EPU - Informatique ROB4
Informatique Système
Introduction aux signaux

Miranda Coninx
Presented by
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr

Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

SORBONNE UNIVERSITÉ

ISIR
INSTITUT DES SYSTÈMES INTELLIGENTS ET DE ROBOTIQUE

Plan de ce cours

Processus : quelques approfondissements

- Quelques rappels
- Context switch vs mode switch
- Bloc de contrôle d'un processus (PCB) Linux

Signaux

- Définition
- Signaux et PCB
- Envoi et prise en compte d'un signal
- Prise en compte : quelques détails

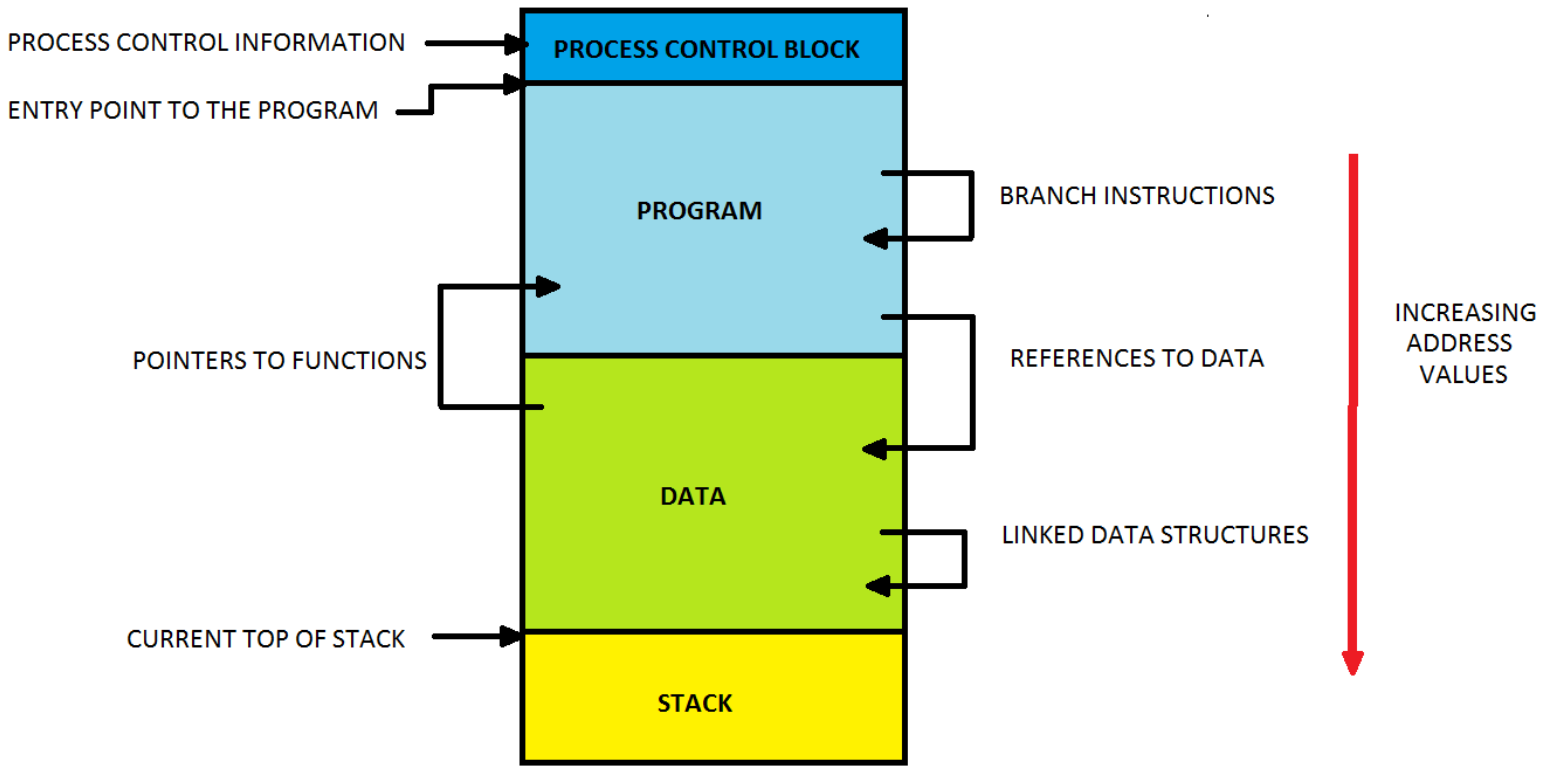
Programmation des signaux

- Envoi avec `kill ()`
- Blocage de signaux
- Associer une fonction à un signal
- Autres fonctions liées aux signaux

Process = program in execution

- Program code, resources (e.g. files in I/O), address space, ...
- Process Descriptor : PID, state, parent process, children, registers, address space information, open files
- One or more threads
- Can communicate through pipes, signals, network, files

↦ **a process in memory**



/proc - a pseudo-file system for process information

- `/proc/PID` : directory for each running process of pid PID
- `/proc/PID/maps` : currently mapped memory regions and access permissions
- `/proc/PID/stat` : the status information about the process, used by `ps`
  - PID, PPID, GID, ..
  - state (R=running, S=sleeping, D=waiting/disk sleep, Z=zombie, T=stopped on a signal, W=paging)
  - virtual memory, ...
- `/proc/PID/status` : the status information about the process in human-friendly mode (readable !)
  - A lot of signals : SigQ (total queued), SigPnd (pending), SigBlk (blocked), SigIgn (ignored), SigCgt (caught)
- `/proc/acpi` : advanced configuration and power interface (battery, etc.)
- `/proc/bus` : directory of installed busses
- `/proc/devices` : major numbers and device groups
- `/proc/interrupts` : interrupts per CPU per IO device
- `/proc/cpuinfo` : information about installed CPUs and cores
- `/proc/meminfo` : information about installed RAM

$\mapsto$ more info : `man proc`

↦ /proc/

```
icub@eva:/proc$ ls
1      1623  1793  1937  27   470  77          cpuinfo        modules
10     1660  18    1938  272  48   78          crypto         mounts
1019   1671  180   2     275  488  8           devices        mtrr
1026   17    1800  20    28   493  838         diskstats      net
1037   1706  1809  2046  29   5    843         dma            pagetypeinfo
1040   1709  1811  2059  3    50   855         dri            partitions
1046   1710  1814  2063  30   51   857         driver         sched_debug
1063   1718  1820  2076  31   52   866         execdomains    schedstat
1065   1727  1822  21    32   53   869         fb             scsi
1073   1729  1824  2140  33   54   870         filesystems    self
1075   1740  1826  2186  347  55   873         fs             slabinfo
1091   1744  1828  22    35   56   894         interrupts     softirqs
1092   1746  1830  2247  350  6    9           iomem          stat
1127   1752  1851  23    36   64   908         ioports        swaps
1136   1757  1853  2331  37   65   917         irq            sys
12     1762  1875  2347  38   66   925         kallsyms       sysrq-trigger
1241   1764  1881  24    39   664  927         kcore          sysvipc
1285   1765  1885  2450  4    667  934         key-users      timer_list
13     1769  1891  2451  40   67   98          kmsg           timer_stats
1302   1773  1896  2457  410  68   99          kpagecount     tty
1415   1774  19    2492  42   69   acpi        kpageflags     uptime
149    1780  1901  2493  43   7    asound      latency_stats  version
15     1784  1902  2494  44   70   buddyinfo   loadavg        version_signature
1573   1785  1910  25    45   71   bus         locks          vmallocinfo
1576   1787  1916  26    46   75   cgroups     mdstat         vmstat
16     1788  1924  261   469  761  cmdline     meminfo        zoneinfo
1608   1790  1931  262   47   768  consoles    misc
```

↦ /proc/1 (init)

```
icub@eva:/proc/1$ sudo ls
attr              cpuset   limits      ns              sched       syscall
autogroup         cwd      loginuid    numa_maps       schedstat   task
auxv              environ  maps        oom_adj         sessionid   wchan
cgroup            exe      mem         oom_score       smaps
clear_refs        fd       mountinfo   oom_score_adj   stack
cmdline           fdinfo   mounts      pagemap         stat
comm              io       mountstats  personality     statm
coredump_filter   latency  net         root            status
```

↦ /proc/1 (init)

```
icub@eva:/proc/1$ sudo ls -l
total 0
dr-xr-xr-x 2 root root 0 Oct 25 18:46 attr
-rw-r--r-- 1 root root 0 Oct 25 18:46 autogroup
-r-------- 1 root root 0 Oct 25 18:46 auxv
-r--r--r-- 1 root root 0 Oct 25 18:46 cgroup
--w------- 1 root root 0 Oct 25 18:46 clear_refs
-r--r--r-- 1 root root 0 Oct 25 18:40 cmdline
-rw-r--r-- 1 root root 0 Oct 25 18:46 comm
-rw-r--r-- 1 root root 0 Oct 25 18:46 coredump_filter
-r--r--r-- 1 root root 0 Oct 25 18:46 cpuset
lrwxrwxrwx 1 root root 0 Oct 25 18:46 cwd -> /
-r-------- 1 root root 0 Oct 25 18:46 environ
lrwxrwxrwx 1 root root 0 Oct 25 18:40 exe -> /sbin/init
dr-x------ 2 root root 0 Oct 25 18:46 fd
dr-x------ 2 root root 0 Oct 25 18:46 fdinfo
-r-------- 1 root root 0 Oct 25 18:46 io
-r--r--r-- 1 root root 0 Oct 25 18:46 latency
-r--r--r-- 1 root root 0 Oct 25 18:40 limits
-rw-r--r-- 1 root root 0 Oct 25 18:46 loginuid
-r--r--r-- 1 root root 0 Oct 25 18:46 maps
-rw------- 1 root root 0 Oct 25 18:46 mem
-r--r--r-- 1 root root 0 Oct 25 18:46 mountinfo
-r--r--r-- 1 root root 0 Oct 25 18:46 mounts
-r-------- 1 root root 0 Oct 25 18:46 mountstats
dr-xr-xr-x 5 root root 0 Oct 25 18:46 net
dr-x--x--x 2 root root 0 Oct 25 18:46 ns
-r--r--r-- 1 root root 0 Oct 25 18:46 numa_maps
-rw-r--r-- 1 root root 0 Oct 25 18:46 oom_adj
-r--r--r-- 1 root root 0 Oct 25 18:46 oom_score
-rw-r--r-- 1 root root 0 Oct 25 18:46 oom_score_adj
-r--r--r-- 1 root root 0 Oct 25 18:46 pagemap
```

↦ /proc/1 (init)

```
icub@eva:/proc/1$ cat status
Name: init
State: S (sleeping)
Tgid: 1
Pid: 1
PPid: 0
TracerPid: 0
Uid: 0 0 0 0
Gid: 0 0 0 0
FDSize: 64
Groups:
VmPeak:    24460 kB
VmSize:    24460 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      2404 kB
VmRSS:      2404 kB
VmData:     1000 kB
VmStk:       136 kB
VmExe:       152 kB
VmLib:      2596 kB
VmPTE:        64 kB
VmSwap:        0 kB
Threads: 1
SigQ: 0/46907
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000001000
SigCgt: 00000001a0016623
CapInh: 0000000000000000
CapPrm: ffffffffffffffff
CapEff: ffffffffffffffff
CapBnd: ffffffffffffffff
Cpus_allowed: ffff
Cpus_allowed_list: 0-15
Mems_allowed: 00000000,00000001
Mems_allowed_list: 0
voluntary_ctxt_switches: 1304
nonvoluntary_ctxt_switches: 200
```

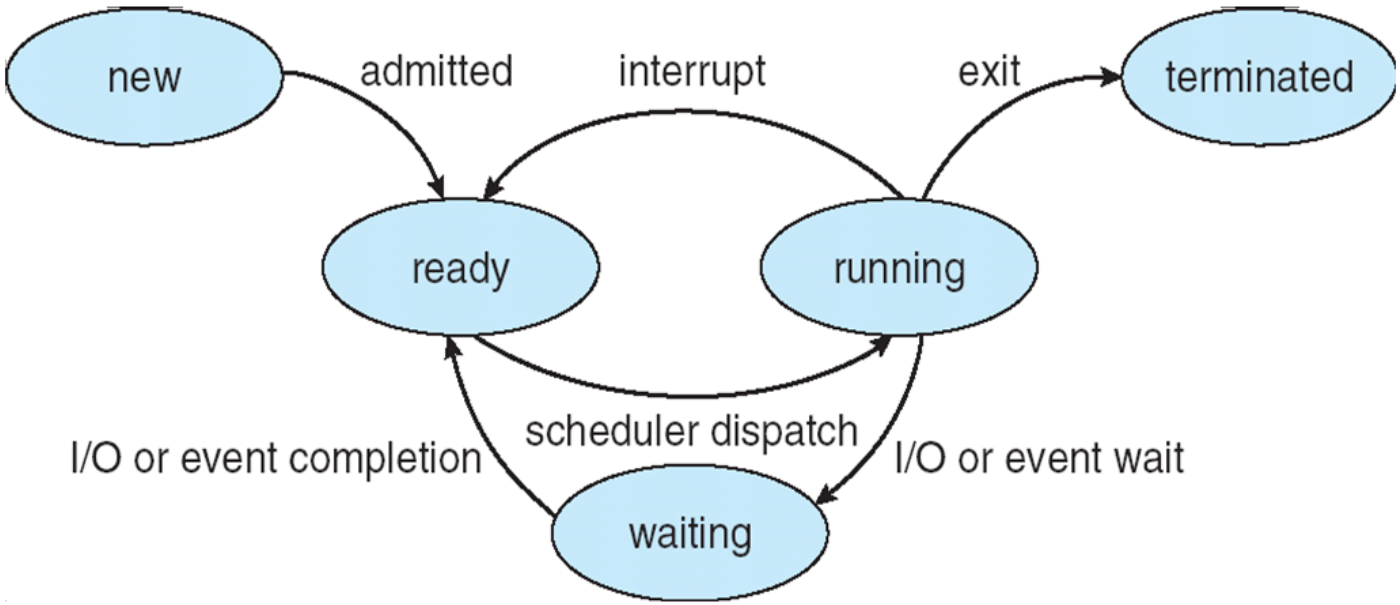
↦ /proc/68 (iwlwifi)

```
Name:   iwlwifi
State:  S (sleeping)
Tgid:   638
Ngid:   0
Pid:    638
PPid:   2
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 64
Groups:
NStgid: 638
NSpid:  638
NSpgid: 0
NSsid:  0
Threads:        1
SigQ:   0/46737
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: ffffffffffffffff
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
Seccomp:        0
Cpus_allowed:   ff
Cpus_allowed_list:      0-7
Mems_allowed:   00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0
```

↦ /proc/2247 (firefox)

```
icub@eva:/proc/2247$ cat status
Name: firefox
State: S (sleeping)
Tgid: 2247
Pid: 2247
PPid: 1
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 128
Groups: 4 24 27 30 46 109 124 1000
VmPeak:  1088444 kB
VmSize:   828008 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:    252736 kB
VmRSS:    155176 kB
VmData:   421336 kB
VmStk:       136 kB
VmExe:        68 kB
VmLib:     65100 kB
VmPTE:      1412 kB
VmSwap:        0 kB
Threads: 23
SigQ: 0/46907
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000001000
SigCgt: 00000001800044ef
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed: ffff
Cpus_allowed_list: 0-15
Mems_allowed: 00000000,00000001
Mems_allowed_list: 0
voluntary_ctxt_switches: 820280
nonvoluntary_ctxt_switches: 63791
```

$\mapsto$ **process : states**



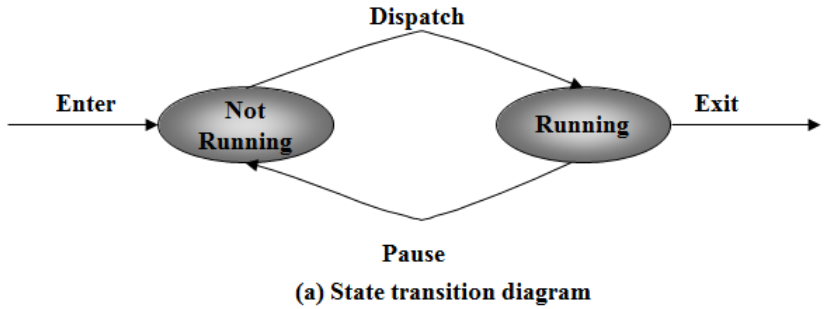
OS - Process Interaction

1. The OS interacts with all processes by **scheduling** the processes and managing each process lifecycle (creation, execution, termination)
   - maximize CPU utilization providing reasonable response time
   - very important for real-time constraints *(you will have a class about real-time programming next year)*

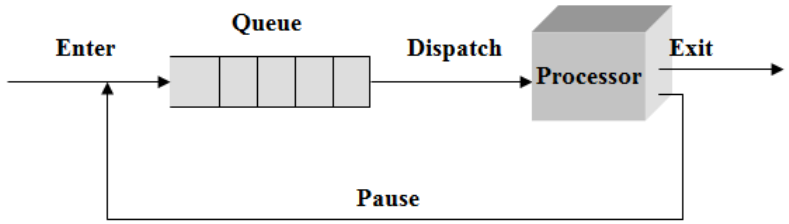
2. The process interacts with the OS via **system calls**, which are special instructions made available by the OS to perform "privileged" operations
   - user mode vs kernel mode
   - system calls can be I/O (read/write on streams), process control (kill, wait, stop), etc.

Scheduler vs dispatcher

- Scheduling and dispatching are usually performed within the same routine. They prevent a single process from monopolizing processor time
- **Scheduler** decides the next process executed by CPU (i.e. the order/sequence of processes)
- **Dispatcher** handles the **process switch**, that is switches the processor from one process to another. It restores the context for the process (program counter etc.)



(a) State transition diagram



(a) Queuing diagram

Context switch VS mode switch

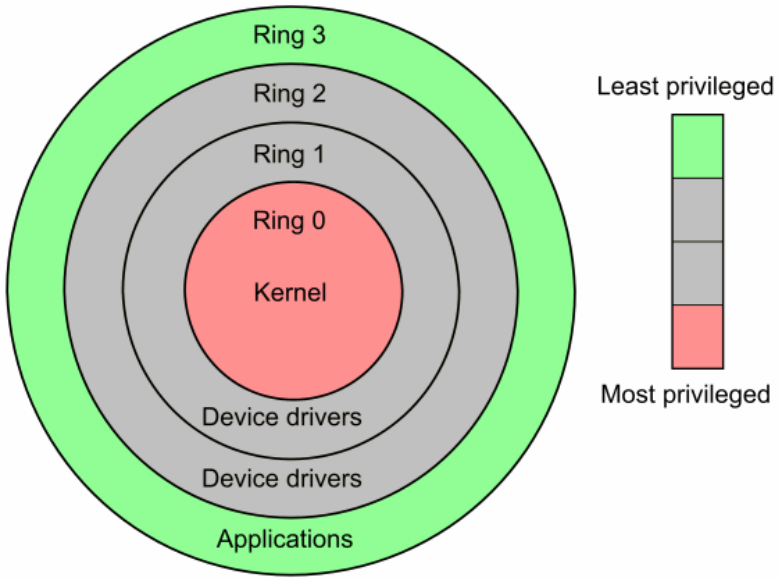
Context switch is the mechanism for in multitasking operating systems we have the illusion of concurrency (multiple processes scheduled on a single CPU)

Context switch VS mode switch/transition

Mode switch/transition

When a transition between user mode and kernel mode is required in an OS, a context switch is not necessary : a **mode transition** is not by itself a context switch.

- However, depending on the operating system, a context switch may also take place.



Modes

- Kernel mode : privileged, has access to all memory locations and system resources
- User mode : initial status of all programs

System calls

- A program (user mode) can run portions of kernel code (e.g. process creation, I/O) via **system calls**, that is a request by an active process for a service performed by the kernel.
- System calls cause rather a **mode switch/transition** than a context switch : because they do not change the current process, but cause the CPU to shift to kernel mode.

Hardware Interrupts (HWI)

HWI are used by physical devices to communicate with the OS about asynchronous events, for example :

- data incoming from external peripheral, device or network
- disk controller signalling its read/writes
- pressing keys on keyboard, moving mouse
- /proc/interrupts

HWI are associated with an **Interrupt Request (IRQ)**

- referenced by an interrupt number (mapped to the hardware, so the OS can monitor which device created the interrupt and when it occurred)

IRQ evoke **Interrupt Service Routines** or **Interrupt Handlers (ISR)**

- ISR are basically routines executed in response to an interrupt
- When an IRQ signal is sent by a device to the processor, the processor completes its current instruction, then stops the execution of the current routine, executes the ISR

↦ /proc/interrupts

```
icub@eva:/proc$ cat interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU
  0:         46          0          0          0          0          0          0
  1:         30          0        152        715          0          0          0
  8:          1          0          0          0          0          0          0
  9:         65          0          0          0          0          0          0
 12:        168          0          0          0          0          0          0
 16:         93          0          0          0          0          0          0
 17:        826          0          0          0          0          0          0
 19:          0          0          0          0          0          0          0
 23:        182      15418          0          0          0          0          0
 41:      17553          0          0          0          0          0          0
 42:      15860          0      44998          0          0          0          0
 43:         14          0          0          0          0          0          0
 44:        521          0          0          7          0          0          0
 45:      44921          0          0          0          0          0          0
NMI:         37         27         28         22          5          5          5
LOC:      28984      23743      22507      32132       9337      12547      10903       111
SPU:          0          0          0          0          0          0          0
PMI:         37         27         28         22          5          5          5
IWI:          0          0          0          0          0          0          0
RES:      64412      86541      83074      72329      18456      20623      20129       222
CAL:        468        490        502        510        486        536        484         5
TLB:        916       1894       1530       1210        609        676        524        95
TRM:          0          0          0          0          0          0          0
THR:          0          0          0          0          0          0          0
MCE:          0          0          0          0          0          0          0
MCP:          4          4          4          4          4          4          4
ERR:          0
MIS:          0
```

↦ **/proc/interrupts**

```
icub@eva:/proc$ cat interrupts
           CPU0       ..       CPU7
  0:         46       ..          0   IO-APIC-edge      timer
  1:         30       ..          0   IO-APIC-edge      i8042
  8:          1       ..          0   IO-APIC-edge      rtc0
  9:         65       ..          0   IO-APIC-fasteoi   acpi
 12:        168       ..          0   IO-APIC-edge      i8042
 16:         93       ..          0   IO-APIC-fasteoi   ehci_hcd:usb1, nouveau
 17:        826       ..          0   IO-APIC-fasteoi   ath9k
 19:          0       ..          0   IO-APIC-fasteoi   xhci_hcd:usb3
 23:        182       ..          0   IO-APIC-fasteoi   ehci_hcd:usb2
 41:      17553       ..          0   PCI-MSI-edge      ahci
 42:      15860       ..          0   PCI-MSI-edge      i915
 43:         14       ..          0   PCI-MSI-edge      mei
 44:        521       ..          0   PCI-MSI-edge      snd_hda_intel
 45:      44921       ..          0   PCI-MSI-edge      eth0
NMI:         37       ..          6   Non-maskable interrupts
LOC:      28984       ..      11185   Local timer interrupts
SPU:          0       ..          0   Spurious interrupts
PMI:         37       ..          6   Performance monitoring interrupts
IWI:          0       ..          0   IRQ work interrupts
RES:      64412       ..      22284   Rescheduling interrupts
CAL:        468       ..        514   Function call interrupts
TLB:        916       ..        955   TLB shootdowns
TRM:          0       ..          0   Thermal event interrupts
THR:          0       ..          0   Threshold APIC interrupts
MCE:          0       ..          0   Machine check exceptions
```

Software Interrupts (SWI)

SWI are the mechanism by which privileged processes can be called by unprivileged ones :

- they are typically used to implement OS calls
- they are called to differentiate privilege levels between processes

Calling a SWI instruction from any mode but supervisor produces :

- a suspension of the currently running process
- a **mode switch** to the supervisor state
- a jump to the SWI interrupt vector

SWI can be generated by

- special instructions throwing interrupts
- exceptions or traps (e.g. divide-by-zero)

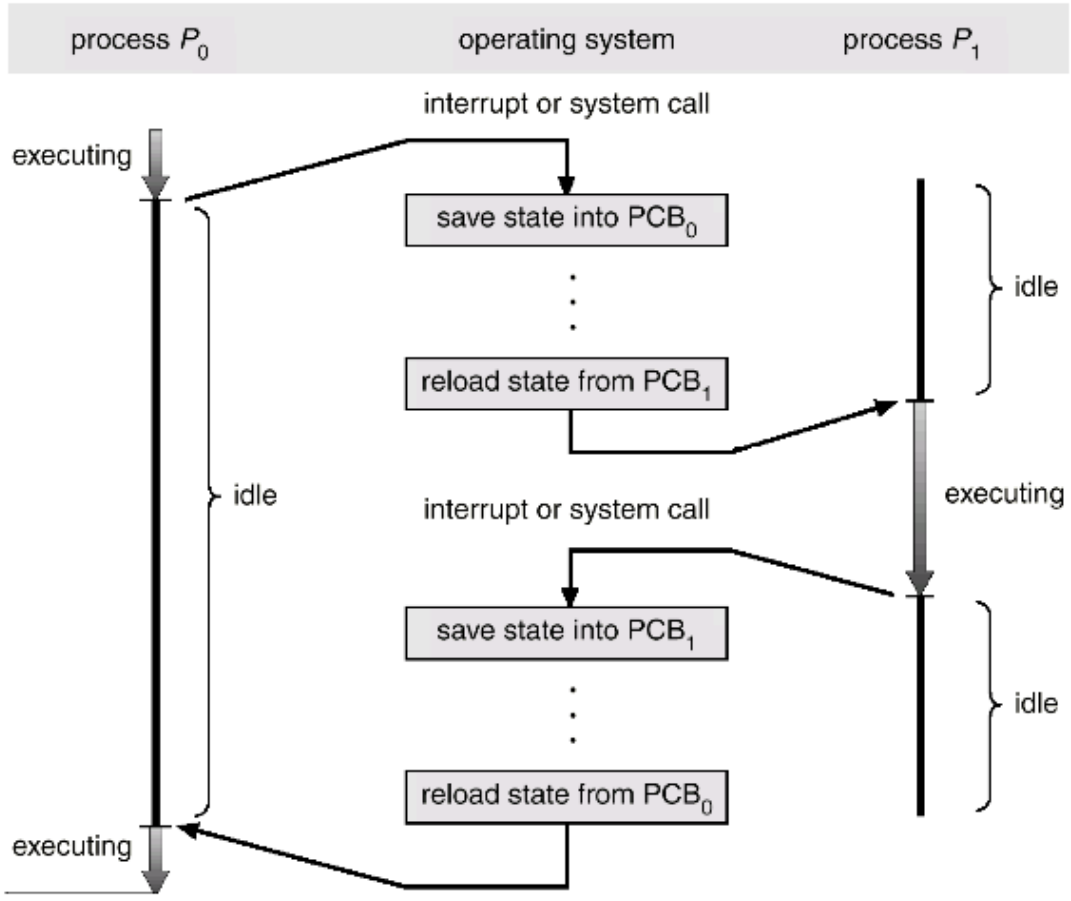
Process Control Block (PCB) or *Task Controlling Struct* or *Task Struct* :

- Chaque processus Linux est représenté par un bloc de contrôle (structure task_struct définie dans linux/sched.h).
- Ce bloc est alloué dynamiquement par le système au moment de la création du processus.
- Il contient tous les attributs permettant de qualifier et de gérer le processus.
- Chaque bloc de contrôle est accessible depuis une table de processus, chaque entrée de la table étant un pointeur vers un bloc de contrôle.

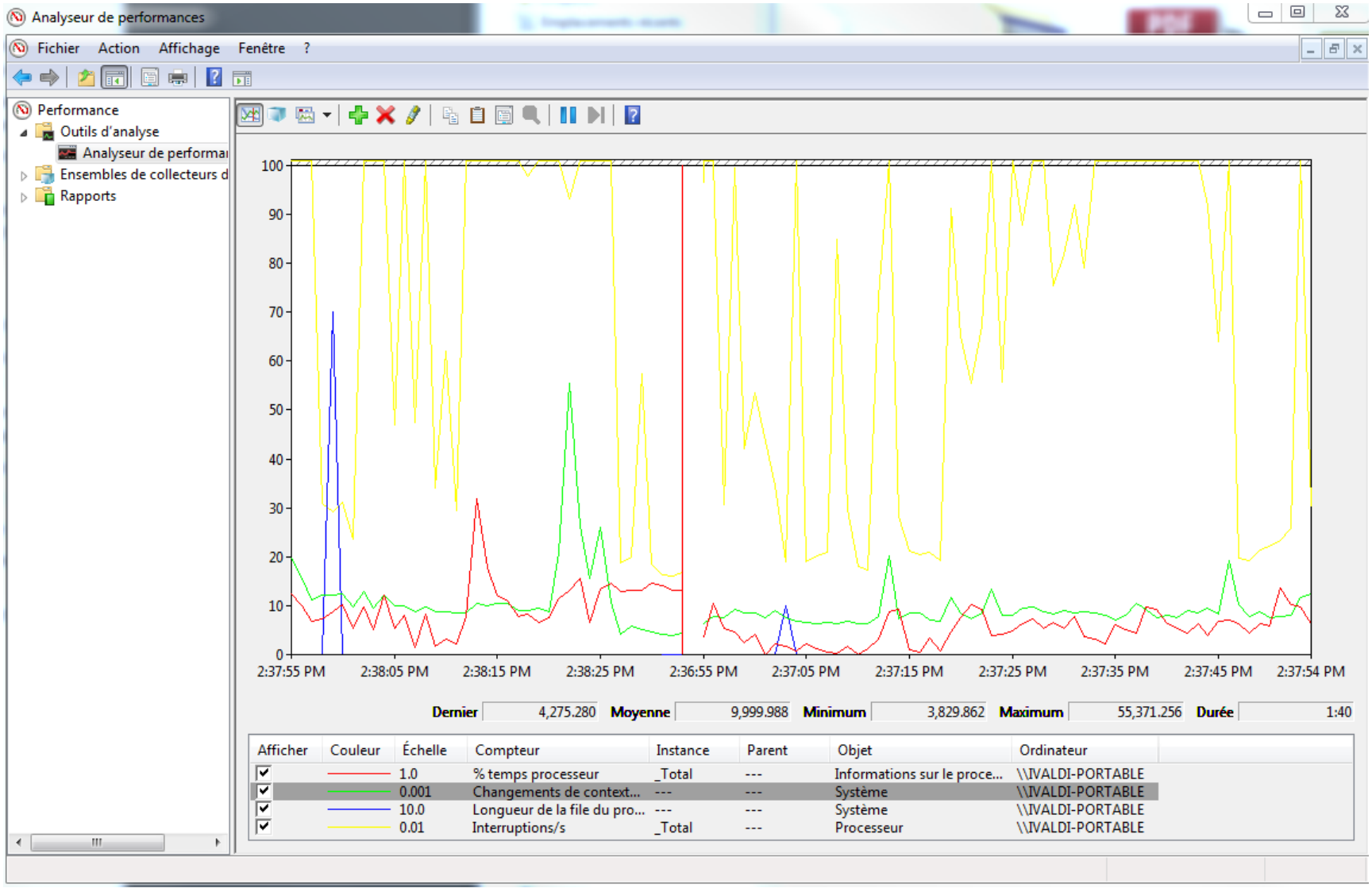
Informations contenues dans le PCB :

- l'état du processus (prêt/élu, bloqué, zombie, arrêté) et les informations d'ordonnancement (pointer to the next PCB - to be scheduled) ;
- les différents identifiants déjà mentionnés : PID, UID, GID, PPID ainsi que l'identifiant du dernier enfant créé et de certains processus frères (cadet le plus jeune et ainé le plus jeune) ;
- les fichiers ouverts par le processus ;
- le terminal attaché au processus ;
- le répertoire courant du processus ;
- le contexte mémoire du processus (address space) ;
- registers, program counter ;
- la partie du contexte processeur pour l'exécution en mode utilisateur ;
- les temps d'exécutions du processus en modes utilisateur et superviseur ainsi que ceux de ses enfant ;
- les outils de communication utilisés par le processus parmi lesquels les sémaphores et les **signaux** ;
- les **signaux** reçus par le processus.

↦ process switching and PCB



↦ process switching, interrupts and performances



(just moving the mouse and browsing web pages)

Signaux ?

Définition

- Les signaux sont un mécanisme de communication inter-processus.
- Les signaux sont envoyées à un ou plusieurs processus. Un signal est en général associé à un événement survenu au niveau du système.
- Le message envoyé est un entier qui ne comporte donc pas d'informations propres si ce n'est une correspondance avec l'événement rencontré par le noyau.
- La prise en compte du signal par le processus oblige celui-ci à exécuter une fonction de gestion du signal appelé **signal handler**.

- Le noyau Linux admet 64 signaux différents,identifiés par un nombre et décrit par un nom préfixé par la constant SIG.
- Seul le signal 0 ne porte pas de nom (signal NULL). Les signaux 1 à 31 correspondent aux signaux classiques tandis que les signaux 32 à 63 sont spécifiquement dédiés à la programmation temps-réel. Parmi ces signaux certains ne sont pas définis par la norme POSIX et sont propres à l'OS.

kill

- La commande `kill` permet d'envoyer des signaux à un processus
- Par défaut cette commande force la terminaison d'un processus
- Le processus à terminer est passé comme argument à la commande `kill` au travers de son PID.

```
icub@icubTest:~$ kill -l
 1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL  5) SIGTRAP
 6) SIGABRT  7) SIGBUS  8) SIGFPE  9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Signaux les plus utiles

number	name	semantics
1	SIGHUP	terminal closed (old) / config. file has changed (new)
2	SIGINT	aborted by user (Ctrl-C)
8	SIGFPE	floating point exception (ex : divide by 0)
9	SIGKILL	terminate process immediately (can't be caught)
11	SIGSEGV	segmentation fault
13	SIGPIPE	write to closed pipe
14	SIGALRM	alarm triggered
15	SIGTERM	terminate process
17	SIGCHLD	child terminated (ignored by default)
18	SIGCONT	resume stopped process (fg)
19	SIGSTOP	stop process (can't be caught)
20	SIGTSTP	stop process (Ctrl-Z)

Le PCB contient plusieurs champs lui permettant de gérer les signaux :

- Le champ `signal`, de type `sigset_t`, qui stocke les signaux envoyés au processus. Cette structure contient deux entiers sur 32 bits, chaque bit représentant un signal. Une valeur 0 indique que le signal correspondant n'a pas été reçu tandis que la valeur 1 indique que le signal a été reçu.
- le champ `blocked`, de type `sigset_t`, qui stocke les signaux bloqués, i.e. les signaux dont la prise en compte est retardée.
- le champ `sigpending` est un drapeau qui indique s'il existe au moins un signal bloqué en attente.
- le champ `gsig` qui est un pointeur vers une structure de type `signal_struct` et qui contient notamment pour chaque signal la définition de l'action qui lui est associée.

Origine des signaux

- envoi par un autre processus par l'intermédiaire d'un appel à `kill` ;
- envoi par le gestionnaire d'exception qui ayant détecté une trappe à l'exécution du processus positionne un signal pour indiquer l'erreur détectée (par exemple, le gestionnaire d'exception `divide_error()` peut positionner le signal SIGFPE).

Positionnement d'un signal

- Lors de l'envoi d'un signal, le noyau exécute la routine noyau `send_sig_info` ;
- Cette routine positionne à 1 le bit correspondant au signal reçu dans le champ `signal` du PCB du processus destinataire.

Prise en compte

- La prise en compte du signal par le processus s'effectue lorsqu'il s'apprête à quitter le mode superviseur noyau pour repasser en mode utilisateur.
- Cette prise en compte est réalisée par la routine noyau `do_signal()` qui traite chacun des signaux pendants du processus.
- Trois types d'actions sont possibles :
  - Ignorer le signal
  - Exécuter l'action par défaut
  - Exécuter une fonction spécifique définie par le programmeur.

Ignorer le signal

- si le signal est ignoré, aucune action n'est entreprise au moment de sa prise en compte ;
- le signal SIGCHLD (terminaison d'un enfant) échappe à cette règle afin que le père puisse mettre à jour son PCB.

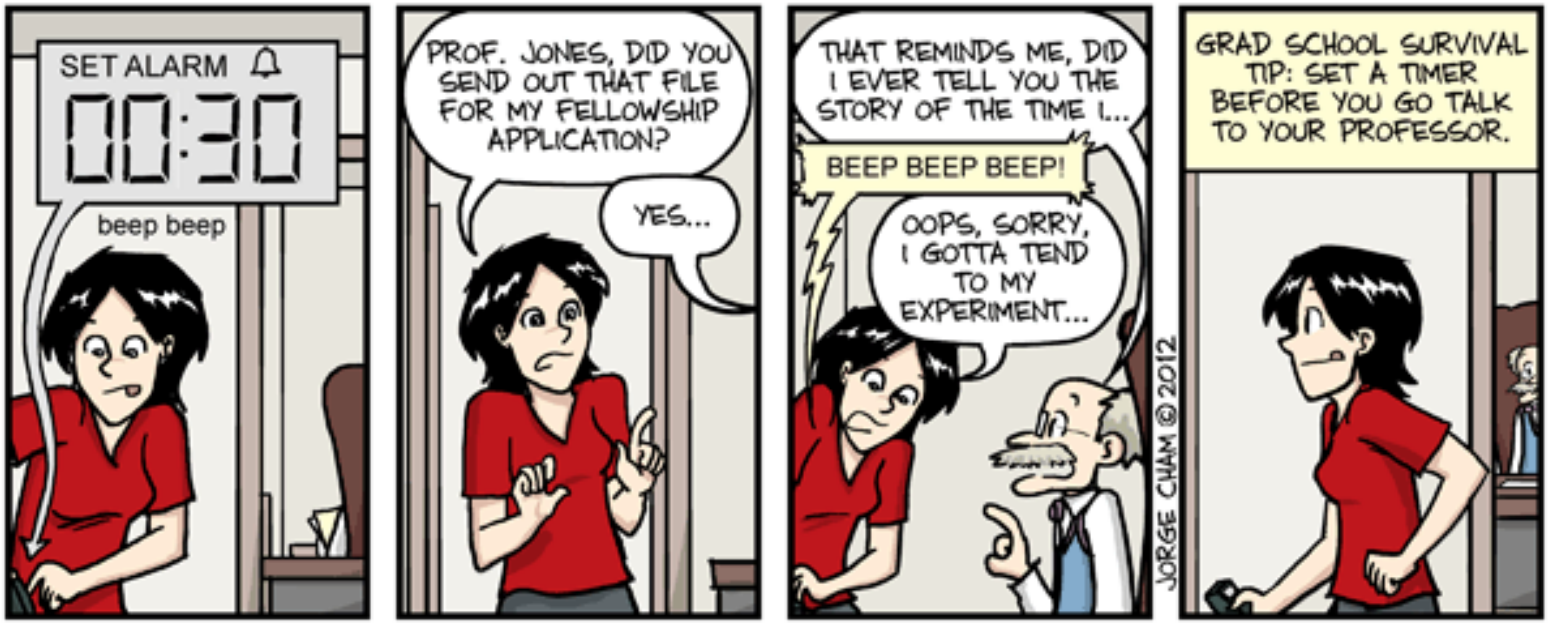
Exécuter l'action par défaut : 5 actions possibles

- term : Abandon du processus (c'est notamment le cas de SIGINT et SIGKILL) ;
- core : Abandon du processus et création d'un fichier **core** contenant son contexte d'exécution exploitable pour le déboggage (c'est notamment le cas de SIGFPE et SIGSEGV) ;
- ign : Signal ignoré (cf. le traitement de SIGCHLD) ;
- stop : Processus stoppé (SIGSTOP, SIGSTP ...) ;
- cont : Processus redémarré (SIGCONT).

Exécution d'une fonction spécifique

Tout processus peut mettre en oeuvre un traitement spécifique pour un signal, défini par une fonction écrite par le programmeur.

Programmation des signaux



Envoi d'un signal

- L'envoi d'un signal à un processus se fait avec la fonction kill () définie dans signal .h et dont le prototype est : **int** kill (pid_t pid , **int** num_sig); .
- L'interprétation du résultat diffère en fonction de la valeur pid :
  - pid > 0 : le signal num_sig est envoyé au processus d'identifiant pid ;
  - pid $= 0$: le signal num_sig est envoyé à tous les processus du groupe de processus auquel appartient le processus appelant ;
    - Rappel : après un fork () le processus enfant appartient au même groupe que son parent.
  - pid < 0 et pid $\neq$ -1 : le signal num_sig est envoyé à tous les processus du groupe de processus auquel appartient le processus d'identifiant abs(pid) ;
  - pid $= -1$: le signal num_sig est envoyé à tous les processus du système sauf le processus 1 et le processus appelant.
- Process must have the permission to send the signal (under Linux : have the `CAP_KILL` capability), or the sending process must have the same user ID of the target process
- Trois cas d'échecs sont possibles (perror) :
  - EINVAL : An invalid signal was specified.
  - EPERM : The process does not have permission to send the signal to any of the target processes.
  - ESRCH : The pid or process group does not exist. Note that an existing process might be a zombie, a process which already committed termination, but has not yet been waited for.
- Il existe une version shell de kill .

Cas particuliers

- **The signal 9 (SIGKILL) can never be blocked, ignored or have a handler installed for it. It always cause the process to immediately terminate.**
- The only signals that can be sent to process ID 1, the `init` process, are those for which `init` has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.
- POSIX.1-2001 requires that if a process sends a signal to itself, and the sending thread does not have the signal blocked, at least one unblocked signal must be delivered to the sending thread before the `kill()` returns.

Signaux bloqués

- Un signal peut être bloqué : il peut alors bien être reçu par le processus, mais sa prise en compte est repoussée à plus tard.
- Les signaux ainsi reçus mais en attente (*pending*) peuvent être lus par la fonction `sigpending` .
- Ces signaux en attente sont immédiatement pris en compte dès que le signal est débloqué

La fonction `sigprocmask()`

La fonction `int sigprocmask(int op, const sigset_t *nouv, sigset_t *anc);` définie dans `signal.h` permet de manipuler le masque de signaux du processus, qui détermine quels signaux sont bloqués.
Opération `op` :

- SIG_SETMASK : affectation du nouveau masque `nouv` ;
- SIG_BLOCK : union des deux ensembles `nouv` et `anc` ;
- SIG_UNBLOCK : "soustraction" `anc - nouv` .

Renvoie 0 en cas de succès et -1 sinon.

Manipulation du vecteur de signaux

Un ensemble de signaux du type `sigset_t` est manipulable grâce aux fonctions suivantes :

- **int** sigemptyset(sigset_t *ens); /* raz */;
- **int** sigfillset (sigset_t *ens) /* ens = 1,2,..., NSIG */;
- **int** sigaddset (sigset_t *ens, **int** sig) /* ens = ens + sig */;
- **int** sigdelset (sigset_t *ens, **int** sig) /* ens = ens − sig */ .

Ces fonctions retournent -1 en cas d'échec et 0 sinon.
int sigismember(sigset_t *ens, **int** sig); retourne vrai si le signal appartient à l'ensemble.

fonction sigpending

int sigpending(sigset_t *set); définie dans signal .h

- Remplit le `sigset_t *set` avec l'ensemble des signaux en attente (*pending*)
- Il s'agit donc des signaux qui ont été reçus mais sont actuellement bloqués.

Modification de l'action associée à un processus

- On peut modifier la réaction d'un processus à un signal par l'appel à `sigaction()` qui est la version normalisée POSIX de l'appel `signal` dont l'emploi peut s'avérer peut portable.
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);` est défini dans `signal.h`.
- Les différents paramètres ont la signification suivante :
  - `signum` : numéro de signal pour lequel le comportement est modifié ;
  - `const struct sigaction *act` : définition de la nouvelle action associée au signal ;
  - `struct sigaction *oldact` : sauvegarde de l'ancienne action associée au signal (NULL si on ne souhaite rien sauvegarder).

sigaction

```
struct sigaction
{
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void); // obsolete, non used
};
```

Structure **struct** sigaction

La structure sigaction contient les champs suivants :

- **int** sa_flags : options de comportement du *handler*. 0 pour les options par défaut.
- **void** (* sa_handler)() : pointeur sur la fonction à exécuter pour la gestion du signal ;
  - On peut aussi utiliser les constantes symboliques SIG_DFL pour spécifier l'action par défaut et SIG_IGN pour ignorer le signal.
  - Alternativement, si sa_flags contient SA_SIGINFO , **void** (*sa_sigaction)(**int** , siginfo_t *, **void** *); est utilisé et permet au gestionnaire de signal de recevoir des informations supplémentaires.
- sigset_t sa_mask : ensemble des signaux à bloquer pendant l'exécution de la fonction pointée par sa_handler ;

siginfo_t

```
siginfo_t
{
    int      si_signo;    /* Signal number */
    int      si_errno;    /* An errno value */
    int      si_code;     /* Signal code */
    int      si_trapno;   /* Trap number that caused
                             hardware-generated signal
                             (unused on most architectures) */
    pid_t    si_pid;      /* Sending process ID */
    uid_t    si_uid;      /* Real user ID of sending process */
    int      si_status;   /* Exit value or signal */
    clock_t  si_utime;    /* User time consumed */
    clock_t  si_stime;    /* System time consumed */
    sigval_t si_value;    /* Signal value */
    int      si_int;      /* POSIX.1b signal */
    void    *si_ptr;      /* POSIX.1b signal */
    int      si_overrun;  /* Timer overrun count; POSIX.1b timers */
    int      si_timerid;  /* Timer ID; POSIX.1b timers */
    void    *si_addr;     /* Memory location which caused fault */
    long     si_band;     /* Band event (was int in
                             glibc 2.3.2 and earlier) */
    int      si_fd;       /* File descriptor */
    short    si_addr_lsb; /* Least significant bit of address
                             (since kernel 2.6.32) */
}
```

pause

int pause() définie dans unistd.h

- Attente de délivrance d'un signal
- Met le processus en sommeil jusqu'à ce qu'un signal ait été recu et traité (reprend après le gestionnaire de signal)
- Les signaux ignorés ne mettent pas fin à une pause()

alarm

unsigned int alarm(**unsigned int** nb_sec) définie dans unistd.h

- Arme une temporisation de nb_sec secondes, qui délivre le signal SIGALRM à son issue.
- alarm(0) annule l'alarme.
- Permet de mettre en place des *timeout*, par exemple pour clore un processus si un temps d'attente (ouverture d'un fichier, écriture dans un pipe, etc.) est jugé trop long.

Recap

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

int kill(pid_t pid, int sig);
int sigwait (const sigset_t *set, int *sig);
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldac
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
int sigpending(sigset_t *set);
```

note :

```
/* A sigset_t has a bit for each signal.  */
# define _SIGSET_NWORDS (1024 / (8 * sizeof (unsigned long int)))
typedef struct
{
  unsigned long int __val[_SIGSET_NWORDS];
} __sigset_t;
```

Example

*** block_ctrlc.c ***

A simple program that blocks SIGINT (the signal generated by pressing CTRL+C) to allow a certain computation to continue without being interrupted.
Temporary blocking of signals is used to prevent interrupts during critical parts of the code (e.g. computing a priority control signals for the safety of the robot) : if such signals arrive, they are queued and delivered later, when unblocked in the code.

**** block_ctrlc.c ****

```c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_INT 5000

int main(int argc, char *argv[])
{
        sigset_t signals1, signals2;
        unsigned int i = 0;

        sigemptyset(&signals1);        //empty signals1
        sigaddset(&signals1,SIGINT);   //select SIGINT
        sigprocmask(SIG_SETMASK,&signals1,0); //block the signals selected by
             signals1

        printf("PID %d ; starting to count - cannot block me with CTRL+C\n",
             getpid());
        while(i < MAX_INT)
        {
                printf("\r%d going to %d, can't stop me   ",MAX_INT,i);
                usleep(1000); //wait 1 ms
                i = i+1;
        }

        sigpending(&signals2);      //retrieve pending signals
        if(sigismember(&signals2,SIGINT))
                printf("\nSIGINT is pending : 'CTRL C' was pressed\n");

        sigemptyset(&signals1);     //empty signals1
        sigprocmask(SIG_SETMASK,&signals1,0); //unblock SIGINT

        printf("\nSIGINT unblocked and not used.\n");
        return 0;
}
```

Output when pressing CTRL+C during execution :

```
$ ./block
Starting to count - cannot block me with CTRL+C
5000 going to 4999, can't stop me
SIGINT is pending : 'CTRL C' was pressed
```

Output when not pressing CTRL+C during execution :

```
$ ./block
Starting to count - cannot block me with CTRL+C
5000 going to 4999, can't stop me
SIGINT unblocked and not used.
```

Exercice

*** new_handler.c ***

A simple program that define a specific handler for a specified signal. It catches segmentation faults (signal SIGSEGV), and when too many segfaults are generated, send a SIGKILL signal to terminate the program.

**** new_handler.c ****

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

void segv(int sig) // Handler associated to a segfault signal
{
        static int segf_counter = 0;
        segf_counter += 1;
        printf("Segmentation fault n. %d (signal %d)\n",segf_counter,sig);
        if(segf_counter > 5)
        {
                kill(getpid(),SIGKILL); //kill myself - too many segfaults
                exit(0);
        }
}

int main(int argc, char *argv[])
{
        struct sigaction action;
        unsigned int k;
        int buff[5];

        sigemptyset(&action.sa_mask); // empty signal mask
        action.sa_flags = 0;
        action.sa_handler = segv;      // set the handler function
        sigaction(SIGSEGV,&action,0); // catch SIGSEGV

        for(k=0;;k++)      // this loop is a segfault producer!!
                        buff[k]=k;

        printf("Ouch!");
        return 0;
}
```

Output (the program terminates alone when too many segfaults are catched) :

```
$ ./handler
Segmentation fault n. 1 (signal 11)
Segmentation fault n. 2 (signal 11)
Segmentation fault n. 3 (signal 11)
Segmentation fault n. 4 (signal 11)
Segmentation fault n. 5 (signal 11)
Segmentation fault n. 6 (signal 11)
Killed
```

Exercice

*** sigaction.c ***

A program that defines a specific handler for SIGTERM using `sa_sigaction` . It prints the PID and UID of the process sending the signals.

**** sigaction.c ****

```c
// from linuxprogrammingblog.com

/* Example of using sigaction() to setup a signal handler with 3 arguments
 * including siginfo_t.
 */
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
        printf ("Sending PID: %ld, UID: %ld\n",
                        (long)siginfo->si_pid, (long)siginfo->si_uid);
}

int main (int argc, char *argv[])
{
        struct sigaction act;

        memset (&act, '\0', sizeof(act));

        /* Use the sa_sigaction field because the handles has two additional
             parameters */
        act.sa_sigaction = &hdl;

        /* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field,
             not sa_handler. */
        act.sa_flags = SA_SIGINFO;

        if (sigaction(SIGTERM, &act, NULL) < 0) {
                perror ("sigaction");
                return 1;
        }
        printf("I have PID %d and am waiting for SIGTERM\n", getpid());

```

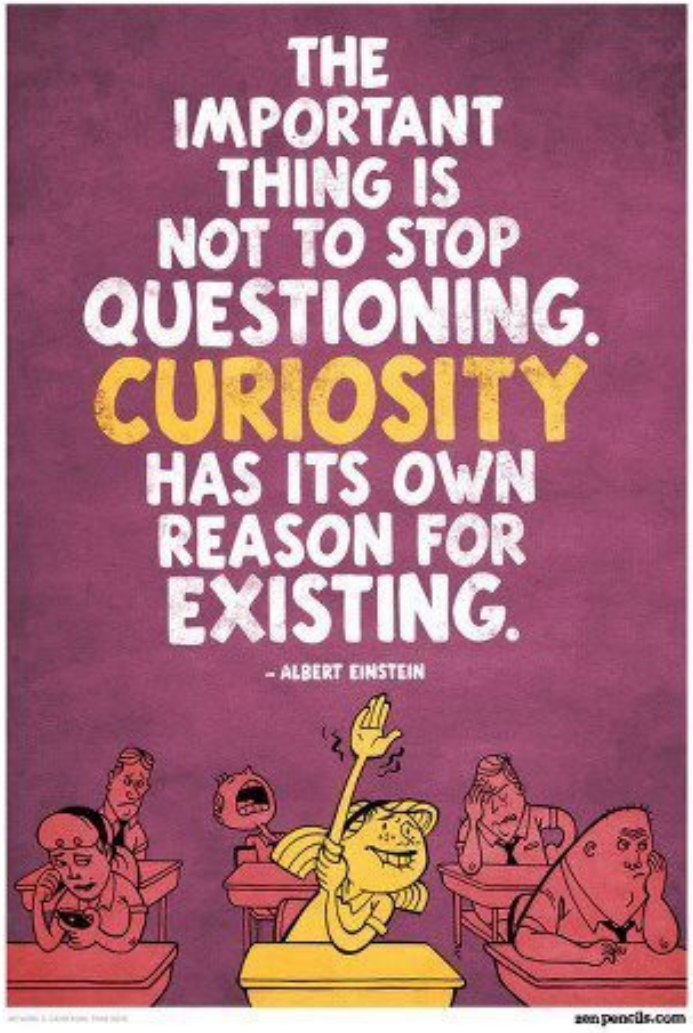
Output :

```
$ ./sigaction
I have PID 29984 and am waiting for SIGTERM
```

Sending signal : `kill 29984`
Result :

```
$ ./sigaction
I have PID 29984 and am waiting for SIGTERM
Sending PID: 29875, UID: 1000
```

Questions ?


THE IMPORTANT THING IS NOT TO STOP QUESTIONING. CURIOSITY HAS ITS OWN REASON FOR EXISTING.
– ALBERT EINSTEIN
zenpencils.com