code>./linux/include/asm/unistd.h</code>			
↑			
<code>./linux/arch/386/kernel/syscall_table.S</code>			

Adaptado (tinha um erro) de:  
[developer.ibm.com/tutorials/l-system-calls](http://developer.ibm.com/tutorials/l-system-calls)

# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/V/VS Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

*Gestão de Processos:*  
Brevíssima introdução à API do Unix

# *API da Gestão de Processos*

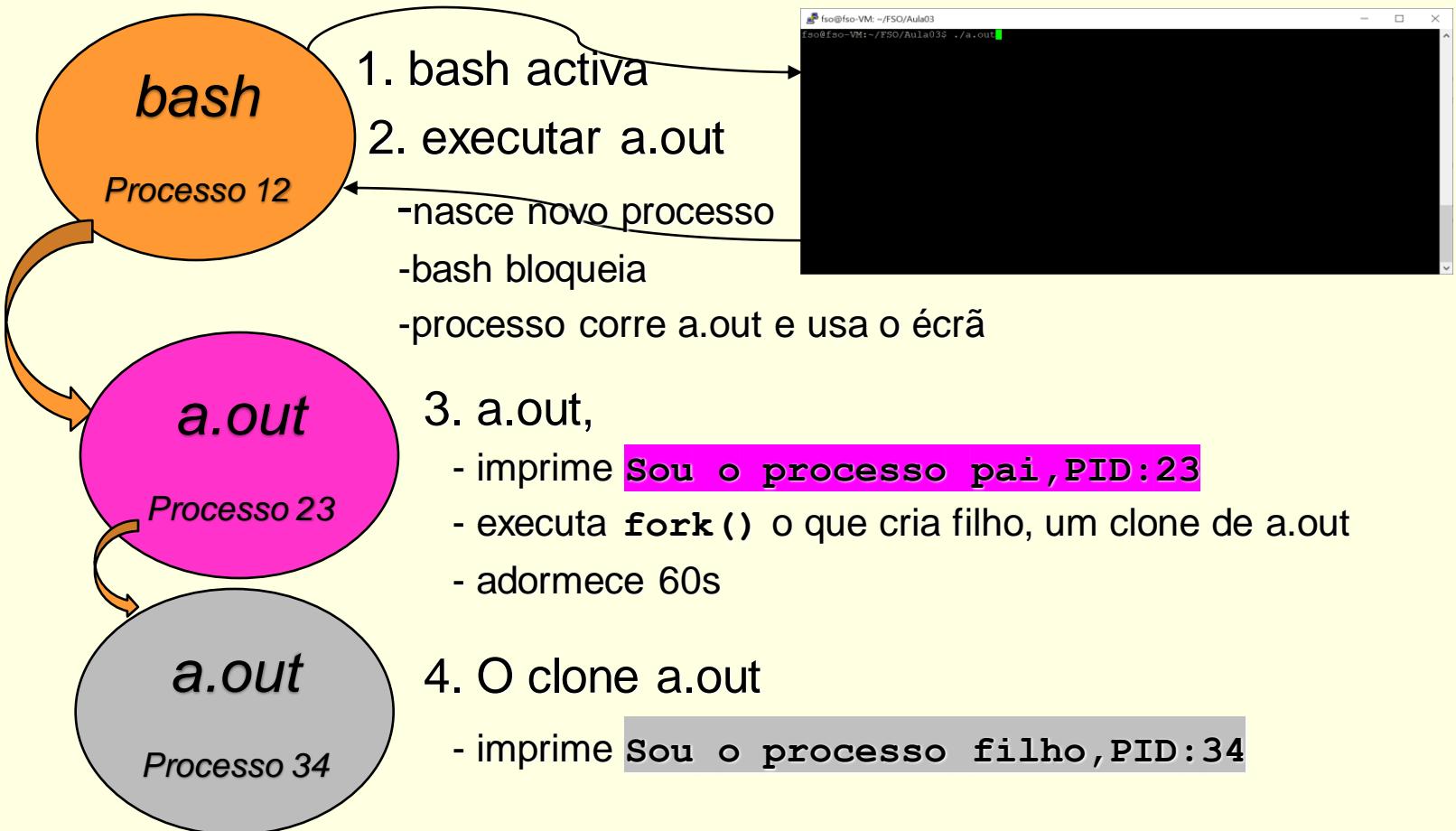
## □ Demo fork() Linux (“intro” ao Lab-03 ☺)

```
... includes ...

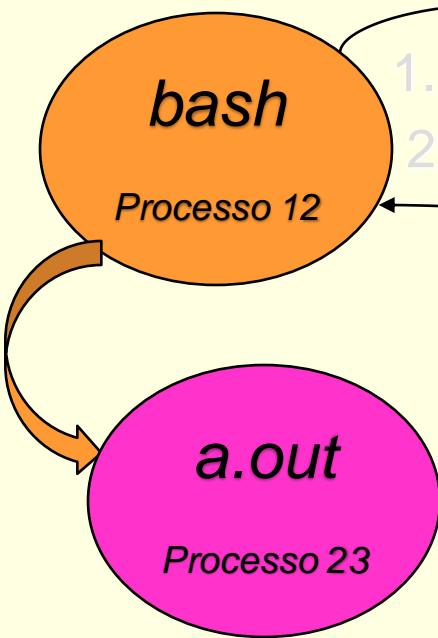
int main(int argc, char *argv[]) {

    printf("Sou o processo pai, PID:%d\n", getpid());
    if (fork()) {
        printf("Vou dormir 60s para dar tempo para que se vejam coisas interessantes\n");
        sleep(60);
        printf("Acordei! O filho ja deve ter terminado... Vou-me tambem embora\n");
    } else {
        printf("Sou o processo filho, PID:%d\n", getpid());
        printf(" Vou dormir 30s para dar tempo para que se vejam coisas interessantes\n");
        sleep(30);
    }
    return 0;
}
```

# *API dos processos: fork()*



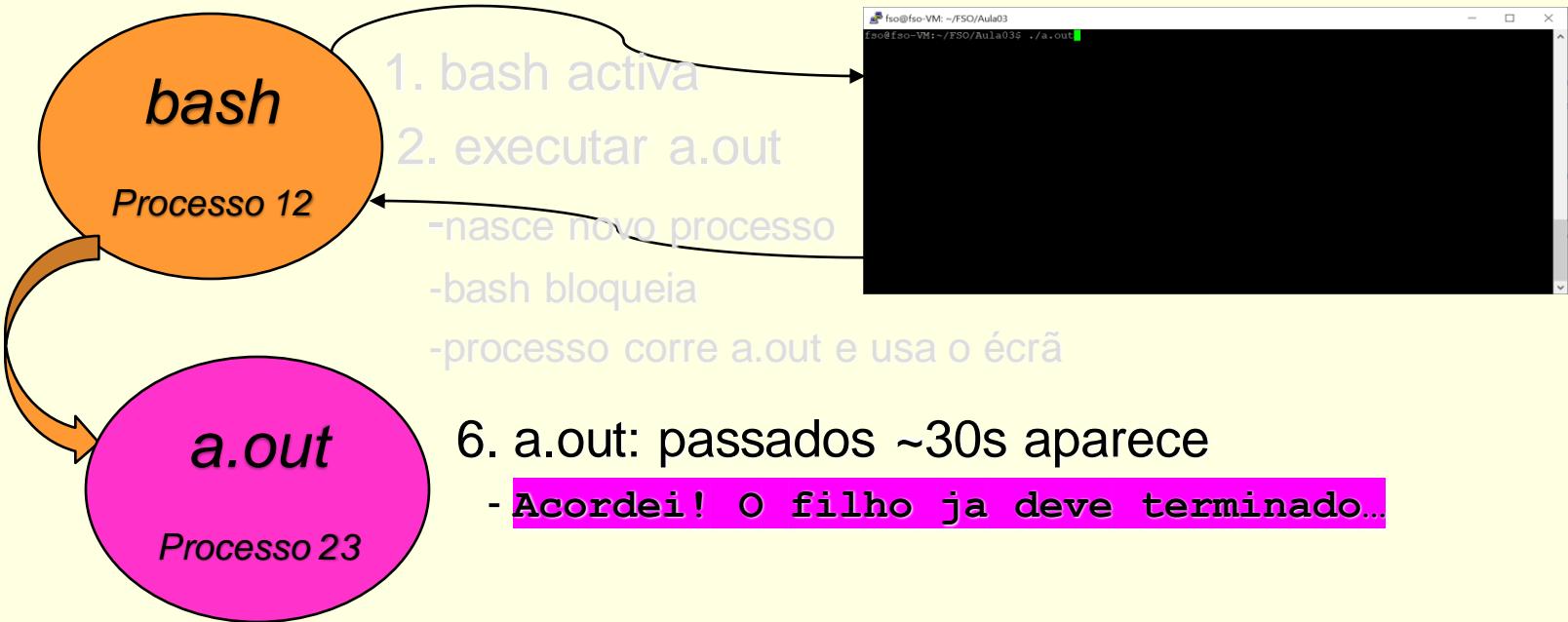
# *API dos processos: fork()*



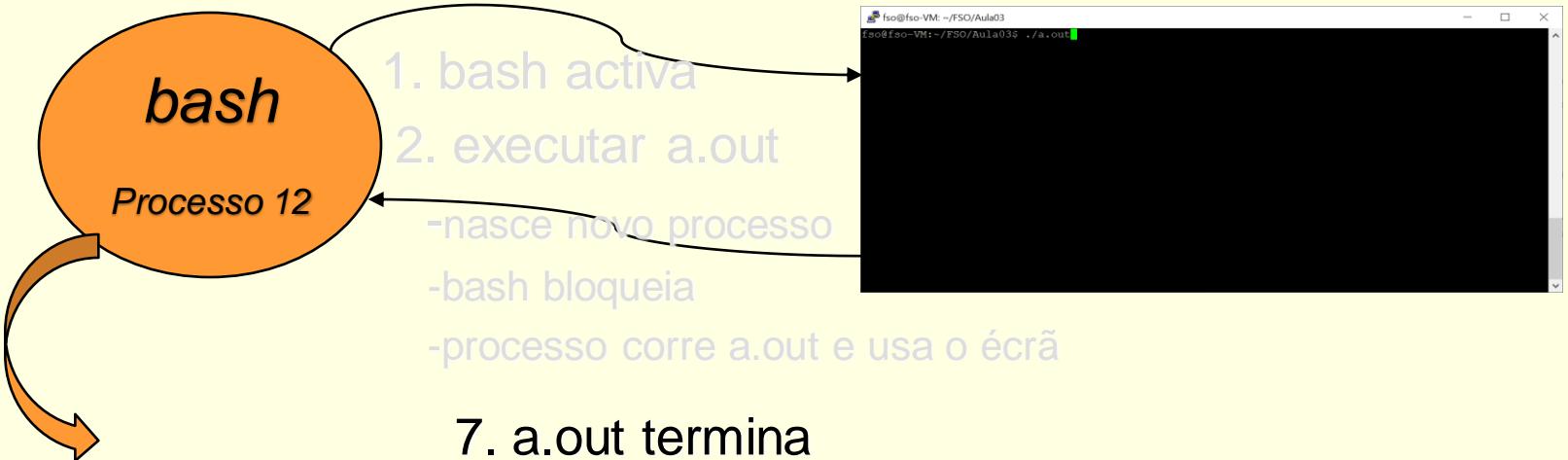
1. bash activa
2. executar a.out
  - nasce novo processo
  - bash bloqueia
  - processo corre a.out e usa o ecrã
3. a.out,
  - imprime **Sou o processo pai, PID: 23**
  - executa `fork()` o que cria filho, clone de a.out
  - adormece 60s
5. O clone a.out termina

```
fso@fso-VM: ~/FSO/Aula03
fso@fso-VM:~/FSO/Aula03$ ./a.out
```

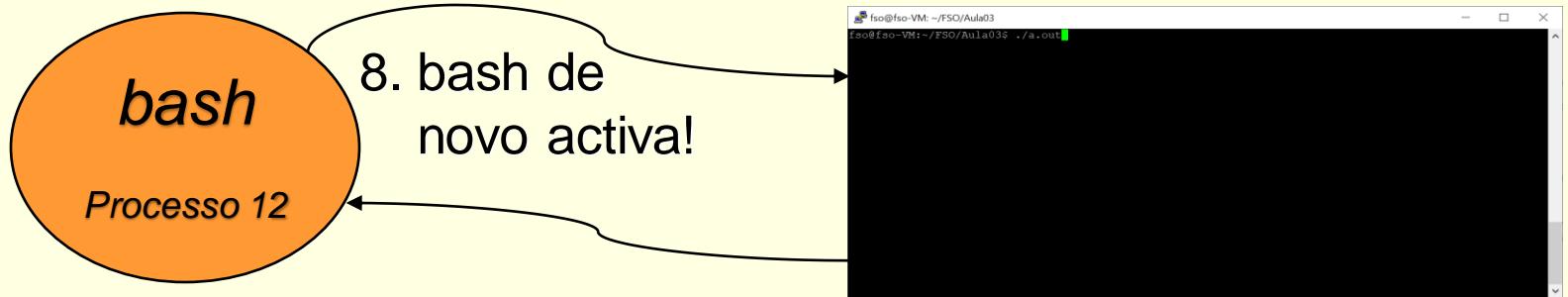
# *API dos processos: fork()*



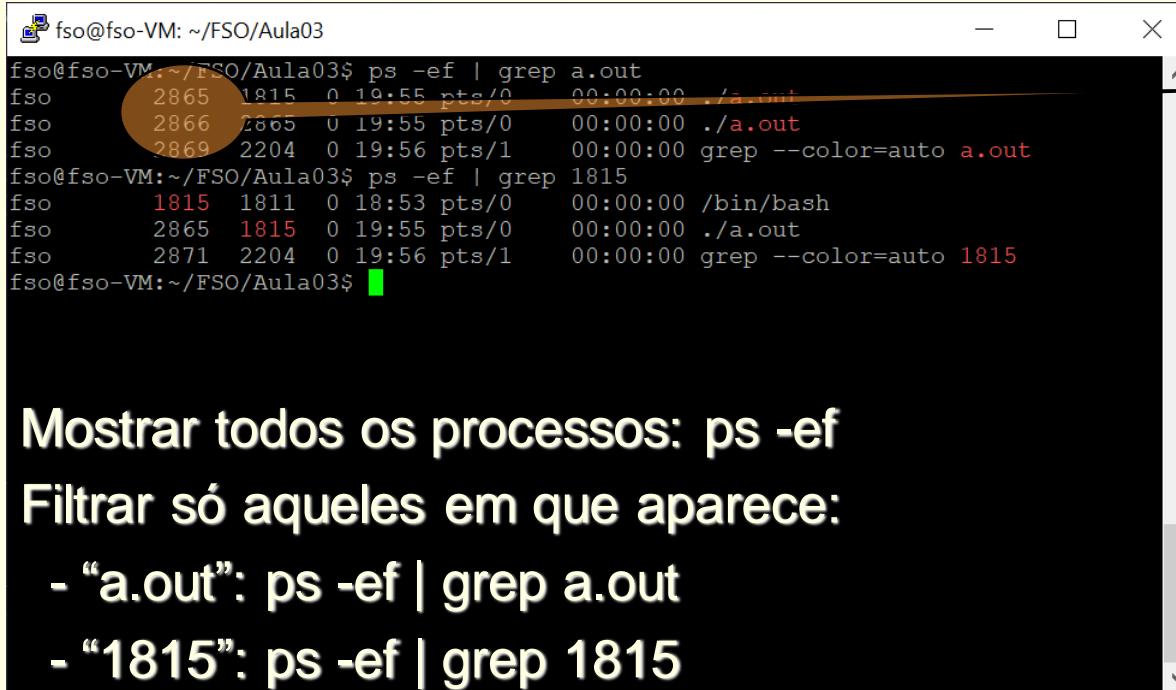
# *API dos processos: fork()*



# *API dos processos: fork()*



# *API dos processos: fork()*



A screenshot of a terminal window titled "fso@fso-VM: ~/FSO/Aula03". The window displays a list of processes from the command "ps -ef | grep a.out". The output shows several processes, with two highlighted in red: PID 2865 and PID 2866. Both are running the command "./a.out". Another process, PID 2869, is running "grep --color=auto a.out". Below this, another "grep" process is shown with PID 2871, also running "grep --color=auto 1815". The terminal prompt is "fso@fso-VM:~/FSO/Aula03\$". A large orange circle highlights the first two lines of the output.

```
fso@fso-VM:~/FSO/Aula03$ ps -ef | grep a.out
fso 2865 1815 0 19:55 pts/0 00:00:00 ./a.out
fso 2866 2865 0 19:55 pts/0 00:00:00 ./a.out
fso 2869 2204 0 19:56 pts/1 00:00:00 grep --color=auto a.out
fso@fso-VM:~/FSO/Aula03$ ps -ef | grep 1815
fso 1815 1811 0 18:53 pts/0 00:00:00 /bin/bash
fso 2865 1815 0 19:55 pts/0 00:00:00 ./a.out
fso 2871 2204 0 19:56 pts/1 00:00:00 grep --color=auto 1815
fso@fso-VM:~/FSO/Aula03$
```

Mostrar todos os processos: **ps -ef**

Filtrar só aqueles em que aparece:

- “a.out”: **ps -ef | grep a.out**
- “1815”: **ps -ef | grep 1815**

PIDs dos processos que nos interessam:  
2865 – a.out  
2866 – clone do a.out

O 2865 é filho do 1815

O 1815 é a bash do utilizador

# *API dos processos: fork()*

```
fso@fso-VM: ~
top - 23:31:39 up 39 min, 2 users, load average: 0,02, 0,01, 0,00
Tasks: 173 total, 1 running, 110 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,0 us, 0,3 sy, 0,0 ni, 99,7 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 1993620 total, 1557808 free, 146092 used, 289720 buff/cache
KiB Swap: 998396 total, 998396 free, 0 used. 1626184 avail Mem

PID USER      PR  NI    VIRT    RES   SHR S %CPU %MEM     TIME+ COMMAND
2163 fso      20   0   2204    520   476 S 0,0  0,0    0:00.00 teste
2164 fso      20   0   2204     56    0 S 0,0  0,0    0:00.00 teste
```

Processos que nos interessam capturados usando o utilitário **top**

Alterou-se o nome do executável de **a.out** para **teste** para ser mais fácil de capturar só o que interessa com o filtro do **top**

- Para filtrar por nome do executável:
  - clicar em “o”
  - escrever COMMAND=teste (e click em <enter>)

# *API dos processos: fork()*

## □ Uma variante...

```
int main(int argc, char *argv[]) {  
    if (fork()) {  
        printf("Sou o processo pai, PID:%d e vou-me ja embora\n", getpid());  
    } else {  
        printf("Sou o processo filho, PID:%d\n", getpid());  
        printf(" Vou dormir 30s para dar tempo para que se vejam coisas interessantes\n");  
        sleep(30);  
        printf("Sou o processo filho, PID:%d, ja acordei!\n", getpid());  
    }  
    return 0;  
}
```

## □ O output é:

```
fso@fso-VM:~/FSO/Aula03$ ./lab-02  
Sou o processo pai, PID:2594 e vou-me ja embora  
fso@fso-VM:~/FSO/Aula03$ Sou o processo filho, PID:2595  
Vou dormir 30s para dar tempo para que se vejam coisas interessantes  
Sou o processo filho, PID:2595, ja acordei!
```

O processo pai terminou muito rapidamente e o bash ficou activo de novo; o filho (órfão/zombie) ficou dependente do PID 1 do Linux (init) e continuou a correr...

O prompt da bash está meio escondido entre os prints do filho...

# *API dos processos: desafio...*

- E agora? Quantos processos são criados?

```
... includes

int main(int argc, char *argv[]) {
    fork();
    fork();

    return 0;
}
```

# *API dos processos: wait()*

## □ Esperar que o filho termine...

```
int main(int argc, char *argv[]) {
    if (fork()) {
        printf("Sou o processo pai, PID:%d e vou esperar que o filho termine\n", getpid());
        wait();
    } else {
        printf("Sou o processo filho, PID:%d\n", getpid());
        printf(" Vou dormir 30s para dar tempo para que se vejam coisas interessantes\n");
        sleep(30);
        printf("Sou o processo filho, PID:%d, ja acordei!\n", getpid());
    }
    return 0;
}
```

## □ Quando se aplica?

- Quando o pai tem de esperar que o filho termine, e não se pode prever quando é que isso acontece... (ver Lab-03 ☺)

# *API da Gestão de Processos*

## □ Demo fork() Linux

```
... includes ...

int main(int argc, char *argv[]) {

    printf("Sou o processo pai, PID:%d\n", getpid());
    if (fork()) {
        printf("Vou dormir 60s para dar tempo para que se vejam coisas interessantes\n");
        sleep(60);
        printf("Acordei! O filho ja deve ter terminado... Vou-me tambem embora\n");
    } else {
        printf("Sou o processo filho, PID:%d\n", getpid());
        printf(" Vou dormir 30s para dar tempo para que se vejam coisas interessantes\n");
        sleep(30);
    }
    return 0;
}
```

# *Chamadas efectuadas pelo processo*

- *Do ponto de vista do programador:*
  - “directas”: *fork()* e *sleep()*
  - *indirectas*: *write()*, chamado pelo *printf()*
- *Vamos ver?*

```
$ strace ./lab-01
execve("./lab-01", ["../lab-01"], /* 31 vars */) = 0
brk(NULL) = 0x804b000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fd5000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7e0c000
mprotect(0xb7fdb000, 8192, PROT_READ)      = 0
mprotect(0x8049000, 4096, PROT_READ)        = 0
mprotect(0xb7ffe000, 4096, PROT_READ)        = 0
munmap(0xb7fc3000, 71173)                  = 0
getpid()                                    = 2545
```

# *Chamadas efectuadas pelo processo* (continuação)

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
brk(NULL)                      = 0x804b000
brk(0x806c000)                 = 0x806c000

write(1, "Sou o processo pai, PID:2545\n") = 29

clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb7e0c768) = 2546

write(1, "Vou dormir 60s para dar tempo para que se vejam coisas interessantes) = 69

nanosleep({60, 0})

Sou o processo filho, PID:2546
Vou dormir 30s para dar tempo para que se vejam coisas interessantes
{29, 997020513})      = ? ERESTART_RESTARTBLOCK (Interrupted by signal)
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=2546, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---
restart_syscall(<... resuming interrupted nanosleep ...) = 0
write(1, "Acordei! O filho ja deve ter terminado... Vou-me tambem embora) = 63
exit_group(0)                  = ?
+++ exited with 0 +++
```