escola superior de tecnologia e gestão
instituto politécnico de leiria

Sistemas Operativos
fork, exec & exit

Patrício Domingues

ESTG/IPLeiria

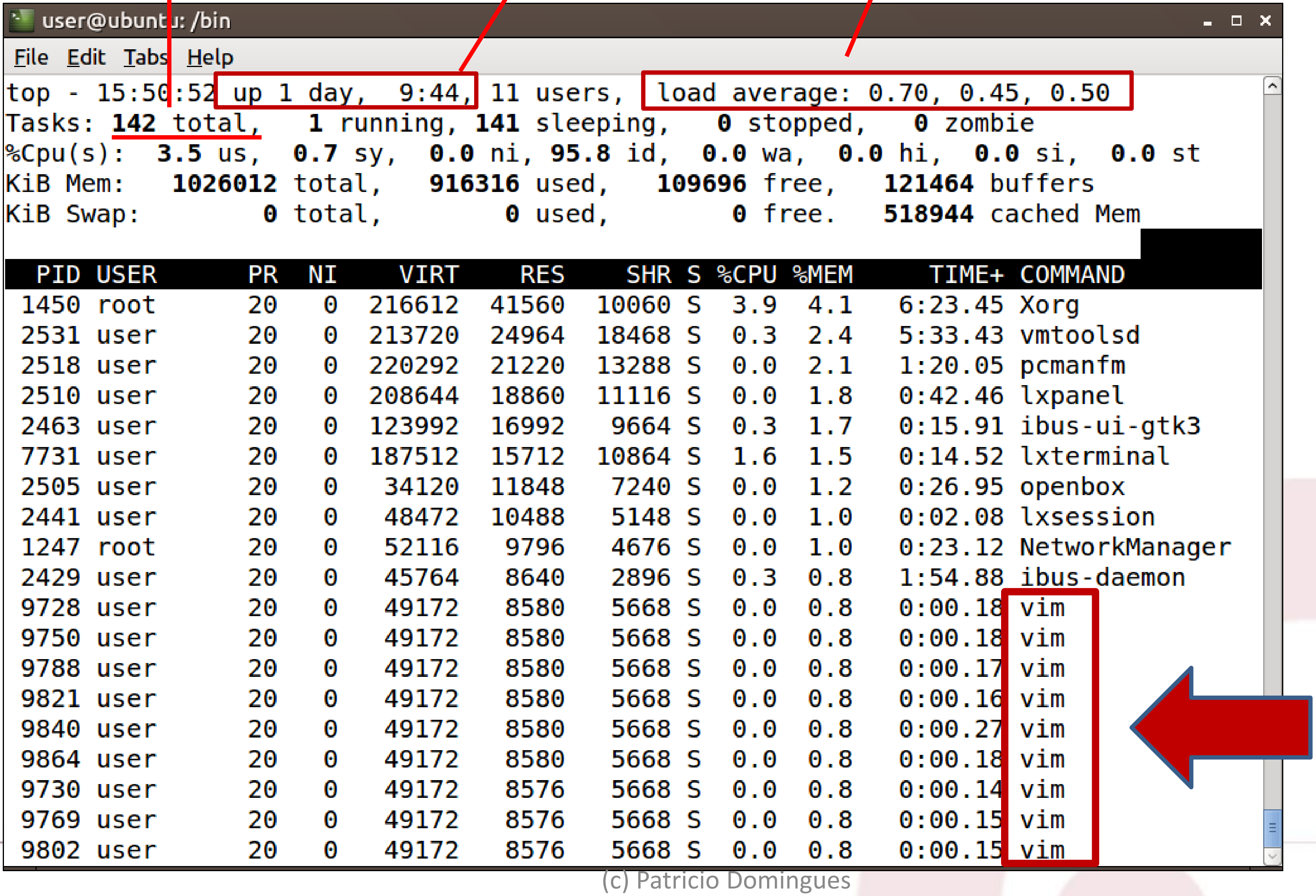
May 2019

“top” / linux

142 processes

uptime

Average load of the system

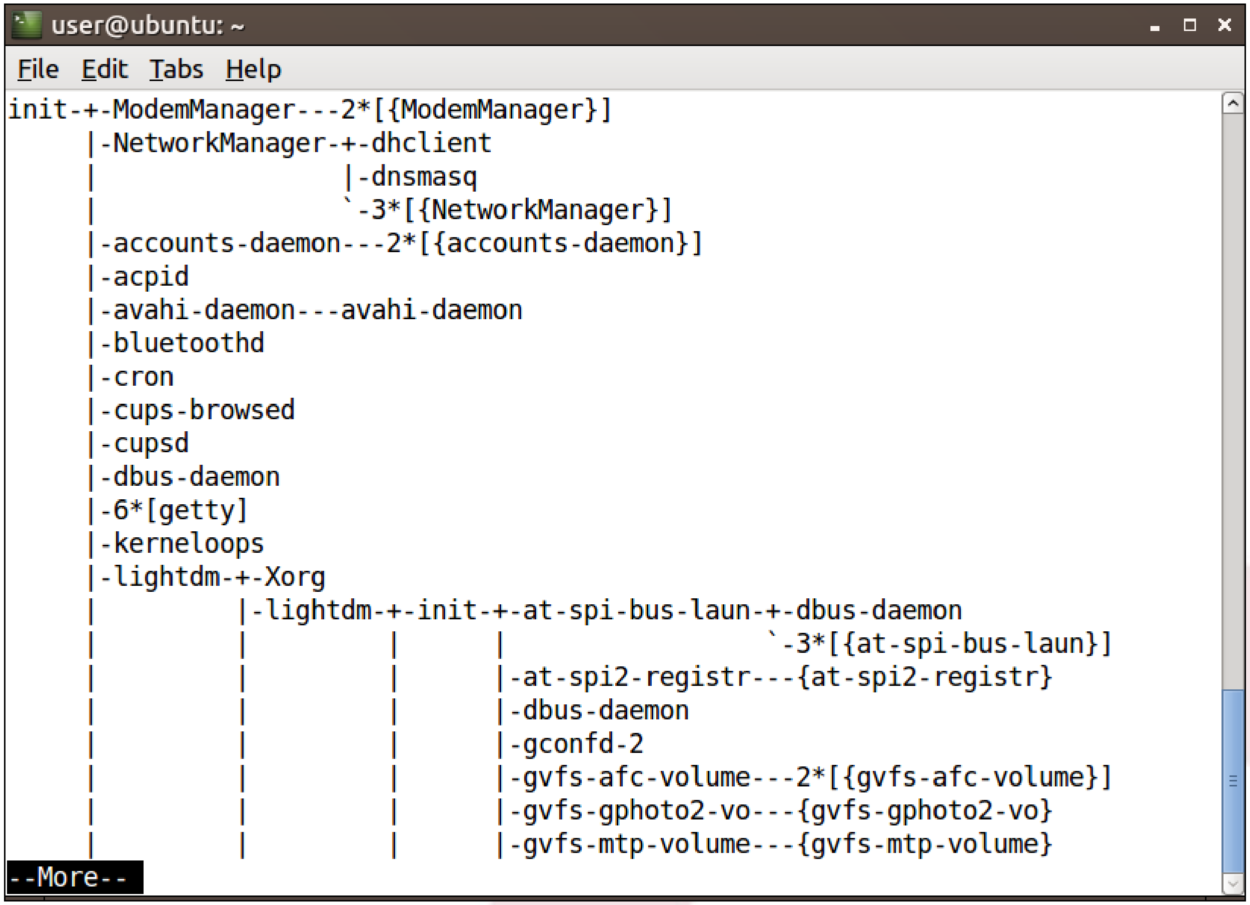


```
top - 15:50:52 up 1 day,  9:44, 11 users,  load average: 0.70, 0.45, 0.50
Tasks: 142 total,   1 running, 141 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.5 us,  0.7 sy,  0.0 ni, 95.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1026012 total,   916316 used,   109696 free,   121464 buffers
KiB Swap:        0 total,        0 used,        0 free.   518944 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1450 root      20   0  216612  41560  10060 S  3.9  4.1   6:23.45 Xorg
 2531 user      20   0  213720  24964  18468 S  0.3  2.4   5:33.43 vmtoolsd
 2518 user      20   0  220292  21220  13288 S  0.0  2.1   1:20.05 pcmanfm
 2510 user      20   0  208644  18860  11116 S  0.0  1.8   0:42.46 lxpanel
 2463 user      20   0  123992  16992   9664 S  0.3  1.7   0:15.91 ibus-ui-gtk3
 7731 user      20   0  187512  15712  10864 S  1.6  1.5   0:14.52 lxterminal
 2505 user      20   0   34120  11848   7240 S  0.0  1.2   0:26.95 openbox
 2441 user      20   0   48472  10488   5148 S  0.0  1.0   0:02.08 lxsession
 1247 root      20   0   52116   9796   4676 S  0.0  1.0   0:23.12 NetworkManager
 2429 user      20   0   45764   8640   2896 S  0.3  0.8   1:54.88 ibus-daemon
 9728 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.18 vim
 9750 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.18 vim
 9788 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.17 vim
 9821 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.16 vim
 9840 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.27 vim
 9864 user      20   0   49172   8580   5668 S  0.0  0.8   0:00.18 vim
 9730 user      20   0   49172   8576   5668 S  0.0  0.8   0:00.14 vim
 9769 user      20   0   49172   8576   5668 S  0.0  0.8   0:00.15 vim
 9802 user      20   0   49172   8576   5668 S  0.0  0.8   0:00.15 vim
```

Processes (1)

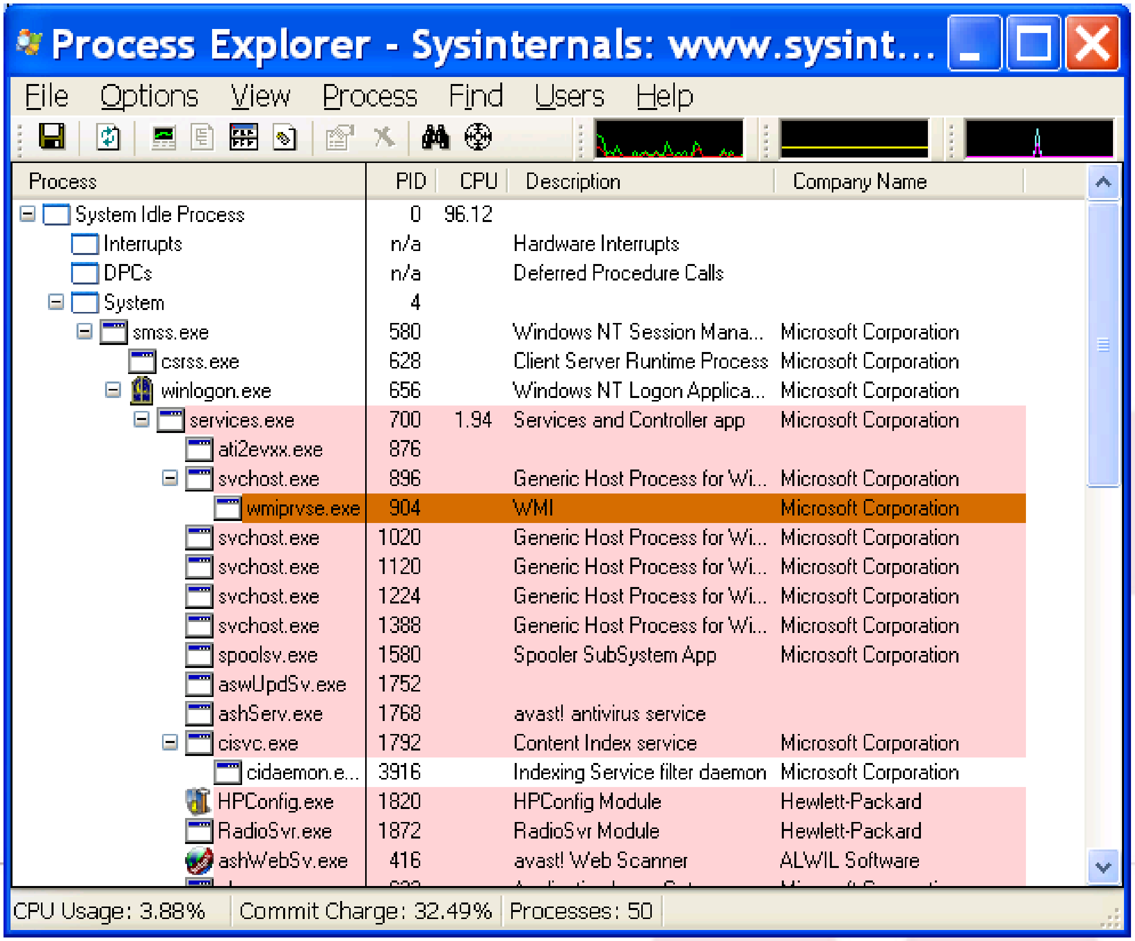
- ✓ Processes in Unix
  - All are descendant from the *init* (PID=1) process
  - A process is created…by another process
    - Parent / son relationship
    - Tree of processes



```
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--
```

Processes (2)

- ✓ There is also a process hierarchy in Microsoft Windows
  - – "process explorer" (sysinternals.com)

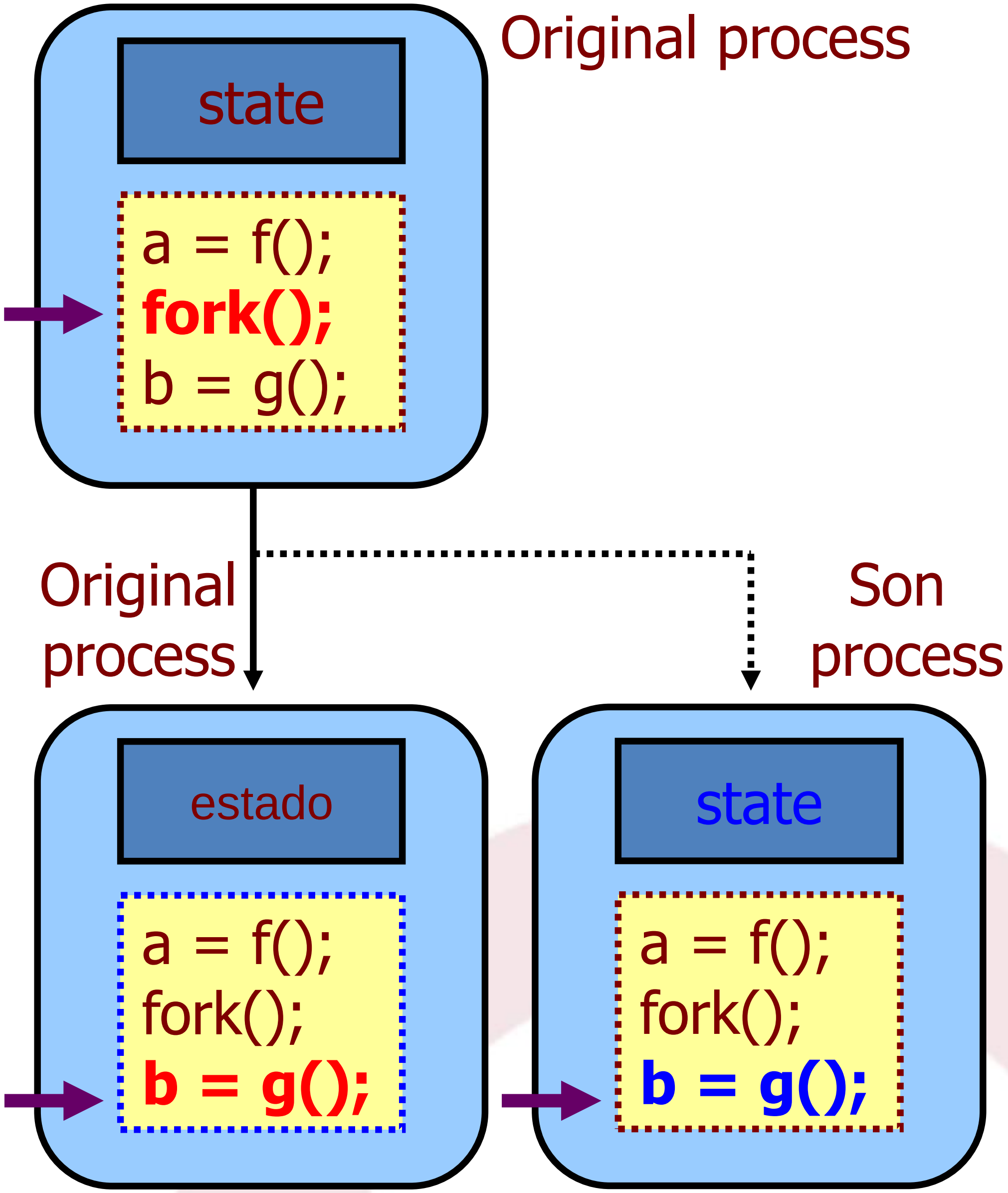


Process	PID	CPU	Description	Company Name
⊟ System Idle Process	0	96.12		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure Calls	
⊟ System	4			
⊟ smss.exe	580		Windows NT Session Mana...	Microsoft Corporation
csrss.exe	628		Client Server Runtime Process	Microsoft Corporation
⊟ winlogon.exe	656		Windows NT Logon Applica...	Microsoft Corporation
⊟ services.exe	700	1.94	Services and Controller app	Microsoft Corporation
ati2evxx.exe	876			
⊟ svchost.exe	896		Generic Host Process for Wi...	Microsoft Corporation
wmiprvse.exe	904		WMI	Microsoft Corporation
svchost.exe	1020		Generic Host Process for Wi...	Microsoft Corporation
svchost.exe	1120		Generic Host Process for Wi...	Microsoft Corporation
svchost.exe	1224		Generic Host Process for Wi...	Microsoft Corporation
svchost.exe	1388		Generic Host Process for Wi...	Microsoft Corporation
spoolsv.exe	1580		Spooler SubSystem App	Microsoft Corporation
aswUpdSv.exe	1752			
ashServ.exe	1768		avast! antivirus service	
⊟ cisvc.exe	1792		Content Index service	Microsoft Corporation
cidaemon.e...	3916		Indexing Service filter daemon	Microsoft Corporation
HPConfig.exe	1820		HPConfig Module	Hewlett-Packard
RadioSvr.exe	1872		RadioSvr Module	Hewlett-Packard
ashWebSv.exe	416		avast! Web Scanner	ALWIL Software

CPU Usage: 3.88% | Commit Charge: 32.49% | Processes: 50

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

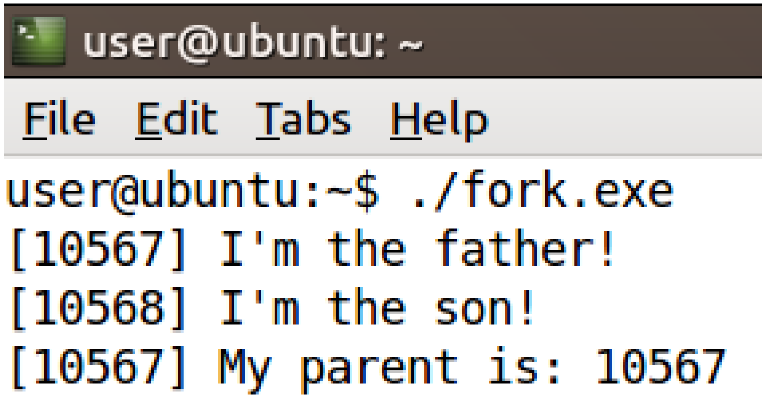
- ✓ `pid_t fork(void);`
- ✓ The fork system call returns an integer:
  - `0` to the newly created son process
  - `> 0` to the calling parent process
    - The return value corresponds to the PID of the newly created process
- ✓ It can also returns `-1` if an error has ocurred

Example – fork system call

```
#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);
    }
  return 0;
}
```



```
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?

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

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

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

`printf` format specifiers (2)

- ✓ Source: http://www.pixelbeat.org/programming/gcc/int_types/

```
uint32_t uint32=0xffffFFFF;
uintmax_t uintmax=UINTMAX_MAX;
off_t offset=TYPE_MAX(off_t); /* Depends on _FILE_OFFSET_BITS */
time_t time=TYPE_MAX(time_t); /* May be float! */
size_t size=TYPE_MAX(size_t); /* Depends on int size */

printf("native int bits %20zu %16x\n"
       "native long bits%20zu %16lx\n"
       "uint32_t max    %20"PRIu32" %16"PRIx32"\n"
       "uintmax_t max   %20ju %16jx\n" /* try PRIuMAX if %ju unsupported */
       "off_t max       %20jd %16jx\n" /* try PRIdMAX if %jd unsupported */
       "time_t max      %20jd %16jx\n"
       "size_t max      %20zu %16zx\n",
       sizeof(int)*CHAR_BIT, UINT_MAX,
       sizeof(long)*CHAR_BIT, ULONG_MAX,
       uint32, uint32,
       uintmax, uintmax,
       (intmax_t)offset, (intmax_t)offset,
       (intmax_t)time, (intmax_t)time,
       size, size);
```

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?

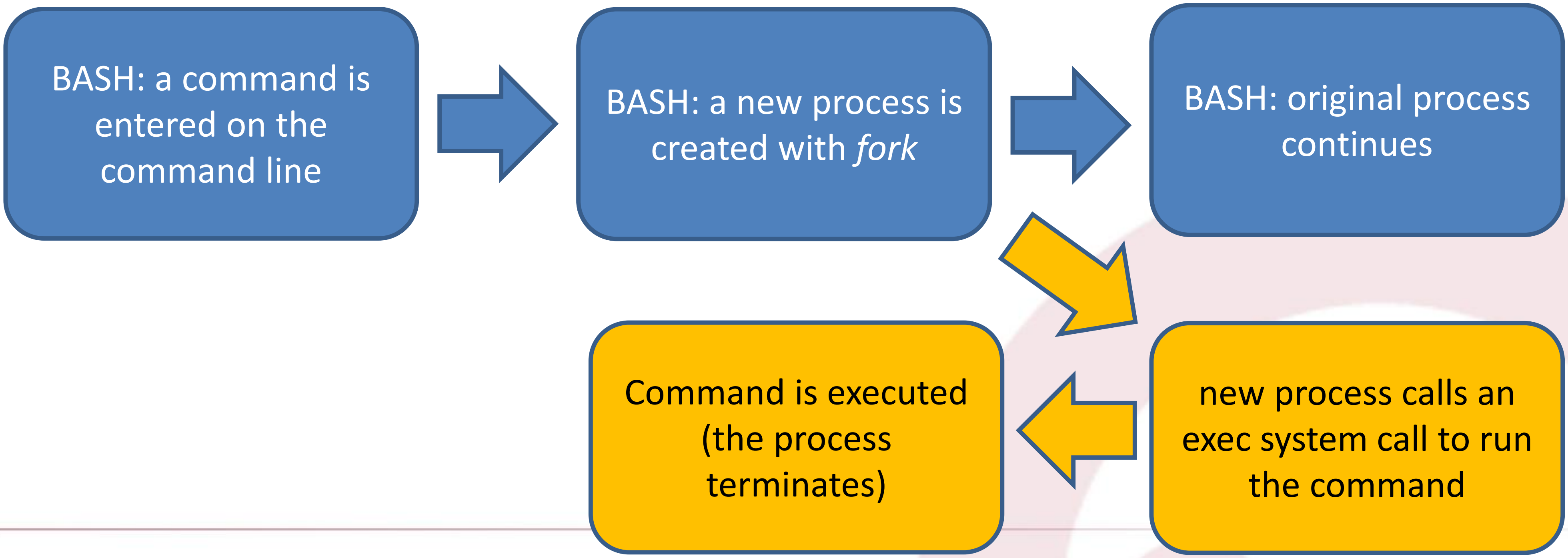
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
  if (execlp("ls", "ls", "-a", NULL) == -1)
    perror("Error executing ls: ");
  else
    printf("This cannot happen!\n");
  return 0;
}
```

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



BASH: a command is entered on the command line

BASH: a new process is created with *fork*

BASH: original process continues

new process calls an exec system call to run the command

Command is executed (the process terminates)

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

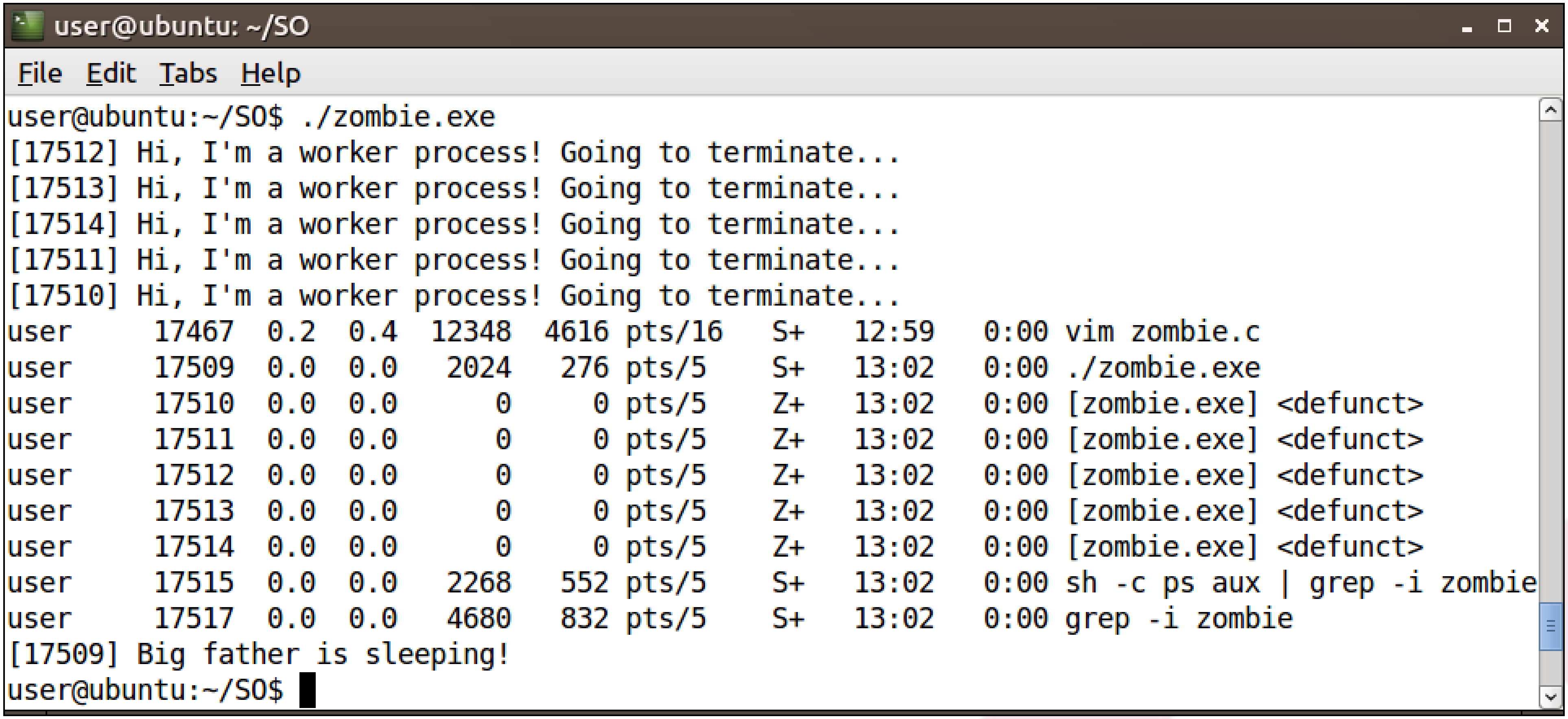
Creating zombies…

```
#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 defunct");
  printf("[%d] Big father is sleeping!\n", getpid());
  sleep(10);
  return 0;
}
```

Question: how many processes are created?

Creating zombies…

✓ Results



```
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$ 
```

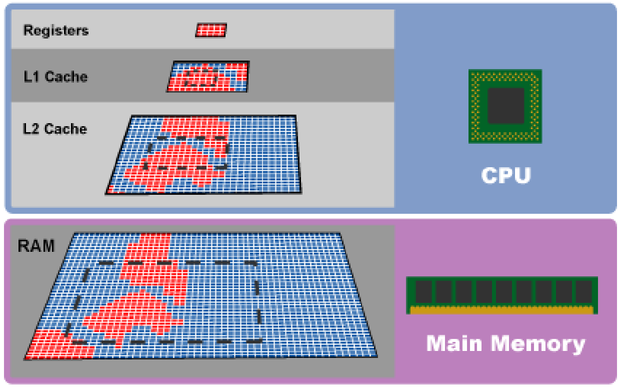
The `system` function

- ✓ 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)



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 (#2)

- ✓ 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 (#3)

✓Example

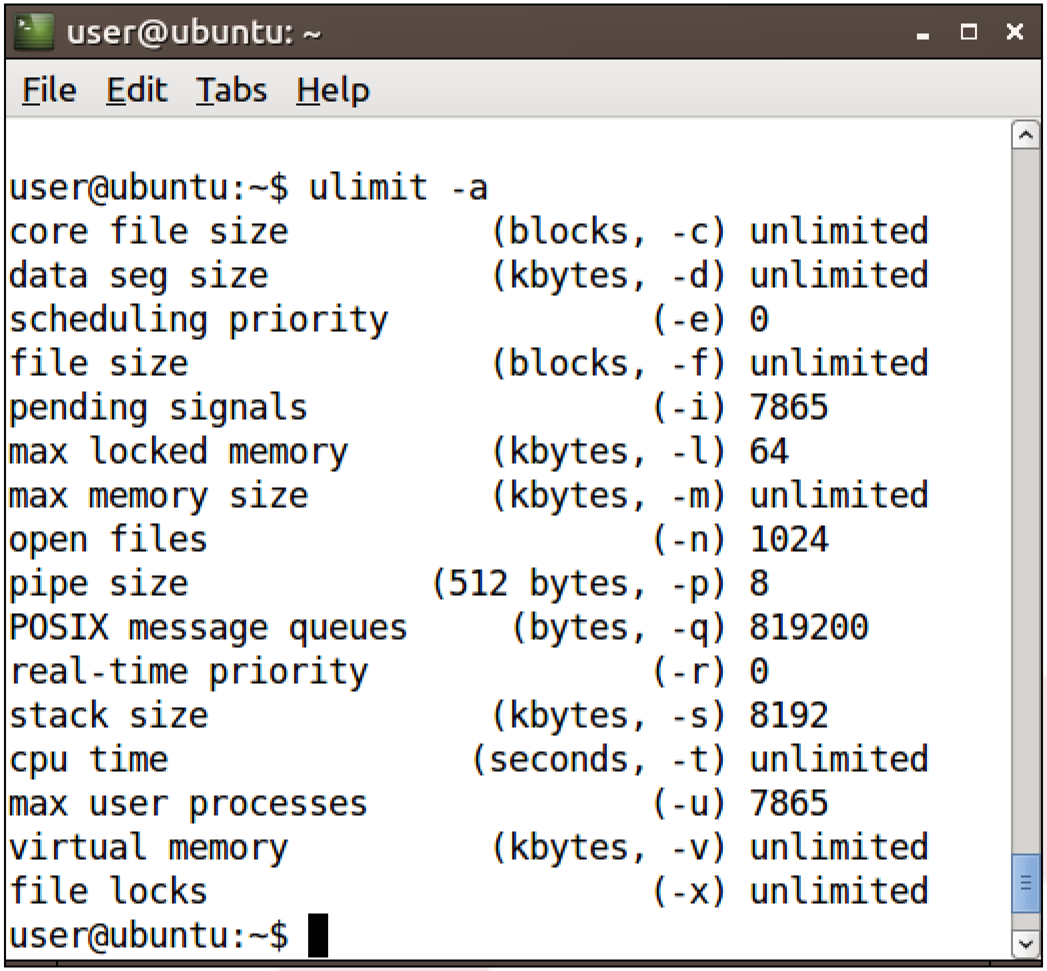
```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>cpu_set_t set;
int ret, i;
CPU_ZERO (&set);
ret = sched_getaffinity(0, sizeof (cpu_set_t), &set);
if (ret == -1){
      perror ("sched_getaffinity");
}
for (i=0; i < CPU_SETSIZE; i++) {
      int cpu;
      cpu = CPU_ISSET(i,&set);
      printf ("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
  - – …

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



```
user@ubuntu:~$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7865
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7865
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
user@ubuntu:~$ 
```

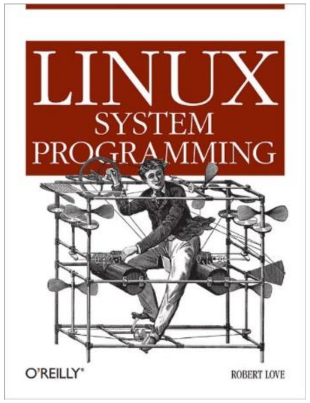
ulimits and core files

- ✓ ulimits has a parameter that controls the size of core file
  - `ulimit -c 99999999` ➔ unlimited
- ✓ With unlimited, the crash of an executable generates a core dump, that is, a file named “`core`”
- ✓ This core can be examined with a debugger, namely gdb
  - `gdb a.exe core`
  - The program needs to be compiled with the right options
    - ▪ `-g`
    - ▪ `-ggdb`

Bibliography

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

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



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