Programação Avançada (ProgA)

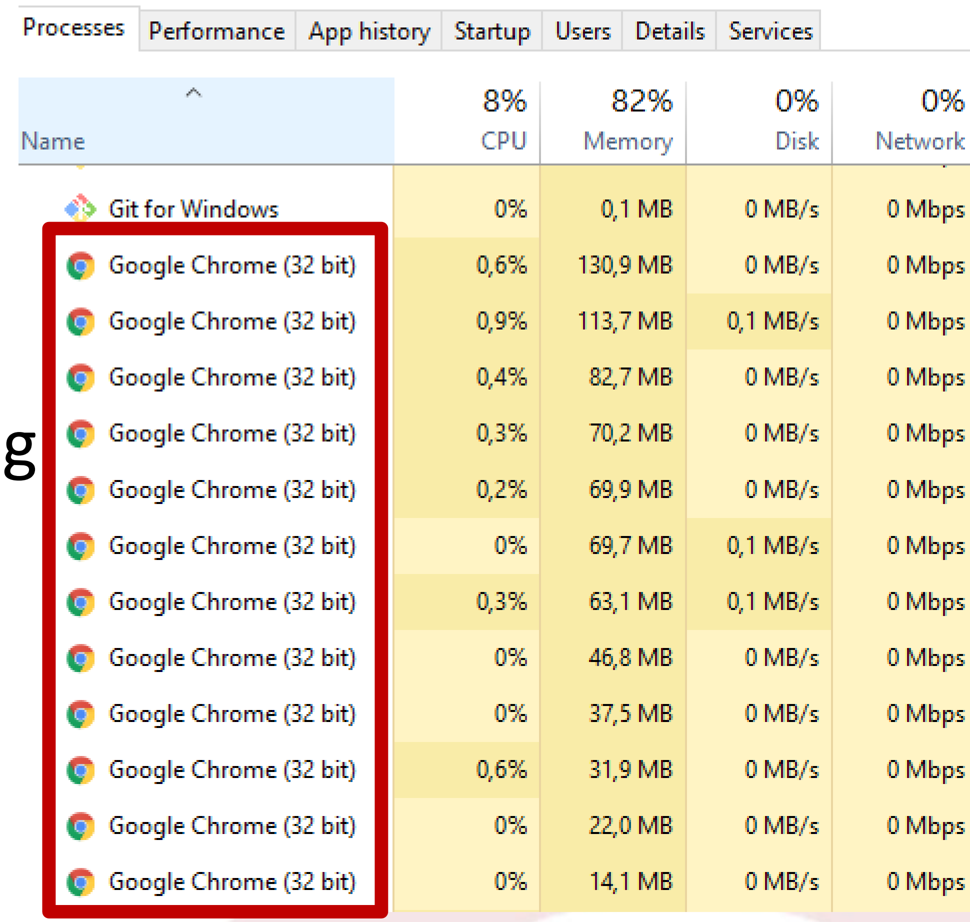
Chapter #1 - Processes

Patrício Domingues
ESTG/IPLeiria, 2019

escola superior de tecnologia e gestão
instituto politécnico de leiria

Process

- ✓ Program
  - – An executable file (**.exe** in Windows)
- ✓ Process
  - – A running program
  - – Several processes can be runing the same program
    - ▪ Example: chrome.exe
  - – Each process
    - ▪ **Program counter (PC)**
    - ▪ **Owner, PID**
    - ▪ **Security attributes**
    - ▪ **Address space**



Processes | Performance | App history | Startup | Users | Details | Services

Name	8% CPU	82% Memory	0% Disk	0% Network
Git for Windows	0%	0,1 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0,6%	130,9 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0,9%	113,7 MB	0,1 MB/s	0 Mbps
Google Chrome (32 bit)	0,4%	82,7 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0,3%	70,2 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0,2%	69,9 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0%	69,7 MB	0,1 MB/s	0 Mbps
Google Chrome (32 bit)	0,3%	63,1 MB	0,1 MB/s	0 Mbps
Google Chrome (32 bit)	0%	46,8 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0%	37,5 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0,6%	31,9 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0%	22,0 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)	0%	14,1 MB	0 MB/s	0 Mbps

Processes in Linux (#1)

- ✓ `top`

Número de tasks (~processos)

Carga média do sistema

```
File  Edit  Tabs  Help
top - 09:26:08 up  7:10,  1 user,  load average: 0.10, 0.04, 0.05
Tasks: 157 total,   1 running, 156 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  0.7 sy,  0.0 ni, 98.3 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :   630988 total,    31236 free,   194964 used,   404788 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   406196 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1996 root      20   0  253324  52416   9892 S  1.0  8.3   0:31.06 Xorg
 2550 user      20   0   75592  25740  19148 S  0.0  4.1   0:39.78 vmtoolsd
 2540 user      20   0  116076  25256  19776 S  0.0  4.0   0:03.64 pcmanfm
 2707 user      20   0   99088  24160  18820 S  1.0  3.8   0:09.60 lxterminal
 2454 user      20   0   95912  23544  18612 S  0.0  3.7   0:00.86 lxsession
 2538 user      20   0  107580  23544  19428 S  0.0  3.7   0:21.26 lxpanel
 2553 user      20   0  125216  22904  19044 S  0.0  3.6   0:01.10 nm-applet
14797 user      20   0   49928  17536  14512 S  0.0  2.8   0:00.06 vim
14785 user      20   0   49928  17476  14452 S  0.0  2.8   0:00.06 vim
 2535 user      20   0   73604  17428  13056 S  0.0  2.8   0:05.07 openbox
14749 user      20   0   49928  17360  14332 S  0.0  2.8   0:00.06 vim
14737 user      20   0   49928  17340  14316 S  0.0  2.7   0:00.12 vim
14773 user      20   0   49928  17280  14256 S  0.0  2.7   0:00.06 vim
14761 user      20   0   49928  17264  14236 S  0.0  2.7   0:00.05 vim
 2562 user      20   0   77116  16508  14308 S  0.0  2.6   0:00.24 xfce4-power+
 4205 user      20   0   50096  15156  11876 S  0.0  2.4   0:00.79 vim
 5537 user      20   0   49992  15028  11932 S  0.0  2.4   0:00.11 vim
```

Processes no Linux (#2)

- ✓ `htop`
  - by default, htop is not installed in ubuntu
  - `sudo apt-get install htop`

```
File  Edit  Tabs  Help

  CPU[|||                               6.2%]   Tasks: 79, 77 thr; 1 running
  Mem[|||||||||||||||||||||||||209M/616M]   Load average: 0.08 0.06 0.05
  Swp[                                 0K/0K]   Uptime: 07:16:10

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
14797 user       20   0 49928 17536 14512 S  0.0  2.8  0:00.06 vim
14785 user       20   0 49928 17476 14452 S  0.0  2.8  0:00.06 vim
14749 user       20   0 49928 17360 14332 S  0.0  2.8  0:00.06 vim
14737 user       20   0 49928 17340 14316 S  0.0  2.7  0:00.12 vim
14773 user       20   0 49928 17280 14256 S  0.0  2.7  0:00.06 vim
14761 user       20   0 49928 17264 14236 S  0.0  2.7  0:00.05 vim
 4205 user       20   0 50096 15156 11876 S  0.0  2.4  0:00.79 vim sc_open.c
 5537 user       20   0 49992 15028 11932 S  0.0  2.4  0:00.11 vim pc

F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Processes in Linux (#3)

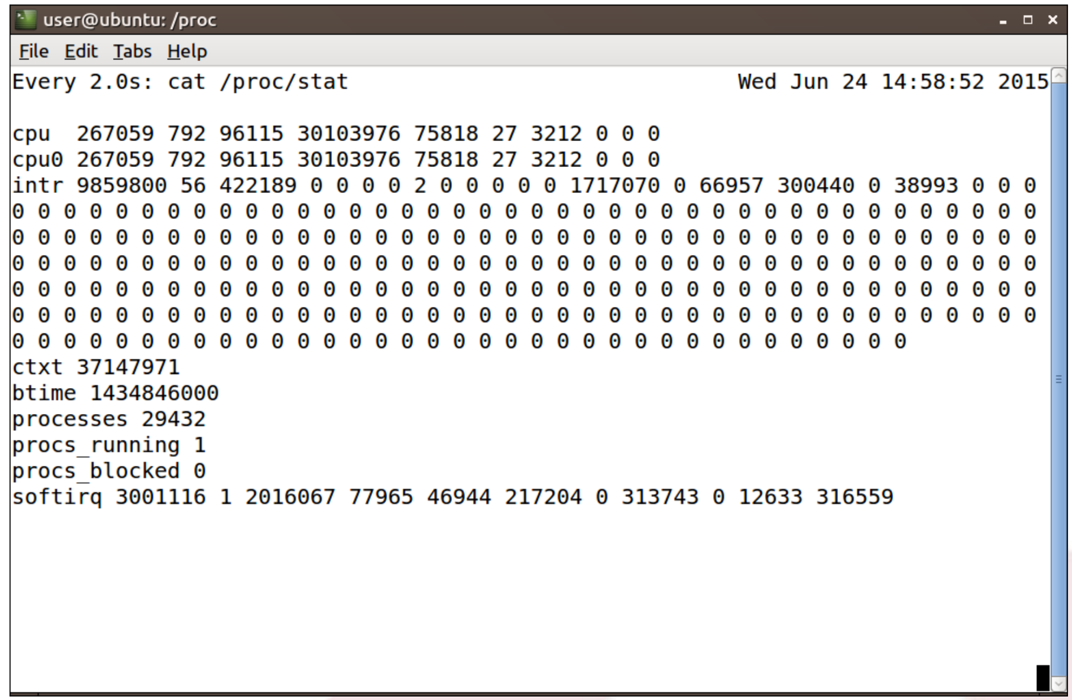
- The pseudo filesystem /proc keeps the data regarding the activity of the system
- For each process, there is a subdir in /proc
  - The name of the subdir is the PID of the process
  - NOTE: in bash, $$ holds the PID of the process running bash

```
user@ubuntu: /proc
File  Edit  Tabs  Help
1249   181   2154  2620  53    73          fb           scsi
13     1824  22    2625  54    74          filesystems  self
1323   1837  2244  2632  541   75          fs           slabinfo
14     1842  2246  2635  544   7520        interrupts   softirqs
14179  1843  23    2648  55    7535        iomem        stat
14264  1846  24    2653  5537  76          ioports      swaps
14725  1856  240   2658  558   78          irq          sys
14736  1874  2445  2663  56    8           kallsyms     sysrq-trigger
14737  1877  2446  2678  57    8436        kcore        sysvipc
14738  1881  2454  2707  58    9           keys         thread-self
14749  1891  2502  2708  586   92          key-users    timer_list
14750  1898  2505  28    59    93          kmsg         timer_stats
14761  19    2506  29    598   94          kpagecgroup  tty
14762  1902  2518  3     60    96          kpagecount   uptime
14773  191   2523  30    61    acpi        kpageflags   version
14774  1966  2535  31    62    asound      loadavg      version_signature
14785  1978  2538  337   63    buddyinfo   locks        vmallocinfo
14786  199   2540  4190  64    bus         mdstat       vmstat
14797  1996  2550  4205  643   cgroups     meminfo      zoneinfo
14820  2     2553  46    65    cmdline     misc
14889  20    2562  47    66    consoles    modules
14922  204   2575  49    67    cpuinfo     mounts
14956  206   2576  5     68    crypto      mtrr
user@ubuntu:/proc$
```

Processes in Linux (#4)

- File /proc/stat
  - Keeps stats regarding the operating system
- The **watch** command can be used to periodically execute a given command line
- `-n 2`: every 2 seconds

```
watch -n 2 cat /proc/stat
```



escola superior de tecnologia e gestão
instituto politécnico de leiria

Process Control Block - PCB

- ✓ Process Control Block – PCB
  - – OS struct that keeps the data of a process
  - – PCB is essential for multitasking
    - It allows the process to be interrupted and then resumed from its precise execution point

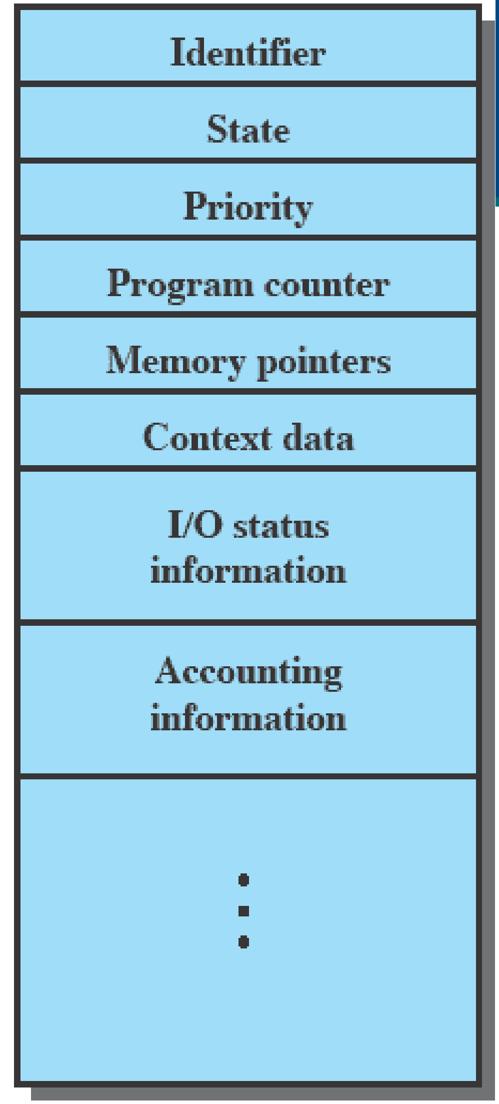
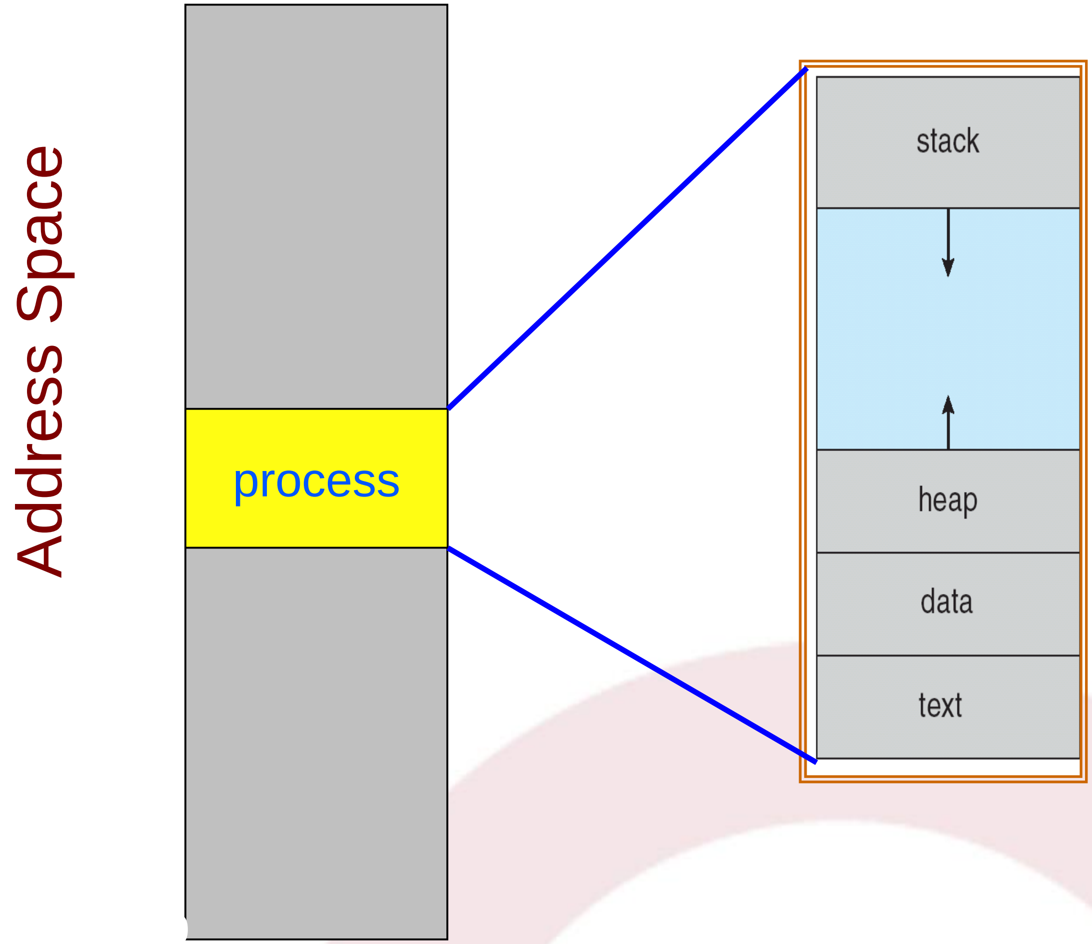


Figure 3.1 Simplified Process Control Block

Process in memory

- ✓ A process has several (logical) segments in memory
  - – Text segment
  - – Data segment
  - – Heap segment
  - – Stack segment



Process in memory (#2)

- ✓ Memory Image
  - (logical) representation of the process in memory
  - Several segments
- ✓ **text** segment
  - Holds the code/instruction of the process
- ✓ **data** segment
  - Holds variables in two subsections
    - .data: global and “static” variables, initialized with a non-zero value
    - .bss: global and “static” variables, non-initialized

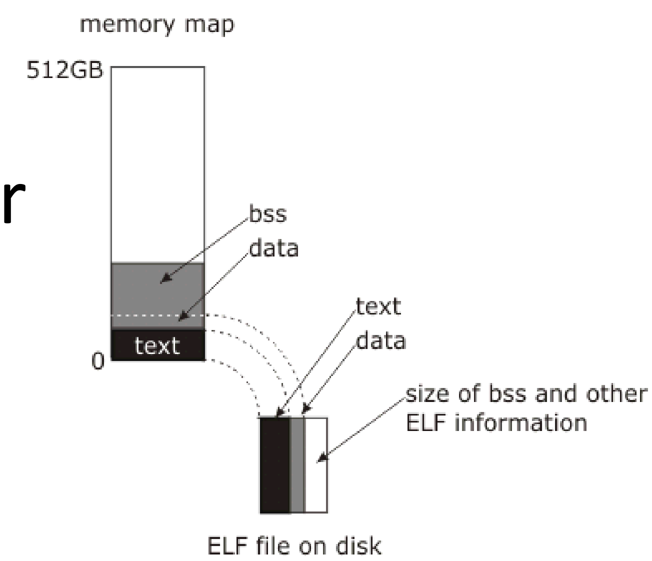
>>

Process in memory (#3)

- Heap segment
  - Used for dynamic memory
    - Example: malloc, calloc, realloc,…
    - It has a variable length
- Stack segment
  - Holds the so-called automatic/local variables
    - Variables that are automatically created and destroyed in functions and methods
    - It has a variable length

Executable file

- ✓ A process executes…an executable
- ✓ BSS: Block Started by Symbol
  - Data segment that hold static and globals variables set with zero as their initial value
  - BSS does not take space in the executable file



- ✓ DATA segment
  - "static" and "globals" variables set with an initial value different from zero
  - The executable file has a DATA segment

size command (1)

- size command
  - Unix tool that reports the length of each section and total size for a given executable file
- Example

```
#include <stdio.h>
int main(void){
        printf("Programa em C\n");
        return 0;
}
```

- `size a.exe`

  `text=877,data=256,bss=8,dec=1141,hex=475,filename=a.exe`
  - `877 + 256 + 8 = 1141 bytes (ou seja, 475 hexadecimal)`
  - Size of the executable file
    - 7136 bytes

size command (2)

- Example #2

```
#include <stdio.h>
char A[100000];  /* Vetor com 100000 chars = 100000 bytes */
int main(void){
        printf("Programa em C\n");
        return 0;
}
```

- `size a.exe`

  `text=877,data=256,`**`bss=100032,dec=101165,hex=18b2d`**`,filename=a.exe`

  – `877 + 256 + 100032 = 101165 bytes (ou seja, 18b2d hexadecimal)`

  – Size of the executable file
    - 7145 bytes
- Conclusion
  – Despite the declaration of the A vector to hold 100000 chars, the executable only increased 9 bytes (7136 to 7145 bytes)
  – Why? A is kept in BSS

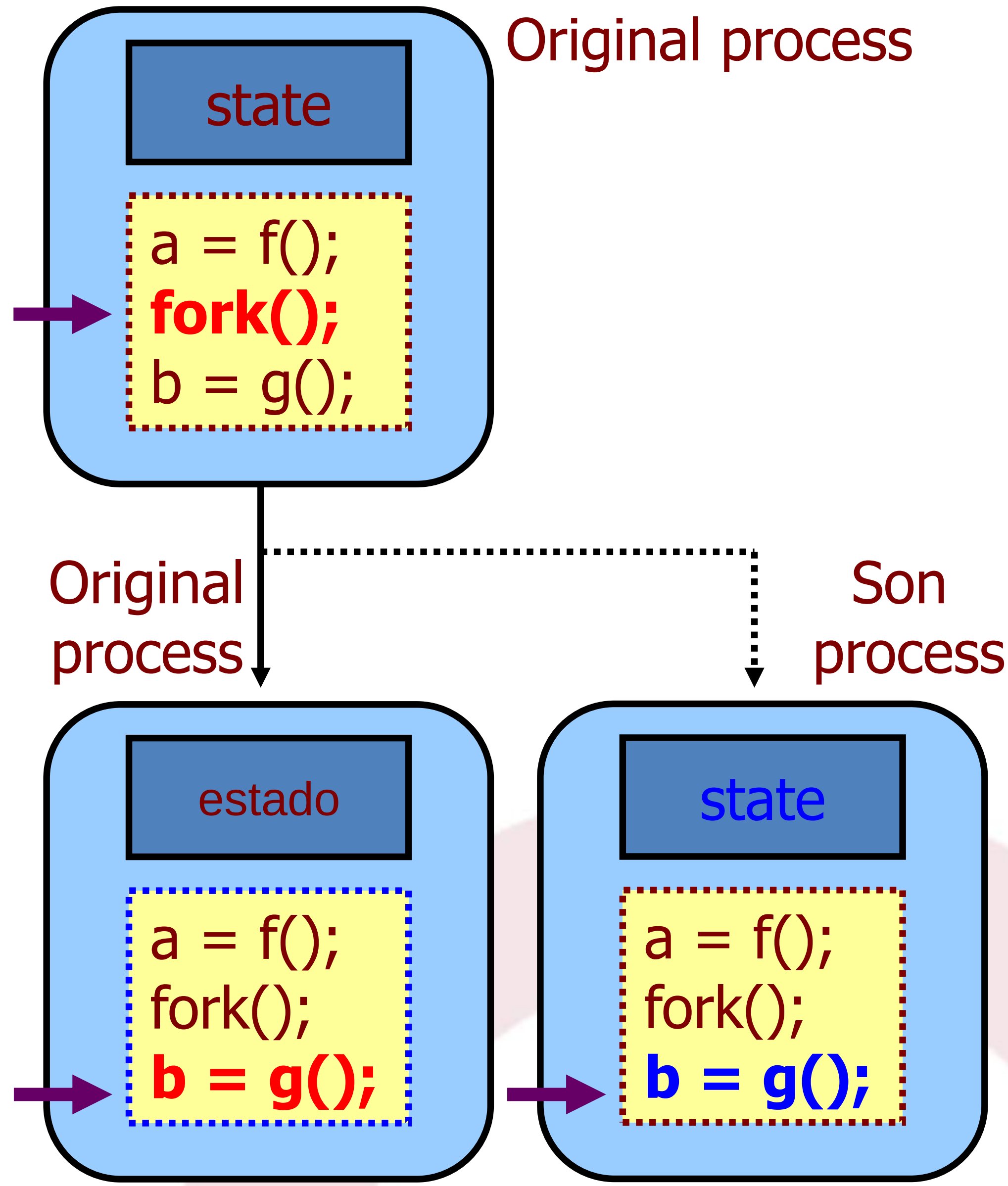
Processes in Unix (1)

- Processes in Unix
  - All are descendant from the *init* (PID=1) process
  - A process is created…by another process
    - Parent / son relationship
    - Tree of processes

```
user@ubuntu: ~
File  Edit  Tabs  Help
init-+-ModemManager---2*[{ModemManager}]
     |-NetworkManager-+-dhclient
     |                |-dnsmasq
     |                `-3*[{NetworkManager}]
     |-accounts-daemon---2*[{accounts-daemon}]
     |-acpid
     |-avahi-daemon---avahi-daemon
     |-bluetoothd
     |-cron
     |-cups-browsed
     |-cupsd
     |-dbus-daemon
     |-6*[getty]
     |-kerneloops
     |-lightdm-+-Xorg
     |         |-lightdm-+-init-+-at-spi-bus-laun-+-dbus-daemon
     |         |         |      |                 `-3*[{at-spi-bus-laun}]
     |         |         |      |-at-spi2-registr---{at-spi2-registr}
     |         |         |      |-dbus-daemon
     |         |         |      |-gconfd-2
     |         |         |      |-gvfs-afc-volume---2*[{gvfs-afc-volume}]
     |         |         |      |-gvfs-gphoto2-vo---{gvfs-gphoto2-vo}
     |         |         |      |-gvfs-mtp-volume---{gvfs-mtp-volume}
--More--
```

Process model in UNIX

- `fork()`
  - system call to create a process
  - It is called by the parent process
- The son process inherits all characteristics of the parent process
  - Variables, program counter, open files, allocated memory, etc.
  - The son is a snapshot of the parent
- After “fork”, each process executes separately
  - The change of a variable in one process does **not** reflect on the other one



`fork` system call

- ✓ `pid_t fork(void);`
- ✓ The fork system call returns an integer:
  - `0` to the newly created son process
  - `> 0` to the calling parent process
    - The return value corresponds to the PID of the newly created process
- ✓ It can also returns `-1` if an error has ocurred
  - `errno` holds the error code
  - `strerror(errno)` returns the error string

Example – fork system call

```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", getppid(), getppid());
    }
  else if (id > 0 )
    {/* code only executed by the parent process */
      printf("[%d] I'm the father!\n", getpid());
      wait(NULL);
    }
  return 0;
}
```

```
user@ubuntu: ~
File  Edit  Tabs  Help
user@ubuntu:~$ ./fork.exe
[10567] I'm the father!
[10568] I'm the son!
[10567] My parent is: 10567
```

Example – `fork` in a for loop

✓ How many processes are created by the following code?

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

Only newly created processes print their PID
Answer: 7
2^n -1, with n=3

Memory cost of a `fork`

- ✓ The fork system call creates a new process by cloning the current one
  - Does this mean that all the memory assigned to the cloned process is copied/duplicated?
    - NO, otherwise, a fork will be computationally expensive…
  - Unix uses the “**copy-on-write**” mechanism

Copy-on-write mechanism >>

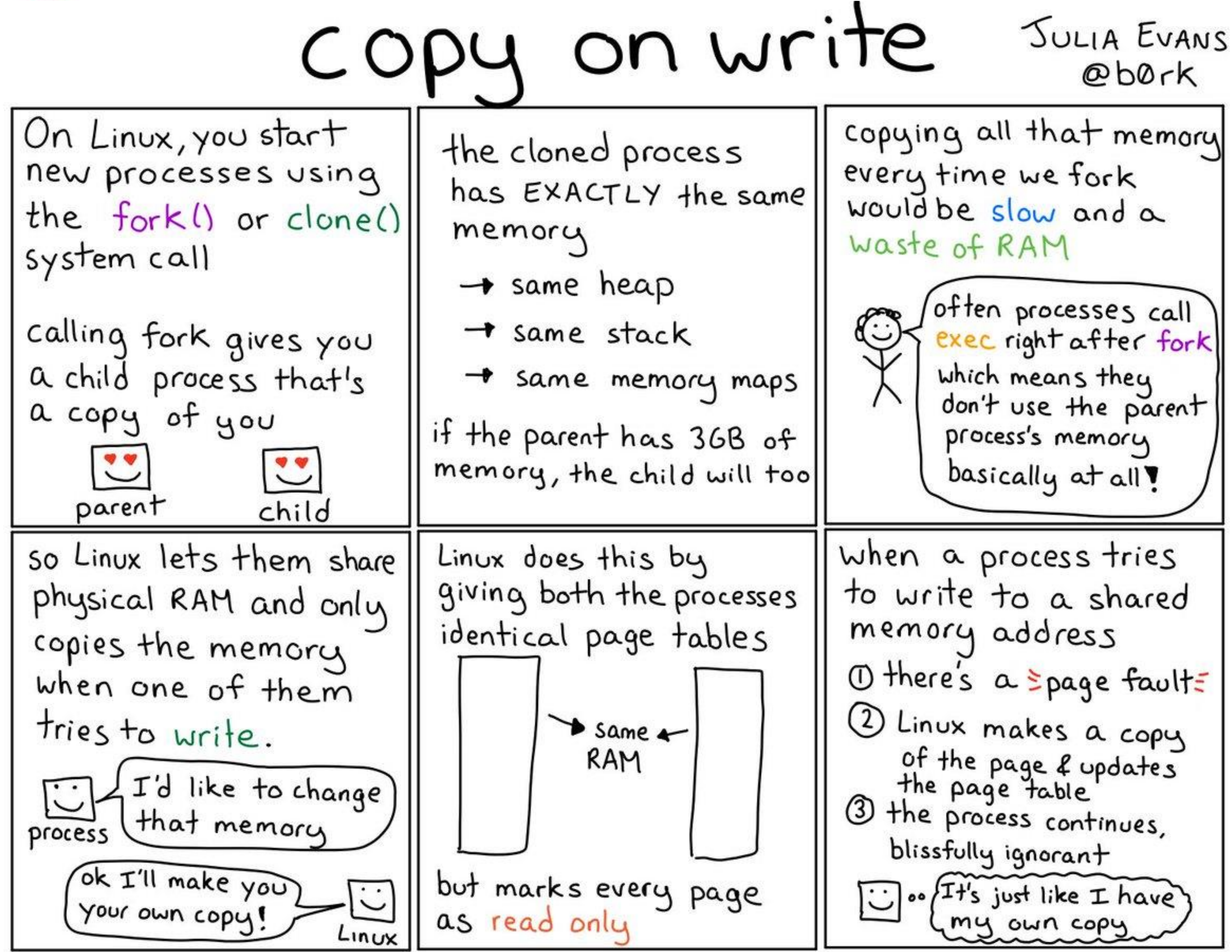
Copy-on-Write (#1)

- ✓ The OS organizes memory in fixed-size blocks called *pages (“paged memory”)*
  - *Usually, a page has 4 KiB*
- ✓ To avoid memory duplication, after a fork, the father and son process share the same memory pages
  - So there is no duplication of memory content

Continue >>

Copy-on-Write (#2)

- ✓ The shared pages are marked as **readonly**
- ✓ When one of the process tries to write a shared page (e.g, to change the value of a variable: `i++`)
  - – The OS flags it and creates a copy of the page
  - – Thereafter, both father and son processes have their own page
    - ▪ They still share the other pages
- ✓ Only written pages ("dirty") are duplicated
  - – This scheme is also known as *copy-on-demand*


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
parent
child
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
often processes call exec right after fork which means they don't use the parent process's memory basically at all!
so Linux lets them share physical RAM and only copies the memory when one of them tries to write.
process
I'd like to change that memory
ok I'll make you your own copy!
Linux
Linux does this by giving both the processes identical page tables
same RAM
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
It's just like I have my own copy

Process ID (PID) – #1

- ✓ Process ID (PID)
  - Integer identifier of a process
- ✓ The Linux kernel allocates process IDs to processes in a strictly linear fashion.
  - If pid 37 is the highest number currently allocated, pid 38 will be allocated next, even if the process last
- ✓ For compatibility with old UNIX, the max value for PID is 32768 (16-bit signed int)
- ✓ This value can be changed
  - `/proc/sys/kernel/pid_max`

Process ID (PID) - #2

- ✓ Within a C program, the PID of the calling process is returned with `getpid()`
  - `pid_t getpid(void);`
- ✓ The PID of the parent process is available through `getppid()`
  - `pid_t getppid(void);`

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

What is `intmax_t`?

- ✓ `intmax_t`
  - – Signed integer capable of representing any value of any signed integer type
- ✓ Defined by C99 and C11
  - – It requires `#include <inttypes.h>`
- ✓ The format string for printf is `%jd (decimal)`
  - – There is also the format string for hexadecimal representation
    - `%jx`

Running applications

- ✓ But…
  - – If all new processes execute the code of their parents, how can new applications be run?
    - The **fork** system call creates a clone of the parent process
- ✓ How do we run an application?
  - – `vim, ps, ls, find, firefox`,...
- ✓ Answer
  - – The "`exec`" family of system calls
    - These syscalls replace the image of the calling process

The *exec* family of system calls

- "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[]);
```

- Usage of an exec system call is
  - exec...(application_to_be_run)
- "exec"
  - Replaces the image of the current process by another one from a given executable
    - Functions with "p" are "PATH"-aware
    - Functions with "v" get their parameters from a vector of strings
    - Functions with "l" get their parameters from a list, where itens are separated by "," and the list ends with NULL
- Example: **execl("/bin/ps", "ps", "aux",NULL);**

Example - running “ls” (1)

- ✓ Running “ls -a” resorting to “execlp”
- ✓ Question
  - ✓ Why the “This cannot happen!” message?

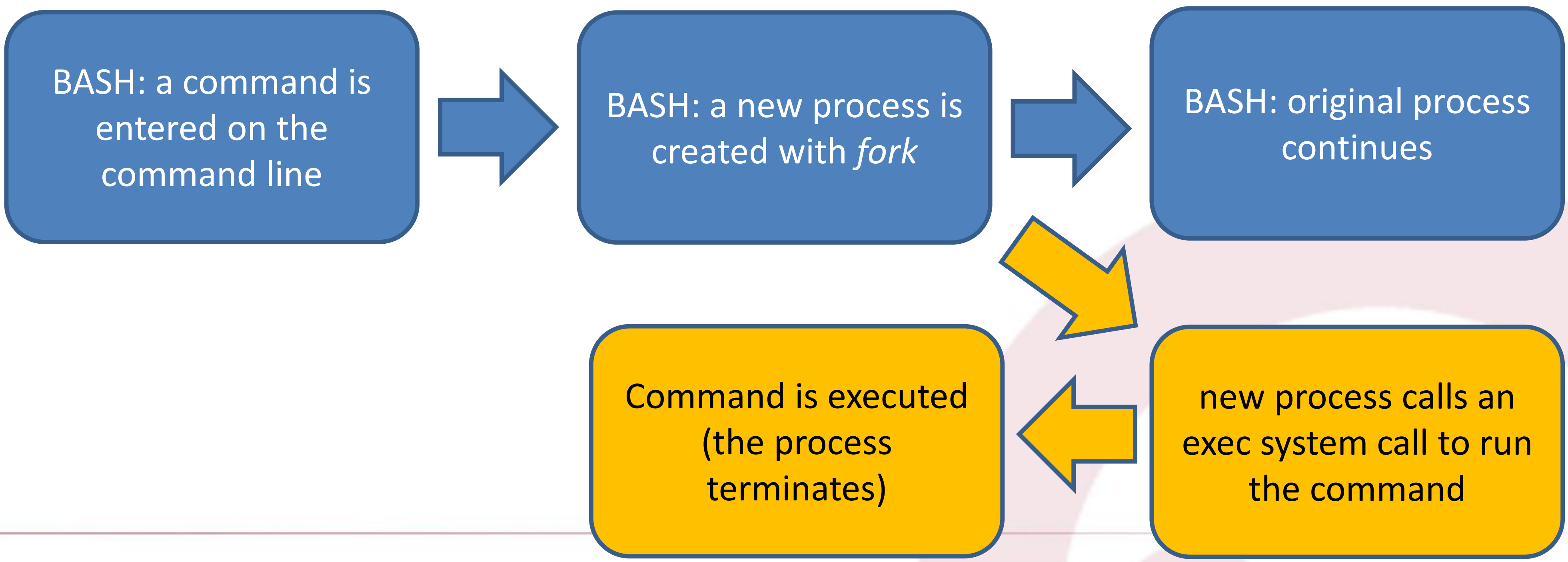
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
  if (execlp("ls", "ls", "-a", NULL) == -1)
    perror("Error executing ls: ");
  else
    printf("This cannot happen!\n");
  return 0;
}
```

Example – running “ls” (2)

- ✓ Process launches “ls” via execlp
  - The exec system calls **NEVER** return when the execution is successful
  - The calling process image is replaced by the image of the executable called via “exec”
    - The calling process runs the executable
      - “ls” in our example
      - Therefore, *printf(“This cannot happen!”)* is removed from memory (as well as all the code of the calling process)
      - The code is replaced by the code of “ls -a”

Executing a command

- ✓ Executing a command
  - – Example with bash
    - Applies to other shells (sh, zsh, etc.)
    - 1st – fork
    - 2nd – system call from the exec family



wait system call

- ✓ `wait` and `waitpid`

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

- – system calls used to synchronize a parent process with its children
- – Wait for state changes on their children processes
  - Child is stopped (SIGSTOP) or terminates
  - Child is resumed by a signal (SIGCONT)
- – Example
  - `wait(&status);` -- waits until a children process terminates
  - `waitpid(-1, &status, 0);` -- same as above

Zombie process

- ✓ A child that terminates, but has not been waited for becomes a "zombie“
- ✓ The kernel maintains a minimal set of information about the zombie process
  – PID, termination status, resource usage information
- ✓ As long as a zombie is not removed from the system via wait, it will consume a slot in the kernel process table
- ✓ If a parent process terminates, then its "zombie" children (if any) are adopted by *init*, which automatically performs a wait to remove the zombies

Creating zombies...

```
#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;
}
```

Question: how many processes are created?

Creating zombies…

✓ Results

user@ubuntu: ~/SO

File Edit Tabs Help

```
user@ubuntu:~/SO$ ./zombie.exe
[17512] Hi, I'm a worker process! Going to terminate...
[17513] Hi, I'm a worker process! Going to terminate...
[17514] Hi, I'm a worker process! Going to terminate...
[17511] Hi, I'm a worker process! Going to terminate...
[17510] Hi, I'm a worker process! Going to terminate...
user     17467  0.2  0.4  12348  4616 pts/16   S+   12:59   0:00 vim zombie.c
user     17509  0.0  0.0   2024   276 pts/5    S+   13:02   0:00 ./zombie.exe
user     17510  0.0  0.0      0     0 pts/5    Z+   13:02   0:00 [zombie.exe] <defunct>
user     17511  0.0  0.0      0     0 pts/5    Z+   13:02   0:00 [zombie.exe] <defunct>
user     17512  0.0  0.0      0     0 pts/5    Z+   13:02   0:00 [zombie.exe] <defunct>
user     17513  0.0  0.0      0     0 pts/5    Z+   13:02   0:00 [zombie.exe] <defunct>
user     17514  0.0  0.0      0     0 pts/5    Z+   13:02   0:00 [zombie.exe] <defunct>
user     17515  0.0  0.0   2268   552 pts/5    S+   13:02   0:00 sh -c ps aux | grep -i zombie
user     17517  0.0  0.0   4680   832 pts/5    S+   13:02   0:00 grep -i zombie
[17509] Big father is sleeping!
user@ubuntu:~/SO$
```

The `system` function

- ✓ In the zombie code, we have the following line of code

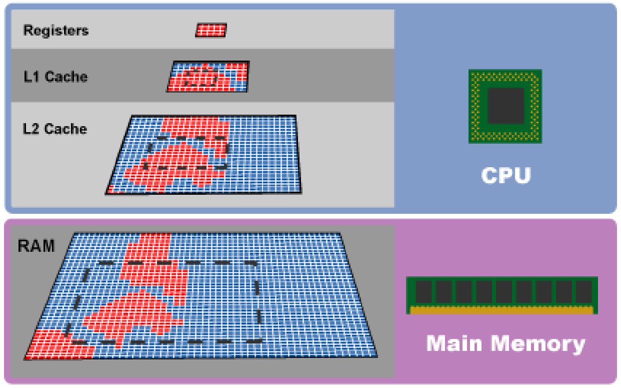
```
system("ps aux | grep -i zombie");
```

- ✓ system launches a shell that executes the command given as string
  - It executes **/bin/sh –c command_line** and waits for its termination
  - The `shell` process is the one that actually executes the command line
  - – It is costly, since it has to i) `fork` a process and then ii) `exec` its image to execute the Shell

CPU affinity (1)

- ✓ On a multicore system, the scheduler needs to decide which processes runs on each CPU
- ✓ Once a process is running on a CPU, it should remain there
  - – Avoid the “cold cache” effect of moving a process to another CPU/core
- ✓ How to enforce CPU affinity at the programming level?

CPU affinity (2)



http://bit.ly/256cAlr

- ✓ On a multicore system, the OS scheduler needs to decide which processes runs on each CPU
- ✓ Once a process is running on a CPU, the OS scheduler tries to keep it there
  - Avoid the "cold cache" effect of moving a process to another CPU/core
    - When a process moves to another CPU/core, the cache(s) of the CPU/core do not have content of the process
      - The caches are *cold*

CPU affinity (3)

- ✓ CPU affinity of a process can be controlled programatically
  - Hard affinity
- ✓ `int sched_setaffinity(pid_t pid, size_t setsize,const cpu_set_t *set);`
- ✓ `int sched_getaffinity(pid_t pid, size_t setsize,cpu_set_t *set);`
- ✓ void **CPU_SET** (unsigned long cpu, cpu_set_t *set);
- ✓ void **CPU_CLR** (unsigned long cpu, cpu_set_t *set);
- ✓ int **CPU_ISSET** (unsigned long cpu, cpu_set_t *set);
- ✓ void **CPU_ZERO** (cpu_set_t *set);

CPU affinity (4)

✓Example

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>cpu_set_t set;
int ret, i;
CPU_ZERO (&set);
ret = sched_getaffinity(0, sizeof (cpu_set_t), &set);
if (ret == -1){
        perror ("sched_getaffinity");
}
for (i=0; i < CPU_SETSIZE; i++) {
        int cpu;
        cpu = CPU_ISSET(i,&set);
        printf ("cpu=%i is %s\n", i, cpu?"set":"unset");
}
```

Termination of a process

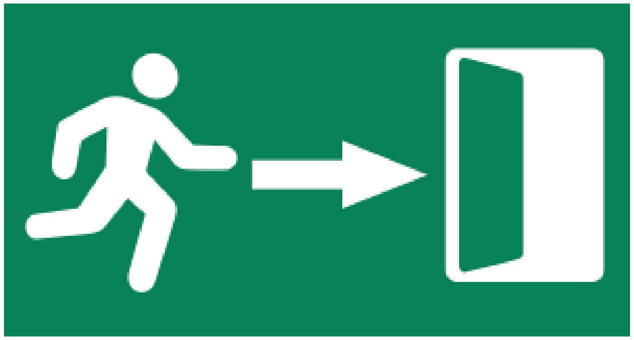
✓ Reason for a process to terminate
- Regular termination
  - exit, return of main function, etc.
- Process has exceeded maximum CPU time (e.g., “ulimit” from bash)
- Not enough memory
- I/O failure
- Invalid instruction (e.g., “divide by zero”)
- OS action
  - Deadlock or OOM (Out of Memory Killer)
- User action
  - Kill -9 PID or killall -9 process_name
- …

the **exit** function

- ✓ The exit function terminates the calling process
  - void exit(int **status**);
- ✓ It returns the int **status** to the operating system
- ✓ The **exit** function can call other functions before terminating the process
  - It calls the function preivously registered with the following functions
    - atexit
    - on_exit

The _exit function

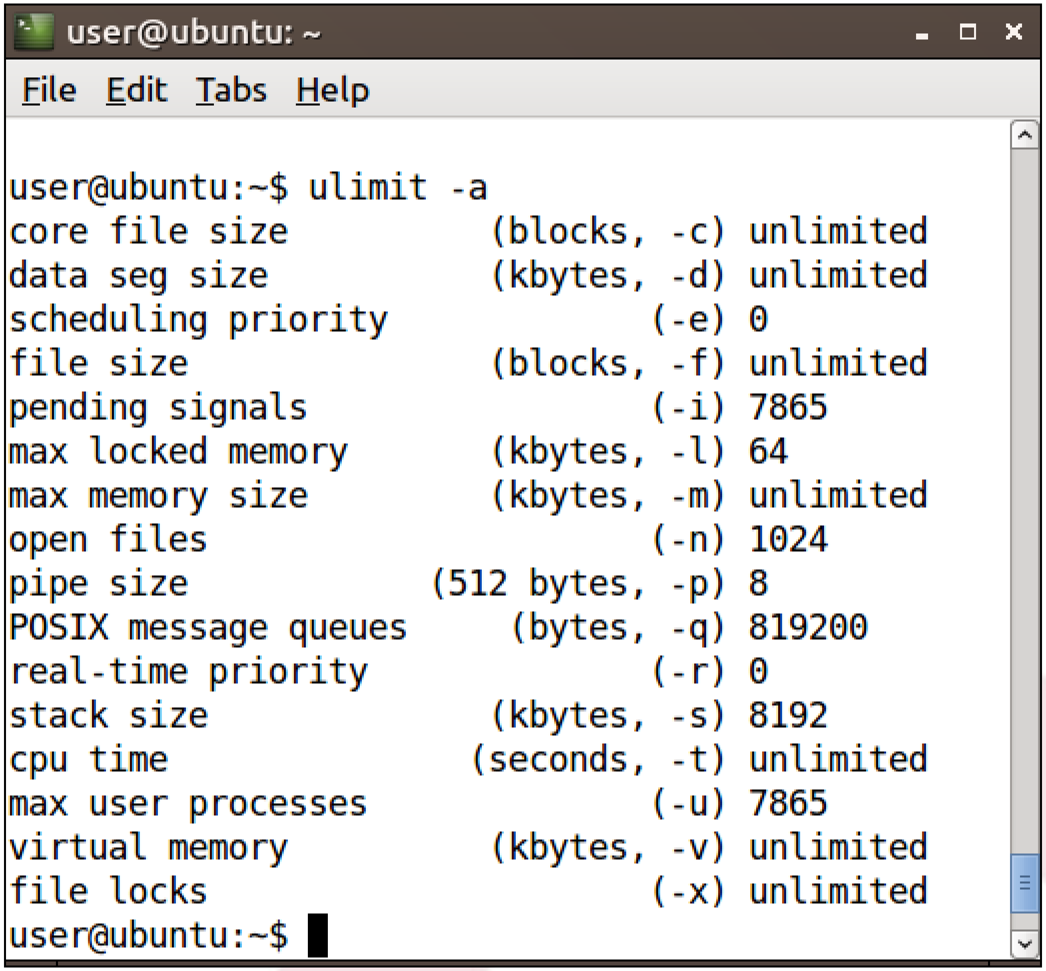
- What about to terminate the current process without calling the function registered with **atexit(3)** and **on_exit(3)**?



- Function **_exit**(2)
  - Note the leading underscore in the name

ulimits (bash)

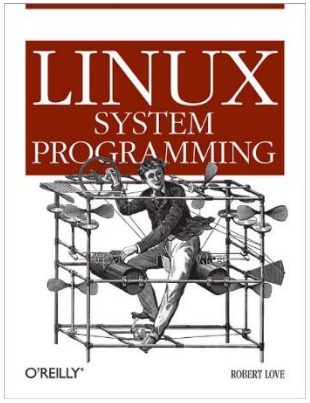
- The bash shell has an internal set of limits
  - `ulimit`
    - internal command
      - Processed by bash, there is no ulimit executable
    - There is no man for `ulimit`
      - help ulimit
  - `ulimit -a`
    - List all limits for the current session



Bibliography

- Man pages
  - man 2 fork
  - man 2 exec
  - man 3 system
  - man 3 exit
  - man bash
  - help ulimit

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



- printf format in C99 and C11
  http://bit.ly/1OvGzGl