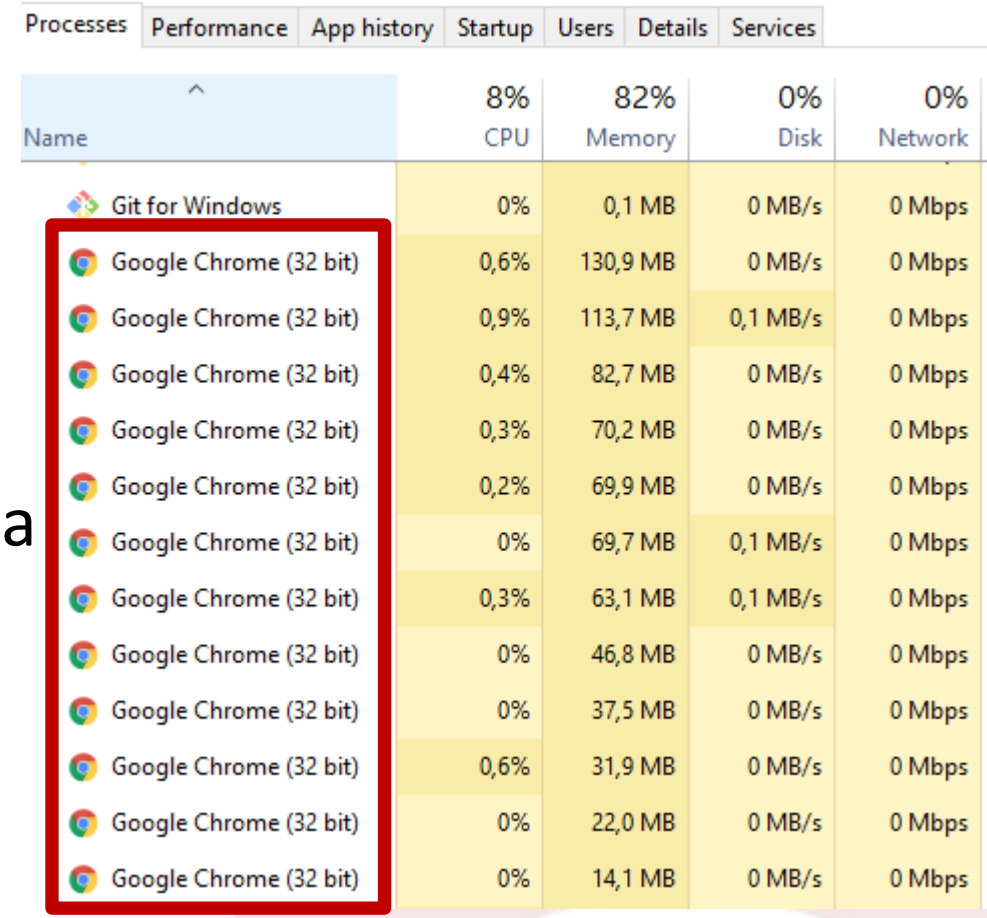
Programação Avançada
Capítulo 1
Processos

Patrício Domingues
ESTG/Politécnico de Leiria, 2025

Processos / lembrete

- ✓ Programa
  - Ficheiro executável (e.g., EXE no Windows)
- ✓ Processo
  - Instância de um programa em execução
  - Vários processos podem estar a executar o mesmo programa
    - Exemplo: **chrome.exe**
  - Cada processo…
    - **Program counter (PC)**
    - **Owner, PID**
    - **Atributos segurança**
    - **Espaço endereçamento**



Processos no Windows

- ✓ `Tasklist` comando (`tasklist /?` **ajuda**)
  - Lista as *tarefas* (similar ao **ps** do Unix)

C:\Windows\System32\cmd.exe

```
c:\>tasklist | more

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0          8 K
System                           4 Services                   0        416 K
Registry                        96 Services                   0     45 668 K
smss.exe                       384 Services                   0        184 K
csrss.exe                      508 Services                   0      1 464 K
wininit.exe                    600 Services                   0        264 K
services.exe                   684 Services                   0      4 724 K
lsass.exe                      724 Services                   0      9 532 K
svchost.exe                    888 Services                   0        272 K
fontdrvhost.exe                916 Services                   0        200 K
svchost.exe                    924 Services                   0     16 652 K
svchost.exe                    368 Services                   0     13 252 K
svchost.exe                    532 Services                   0      2 824 K
svchost.exe                   1092 Services                   0      1 988 K
svchost.exe                   1120 Services                   0      3 928 K
svchost.exe                   1164 Services                   0      1 472 K
svchost.exe                   1212 Services                   0        532 K
svchost.exe                   1276 Services                   0      2 300 K
svchost.exe                   1296 Services                   0      7 752 K
svchost.exe                   1440 Services                   0      4 768 K
svchost.exe                   1452 Services                   0        864 K
svchost.exe                   1476 Services                   0      5 136 K
svchost.exe                   1500 Services                   0      3 584 K
svchost.exe                   1624 Services                   0      3 088 K
svchost.exe                   1636 Services                   0      9 992 K
Memory Compression            1696 Services                   0     80 476 K
```

Processos in Linux

✓htop

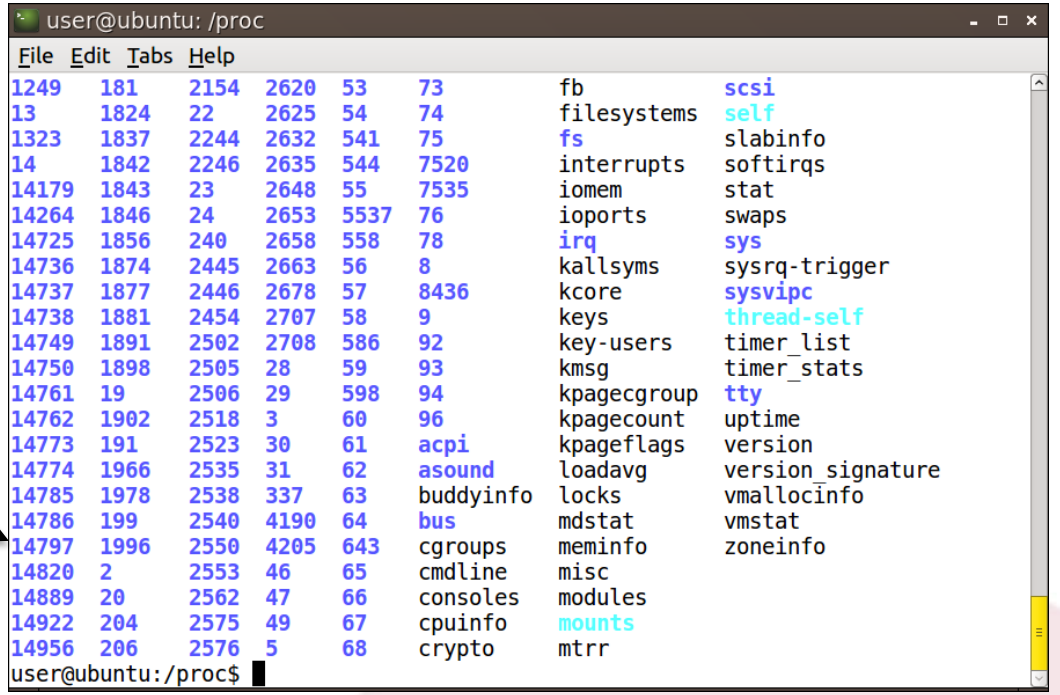
- Por omissão, o *htop* não está instalado no Ubuntu
- `sudo apt-get install htop`

```
  0[|                                        0.6%] Tasks: 75, 137 thr, 181 kthr; 1 running
  1[||                                       1.3%] Load average: 0.04 0.08 0.07
Mem[||||||||||||||||                   569M/3.78G] Uptime: 00:09:29
Swp[                                      0K/512M]

  Main   I/O
    PID USER       PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
   1890 osboxes     20   0 11096  4736  3584 R   4.0  0.1  0:00.51 htop
      1 root        20   0 22496 13592  9496 S   0.0  0.3  0:02.30 /sbin/init splash
    373 root        19  -1 66960 17488 16336 S   0.0  0.4  0:00.40 /usr/lib/systemd/systemd-journald
    420 root        20   0  148M  1420  1280 S   0.0  0.0  0:00.00 vmware-vmblock-fuse /run/vmblock-fuse -o rw,s
    433 root        20   0  148M  1420  1280 S   0.0  0.0  0:00.00 vmware-vmblock-fuse /run/vmblock-fuse -o rw,s
    434 root        20   0  148M  1420  1280 S   0.0  0.0  0:00.00 vmware-vmblock-fuse /run/vmblock-fuse -o rw,s
    454 root        20   0 31652  9600  4864 S   0.0  0.2  0:00.28 /usr/lib/systemd/systemd-udevd
    518 systemd-re  20   0 21576 12672 10496 S   0.0  0.3  0:00.11 /usr/lib/systemd/systemd-resolved
    526 systemd-ti  20   0 91044  7808  6912 S   0.0  0.2  0:00.07 /usr/lib/systemd/systemd-timesyncd
    594 systemd-ti  20   0 91044  7808  6912 S   0.0  0.2  0:00.00 /usr/lib/systemd/systemd-timesyncd
    665 root        20   0 56056 11904 10368 S   0.0  0.3  0:00.05 /usr/bin/VGAuthService
    669 root        20   0  238M  9088  7808 S   0.0  0.2  0:00.84 /usr/bin/vmtoolsd
    725 root        20   0  238M  9088  7808 S   0.0  0.2  0:00.00 /usr/bin/vmtoolsd
    727 root        20   0  238M  9088  7808 S   0.0  0.2  0:00.03 /usr/bin/vmtoolsd
    728 root        20   0  238M  9088  7808 S   0.0  0.2  0:00.00 /usr/bin/vmtoolsd
    966 root        20   0  305M  7584  6816 S   0.0  0.2  0:00.06 /usr/libexec/accounts-daemon
    967 root        20   0  8288  2432  2304 S   0.0  0.1  0:00.01 /usr/sbin/anacron -d -q -s
    969 avahi       20   0  8608  4352  3968 S   0.0  0.1  0:00.06 avahi-daemon: running [osboxes.local]
    971 root        20   0  9424  2688  2560 S   0.0  0.1  0:00.01 /usr/sbin/cron -f -P
    972 messagebus  20   0 10892  6144  4480 S   0.0  0.2  0:00.23 @dbus-daemon --system --address=systemd: --no
F1Help  F2Setup  F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

/proc no Linux

- Pseudo sistema de ficheiros **/proc** regista dados da atividade do sistema
- Para cada processo ativo, existe uma pasta no **/proc**
  - O nome da pasta é o PID do processo

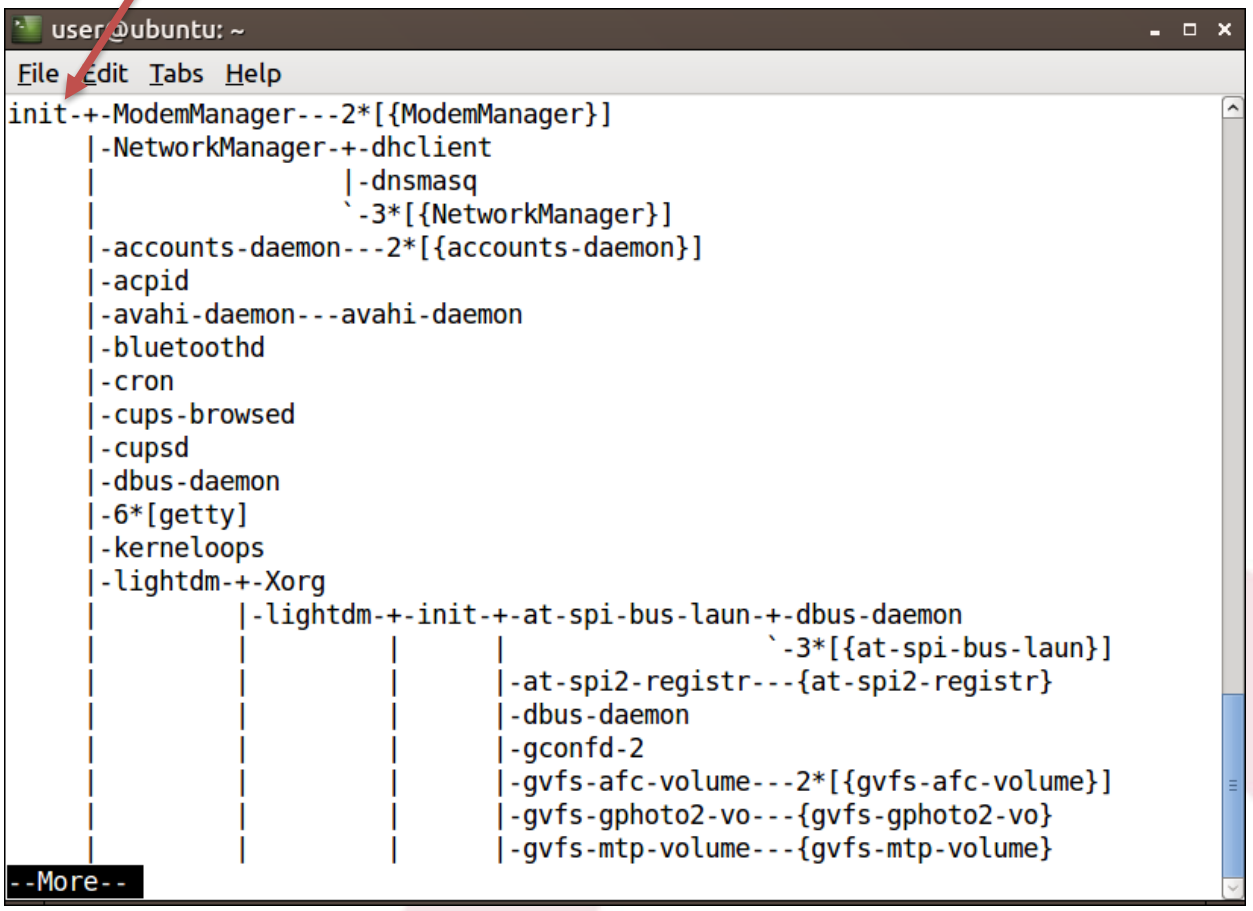


CRIAÇÃO E GESTÃO DE PROCESSOS EM LINUX

Processos no Unix (#1)

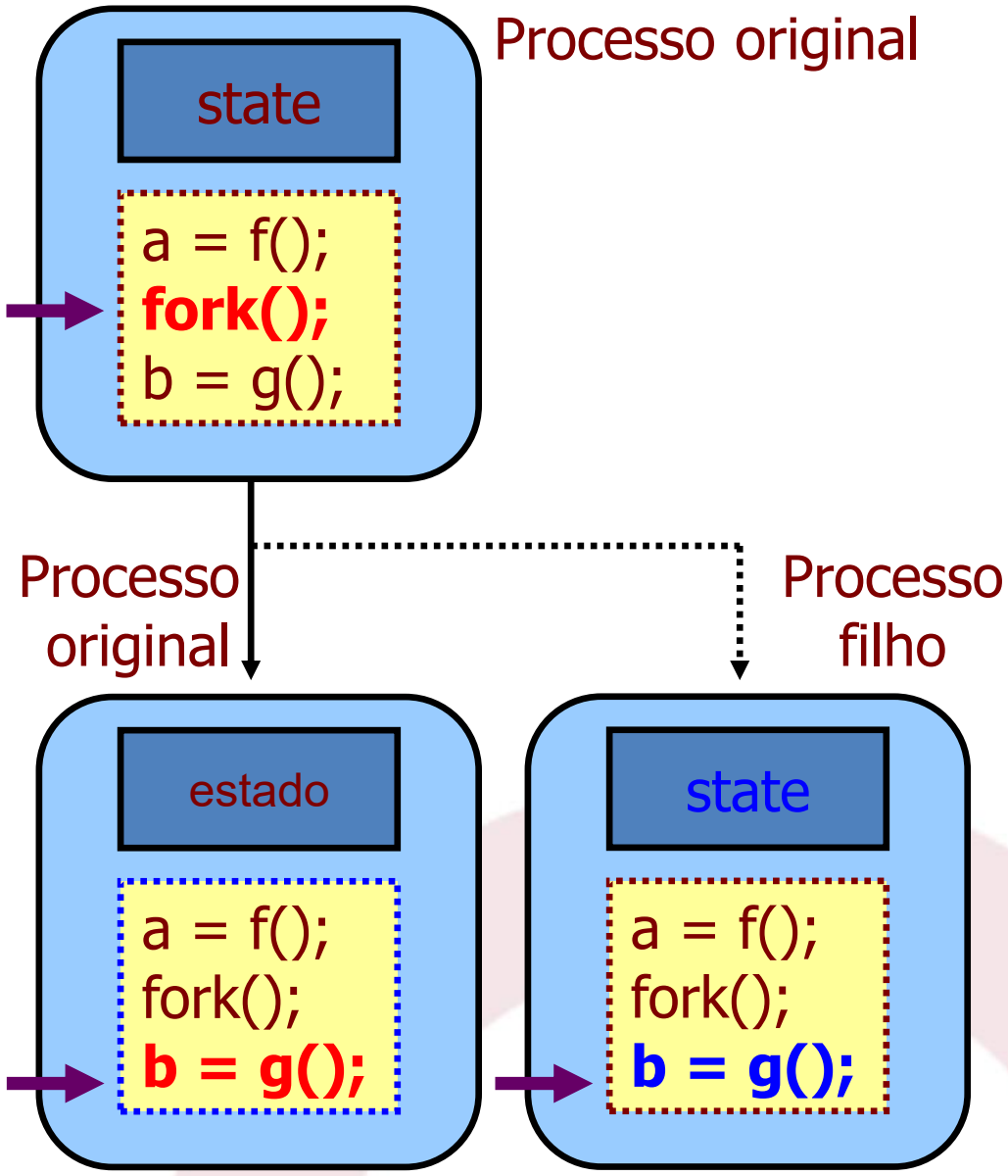
- ✓ Processos no Unix
  - Todos os ficheiros têm uma origem comum
    - Processo “init” (PID=1)
    - Um processo é criado por outro processo
    - Relacionamento pai/filho
    - Hierarquia de processos

Utilitário **pstree**



Modelo de processos em UNIX

- `fork()`
  - Chamada ao sistema para criar um processo
  - Empregue pelo "processo pai"
- O processo filho herda as características do processo pai
  - O processo filho é "fotocópia" do processo pai
  - Variáveis, Contador de programa, ficheiros abertos, memória alocada, etc.
- Após o "fork", cada processo corre separadamente e independentemente
  - A alteração de uma variável num processo não altera o valor no outro processo



Chamada ao sistema `fork`

- ✓ `pid_t fork(void);`
- ✓ A chamada ao sistema fork devolve um inteiro que difere consoante o processo:
  - `0` para o processo recém-criado
  - `> 0` para o processo "pai"
    - O valor corresponde ao PID do processo filho
- ✓ fork devolve `-1` quando ocorre um erro
  - `errno` contém código numérico de erro
  - `strerror(errno)` para obter a *string* de erro

Exemplo – fork

```
user@so-vm:~$ ./simple_fork
[7130] I'm the father!
[7131] I'm the son!
[7131] My parent is: 7130
[7130] father: wait is over.
```

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

int main()
{
  pid_t id;

  id = fork(); /* returns 0: son process; > 0 to the parent */
  if (id == 0)
    {  /* code only executed by the son process */
      printf("[%d] I'm the son!\n", getpid());
      printf("[%d] My parent is: %d\n", getpid(), getppid());
    }
  else if (id > 0 )
    {/* code only executed by the parent process */
      printf("[%d] I'm the father!\n", getpid());
      wait(NULL);
       printf("[%d] father: wait is over.\n", getpid());
    }
  return 0;
}
```

https://pastebin.com/EP36FQZz

Exemplo – ciclo for com `fork`

✓ Quantas linhas são mostradas na saída padrão?

```
#include <...>
int main(void){
        int i;
        for(i=0;i<3;i++){
                if( fork() == 0 ){
                        printf("PID=%u\n", getpid());
                        fflush(stdout);
                }
        }
        return 0;
}
```

Apenas processos "novos" chamam o "printf"
Resposta: 7
2^n-1 com n=3

Custos de um "fork"

- ✓ A chamada ao sistema cria um processo novo através da clonagem do processo corrente
  - Será que isso significa que toda a memória atribuída ao processo pai é copiada/duplicada?
    - Não, caso contrário a chamada fork seria computacionamente onerosa…
  - O Unix faz uso do mecanismo "**copy-on-write**" (COW)

Copy-on-write >

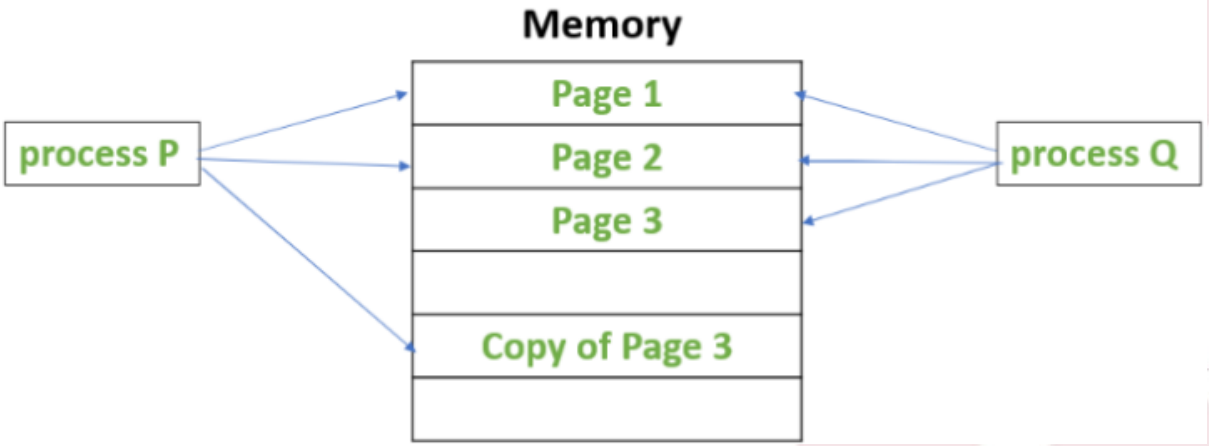
Copy-on-Write (#1)

- ✓ O SO organiza a memória em blocos de tamanho fixo *(“memória paginada”)*
  - *Uma página tem, normalmente, 4 KiB (e.g., X86_64)*
- ✓ Para evitar duplicação de conteúdo de memória, após um fork, os processos pai e filho partilham as mesmas páginas de memória
  - A tabela de páginas de cada processo aponta para as mesmas páginas
  - Não existe duplicação de conteúdo

Continua >>

Copy-on-Write (#2)

- ✓ As páginas partilhadas são marcadas como sendo apenas para leitura (**readonly**)
- ✓ Quando um processo tenta escrever numa página partilha (e.g., alterar o valor de uma varíavel: i++)
  - O hardware notifica o SO da situação e cria uma cópia da página
  - Após a cópia, cada processo tem a sua propria página com aquele conteúdo
    - Continuam a partilhar as outras páginas
- ✓ Apenas as páginas "escritas" são duplicadas
  - Mecanismos também conhecido como "*copy-on-demand*"



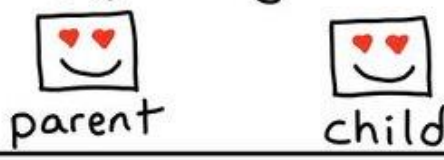
https://www.geeksforgeeks.org/copy-on-write/

copy on write

Julia Evans
@b0rk

On Linux, you start new processes using the fork() or clone() system call

calling fork gives you a child process that's a copy of you

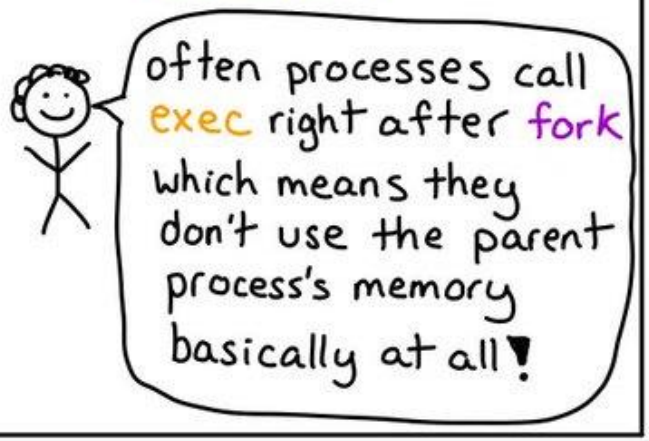


the cloned process has EXACTLY the same memory

- → same heap
- → same stack
- → same memory maps

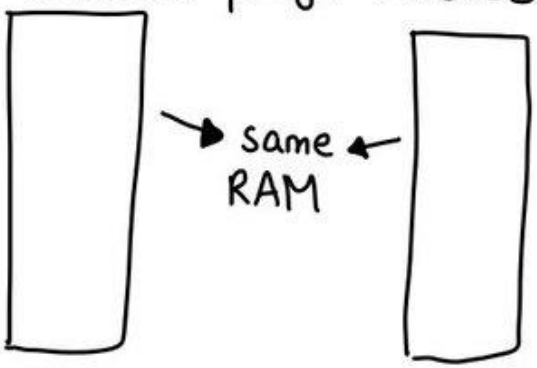
if the parent has 3GB of memory, the child will too

copying all that memory every time we fork would be slow and a waste of RAM



so Linux lets them share physical RAM and only copies the memory when one of them tries to write.

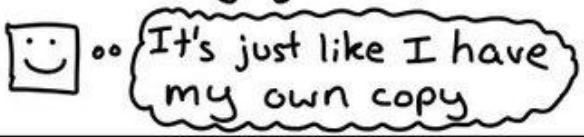
Linux does this by giving both the processes identical page tables



but marks every page as read only

when a process tries to write to a shared memory address

1. there's a page fault
2. Linux makes a copy of the page & updates the page table
3. the process continues, blissfully ignorant



Process ID (PID) – #1

- ✓ Process ID (PID)
  - Inteiro que identifica um processo
- ✓ O kernel do Linux aloca os IDs de forma sequencial
  - Se pid é 37, o próximo será 38, etc.
- ✓ Valor máximo de um PID
  - `/proc/sys/kernel/pid_max`

4194304

Pergunta: Qual é a linha de comando para obter o valor de pid_max?

Process ID (PID) - #2

- ✓ O PID do processo chamante é devolvido pela função `getpid()`
  - `pid_t getpid(void);`
- ✓ O PID do processo pai é obtido através de `getppid()`
  - `pid_t getppid(void);`

```
printf ("My pid=%jd\n", getpid ());
printf ("Parent's pid=%jd\n", getppid ());
```

Execução de aplicações

- ✓ Mas…
  - Se todos os novos processos executam o código do processo pai, como é que se pode executar uma outra aplicação?
    - A chamada ao sistema **fork** cria uma clone do processo pai
- ✓ Como se exexcuta uma outra aplicação?
  - `vim, ps, ls, find, firefox`,…
- ✓ Resposta
  - A família de chamada ao sistema "`exec`"
    - Substituem a imagem do processo chamante

A família *exec* de chamadas ao sistema

- "exec" system calls

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);

// GNU extension (requires _GNU_SOURCE)
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

- Uso da chamada ao sistema exec:
  - exec...(*application_to_be_run*)
- "exec"
  - Substituição da imagem do processo chamante por um dado executável
    - As funções com "p" fazem uso da variável do ambiente "PATH"
    - As funções com "v" obtém os parâmetros a partir de um vetor de strings
    - As funções com "l" obtém os parâmetros de uma lista, com os vários itens separados por "," e o fim da lista assinalado por "NULL"
- Example: **execl("/bin/ps", "ps", "aux",NULL);**

Exemplo – execução "ls" (1)

- ✓ Execução de "ls -a" com recurso a "execlp"
- ✓ Pergunta
  - ✓ Porquê a mensagem: "This cannot happen!"

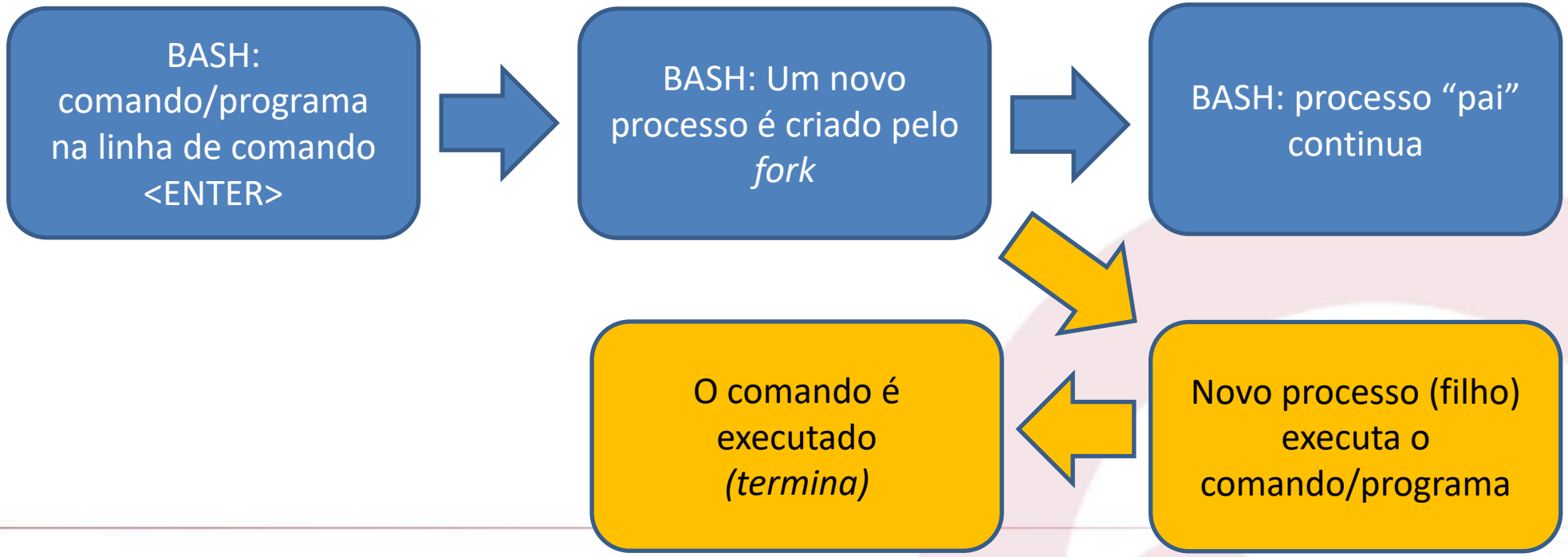
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
  if (execlp("ls", "ls", "-a", NULL) == -1)
    perror("Error executing ls: ");
  else
    printf("This cannot happen!\n");
  return 0;
}
```

Exemplo – execução "ls" (2)

- ✓ O processo executa o "ls" através do `execlp`
  - A chamada exec **NUNCA** retorna quando a execução é bem sucedida
  - A imagem do processo chamante é substituída pela imagem do executável através do "execlp"
    - O processo chamante corre o executável
      - "ls" no caso do exemplo
      - Deste modo, o código *printf("This cannot happen!")* é retirado da memória (bem como o restante códio)
      - O código é substituído por "ls -a"

Execução de um comando

- ✓ Execução de um comando/programa
  - Exemplo com a *shell bash*
    - O mesmo para outros *shells* (sh, zsh, etc.)
    - 1º – fork
    - 2º – Chamada ao sistema da família exec



Chamada ao sistema *wait*

- ✓ `wait` and `waitpid`

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

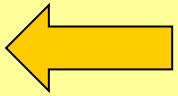
  - Chamadas ao sistema para sincronizar o processo pai com processos filhos
  - Espera pela troca de estado dos processos filhos
    - Processo filho é parado (SIGSTOP) ou termina
    - Processo filho rearranca com o *signal SIGCONT*
  - Exemple
    - `wait(&status);` Espera até ao término de um processo filho
    - `waitpid(-1, &status, 0);` Equivalente ao exemplo de cima

Processos estado "*zombie*"

- ✓ Um processo filho que termina, mas cujo pai não "esperou" por ele, é designado por "processo zombie"
- ✓ O kernel do SO mantém ainda informação referente a um processo zombie
  - PID, estado de terminação, uso de recursos, etc.
- ✓ Um processo zombie consome alguns recursos do kernel (espaço da informação)
- ✓ Caso o processo pai termina, todos os seus processos filhos em estado "zombie" são adotados pelo processo "init" que automaticamente executa um wait, eliminando os processos zombies.

Criação de zombies...

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
void worker() {
  printf("[%d] Hi, I'm a worker process! Going to die...\n",
         getpid());
}
int main()
{ int i;
  for (i=0; i<5; i++) {
    if (fork() == 0) {
      worker();
      exit(0);
    }
  }
  system("ps aux | grep -i zombie");
  printf("[%d] Big father is sleeping!\n", getpid());
  sleep(10);
  return 0;
}
```



Quantos processos são criados?

Criação de zombies...

✓ Resultados

```
user@so-vm:~$ ./zombies
[7291] Hi, I'm a worker process! Going to die...
[7290] Hi, I'm a worker process! Going to die...
[7289] Hi, I'm a worker process! Going to die...
[7288] Hi, I'm a worker process! Going to die...
[7287] Hi, I'm a worker process! Going to die...
user        7286  0.0  0.1   2356  1292 pts/2    S+   00:29   0:00 ./zombies
user        7287  0.0  0.0      0     0 pts/2    Z+   00:29   0:00 [zombies] <defunct>
user        7288  0.0  0.0      0     0 pts/2    Z+   00:29   0:00 [zombies] <defunct>
user        7289  0.0  0.0      0     0 pts/2    Z+   00:29   0:00 [zombies] <defunct>
user        7290  0.0  0.0      0     0 pts/2    Z+   00:29   0:00 [zombies] <defunct>
user        7291  0.0  0.0      0     0 pts/2    Z+   00:29   0:00 [zombies] <defunct>
user        7292  0.0  0.0   2608   604 pts/2    S+   00:29   0:00 sh -c ps aux | grep -i zombie
user        7294  0.0  0.0   9040   732 pts/2    S+   00:29   0:00 grep -i zombie
[7286] Big father is sleeping!
```

Função `system`

- ✓ No código de criação zombie está o seguinte código:

```
system("ps aux | grep -i zombie");
```

- ✓ A chamada "system" lança uma Shell e executa o comando indicado como parâmetro
  - Executa **/bin/sh –c command_line** e espera pelo respetivo término
  - O processo da *shell* é o que efetivamente executa a linha de comando
  - É uma chamada computacionalmente onerosa, dado que efetua i) `fork` de um processo e depois ii) `exec` de um processo shell

PARA ALÉM DO FORK…POSIX_SPAWN

Alternativa posix_spawn (#1)

- ✓ ‘posix_spawn’ e ‘posix_spawnp’
  - Criam um processo e executam o binário ‘path’
    - Correspondem a um *fork + exec*

```
#include <spawn.h>
```

- int **posix_spawn**(pid_t *restrict pid, const char *restrict path, const posix_spawn_file_actions_t *restrict file_actions, const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);

path aware

- int **posix_spawnp**(pid_t *restrict pid, const char *restrict file, const posix_spawn_file_actions_t *restrict file_actions, const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);

Alternativa posix_spawn (#1)

- ✓ Função ‘posix_spawn’
  - int **posix_spawn**(pid_t *restrict pid, const char *restrict path, const posix_spawn_file_actions_t *restrict file_actions, const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);
- ✓ Parâmetros
  - **pid**: passagem por referência para retorno do PID do novo processo
  - **files_actions**: aponta para um objeto *spawn file actions* que especifica ações relacionadas a serem executadas no filho entre as etapas fork(2) e exec(3). Este objeto é inicializado e preenchido antes da chamada posix_spawn() usando as funções posix_spawn_file_actions_init(3) e posix_spawn_file_actions_*().
  - **attrp**: aponta para um objeto attributes que especifica vários atributos do processo filho criado. Este objeto é inicializado e preenchido antes da chamada posix_spawn() usando as funções posix_spawnattr_init(3) e posix_spawnattr_*().
  - **argv:** especifica a lista de argumentos
  - **envp:** especifica a lista de variáveis do ambiente

Exemplo (#1)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main(void) {
  char *argv[] = {"ls", "-l", "/home", NULL};     // Argumentos para a aplicação a ser lançada
  char *envp[] = {NULL}; // Inherit the current environment variables
  pid_t child_pid;
  int status;
  // Novo processo com posix_spawn
  if (posix_spawn(&child_pid, "/bin/ls", NULL, NULL, argv, envp) != 0) {
    perror("posix_spawn");
    return 1;
  }
```

Continuação >>

Exemplo (#2)

(continuação)

```
// Aguarda término do processo filho
  if (waitpid(child_pid, &status, 0) == -1) {
    perror("waitpid");
    return 1;
  }
// Verificar o status de saída do processo filho (opcional)
if (WIFEXITED(status)) {
    printf("Child process exited with status %d\n", WEXITSTATUS(status));
  } else {
    printf("Child process terminated abnormally\n");
  }
  return 0;
}
```

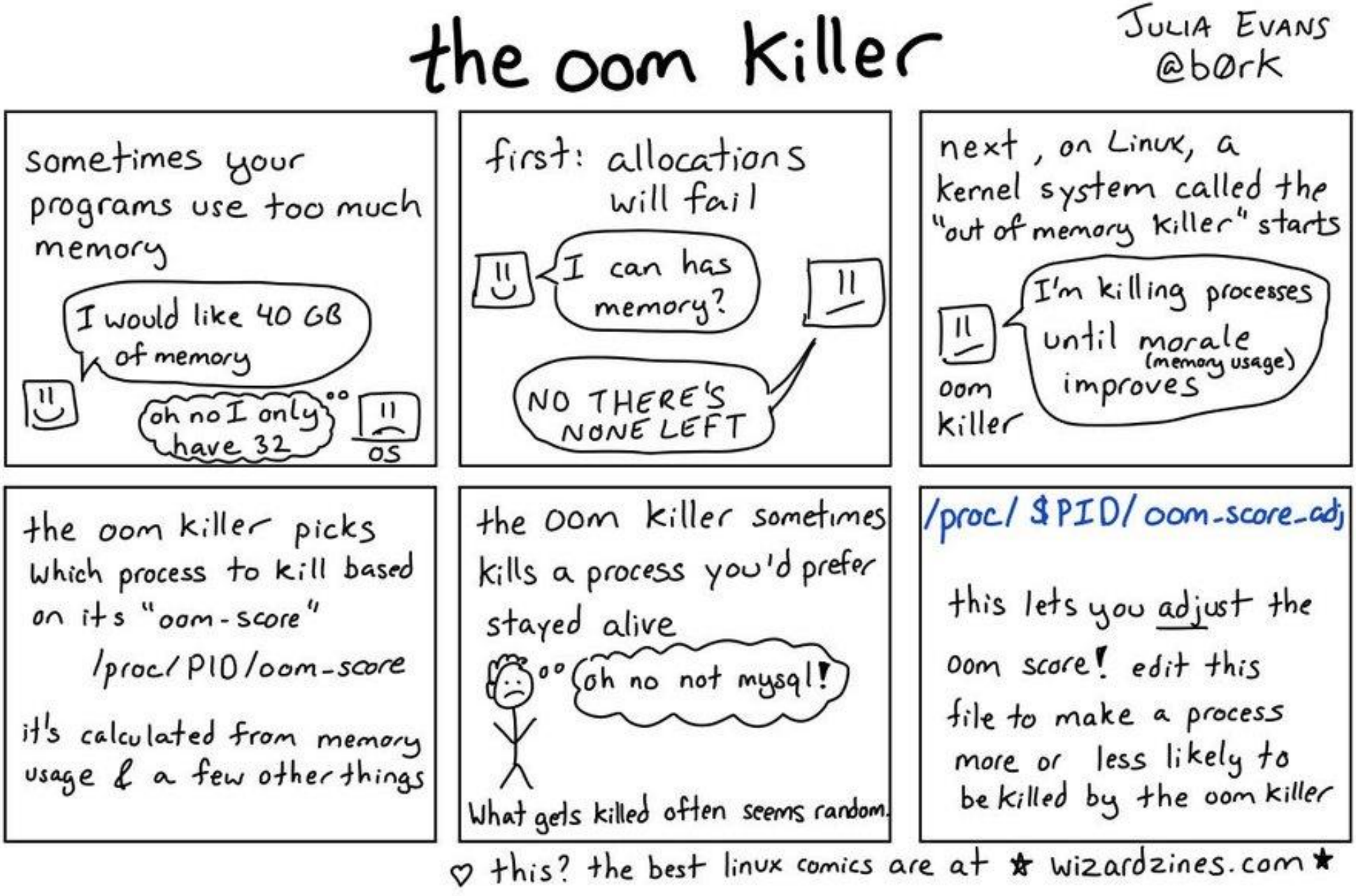
Término de um processo

- ✓ Motivos que levam a término de um processo
  - Término regular
    - exit, return da função main,…
  - Processo excedeu tempo máximo de CPU (e.g., “ulimit” no caso da *bash*)
  - SO sem memória suficiente
  - Falha de dispositivo de E/S
  - Instrução inválida (e.g., “divisão por zero”)
  - Ação do SO
    - *Deadlock* ou OOM (Out of Memory Killer)
  - Acção do utilizador
    - Kill -9 PID ou killall -9 process_name
  - …

A função **exit**

- A função **`exit`** termina o processo chamante
  - void exit(int **status**);
- Devolve o valor "status" ao sistema operativo
  - Na shell consegue obtém-se o valor de retorno da última execução através de $?
- Norma indica que:
  - Uma aplicação que execute normalmente deve devolver 0 (zero)
  - Em caso de erro deve devolver valor positivo entre 1 e 127.

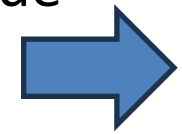
OOM killer


the oom killer
Julia Evans
@b0rk
sometimes your programs use too much memory
I would like 40 GB of memory
oh no I only have 32
OS
first: allocations will fail
I can has memory?
NO THERE'S NONE LEFT
next, on Linux, a kernel system called the "out of memory killer" starts
I'm killing processes until morale (memory usage) improves
oom killer
the oom killer picks which process to kill based on its "oom-score"
/proc/PID/oom_score
it's calculated from memory usage & a few other things
the oom killer sometimes kills a process you'd prefer stayed alive
oh no not mysql!
What gets killed often seems random.
/proc/$PID/oom_score_adj
this lets you adjust the oom score! edit this file to make a process more or less likely to be killed by the oom killer
♡ this? the best linux comics are at ★ wizardzines.com ★

UTILITÁRIOS PARA LISTAGEM E TRATAMENTO DE PROCESSOS

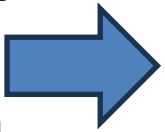
Processos - Utilitários

- **ps**
  - Lista processos
- **pstree**
  - Lista processos sob forma de árvore



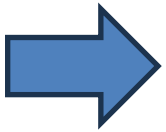
```
systemd-+-ModemManager---3*[{ModemManager}]
        |-NetworkManager---3*[{NetworkManager}]
        |-VGAuthService
        |-accounts-daemon---3*[{accounts-daemon}]
        |-avahi-daemon---avahi-daemon
        |-cron
        |-cups-browsed---3*[{cups-browsed}]
        |-cupsd---dbus
        |-dbus-daemon
        |-fwupd---5*[{fwupd}]
        |-2*[kerneloops]
        |-polkitd---3*[{polkitd}]
```

- **pgrep string**
  - Lista PIDs dos processos cujo comando corresponde totalmente/parcialmente a string



```
osboxes@osboxes:~$ pgrep ps
490
1215
1264
1273
1281
1307
```

- **pwdx PID**
  - Lista pasta corrente do processo PID



```
osboxes@osboxes:~$ pwdx $$
2402: /home/osboxes
```

REDIRECIONAMENTO DE CANAIS PADRÃO (STDOUT,STDERR)

Redirecionamento stdxx

- Redirecionamento através da linha de comando (bash, c-shell, zsh,...)
- Símbolos
  - >, >>, <,<<, …
- Exemplos
  - ps aux > ps.aux.txt
  - ls NaoExiste 2> err.txt
  - ls / *.xpto 2>&1 a.txt
  - …

- Mas... como podemos conseguir um redirecionamento stdout/stderr através da programação (por exemplo, usando a linguagem C)?

Tabela de descritores

- Cada processo tem uma tabela de descritores abertos
- As primeiras posições correspondem a:
  - 0: stdin
  - 1: stdout
  - 2: stderr
- O redirecionamento é realizado substituindo o descritor de um canal de destino pelo arquivo descritor de destino

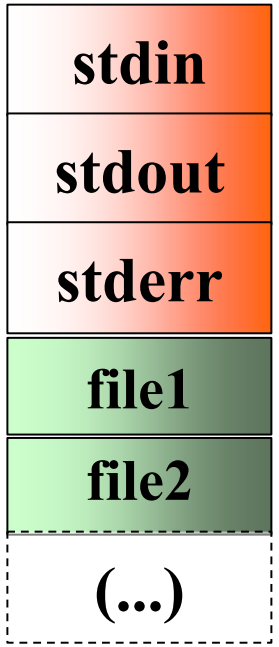


Tabela de descritores

(uma tabela por processo)

Redirecionamento em C (#1)

- Utilização da função dup2
  - `int dup2(int oldfd, int newfd);`
- Permite duplicar o descritor "newfd" para "oldfd"

- Exemplo

```
int f=open("out.txt",…);
dup2(fd, STDOUT_FILENO);
printf("stdout redirecionado para o ficheiro 'out.txt'\n")
```

```
#include <unistd.h>
STDERR_FILENO
    File number of stderr; 2.
STDIN_FILENO
    File number of stdin; 0.
STDOUT_FILENO
    File number of stdout; 1.
```

Redirecionamento em C (#2)

- O redirecionamento pode ser ativado antes de chamar uma função da família exec
  - O EXE executado por "exec" terá o *stdout/stderr* redirecionado
- **Exemplo**

```
#include <…>
int fd = open("out.txt",…);
dup2(fd, STDERR_FILENO);
close(fd);
/* EXEC */
execl("/bin/ps","ps",NULL);
```

Redirecionamento em C (#3)

- **A função freopen também permite o redirecionamento**

```
FILE *freopen(const char *pathname, const char *mode,
FILE *stream);
```

  - **A função redireciona o fluxo para o ficheiro localizado no nome do caminho e fecha o fluxo**
  - **Pode ser usado para redirecionar stdout/stderr para *pathname***
  - **Exemplo**
    - `#include <…>`

```
FILE *f = freopen("/tmp/out.txt","w+",stdout);
if (f==NULL){
   /* error */
}
```

Bibliografia

- Man pages
  - man 2 fork
  - man 2 exec
  - man 3 system
  - man 3 exit
  - man 2 dup
  - man bash
  - help ulimit
- Utilitário **tldr**

- *Chapter 5 – Process management*, “Linux System Programming”, Robert Love, 2013