



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE



Module Système II Éléments de base & Gestion des signaux

Support de cours 1

3^{ème} année INSA CVL
Département STI
2017-2018

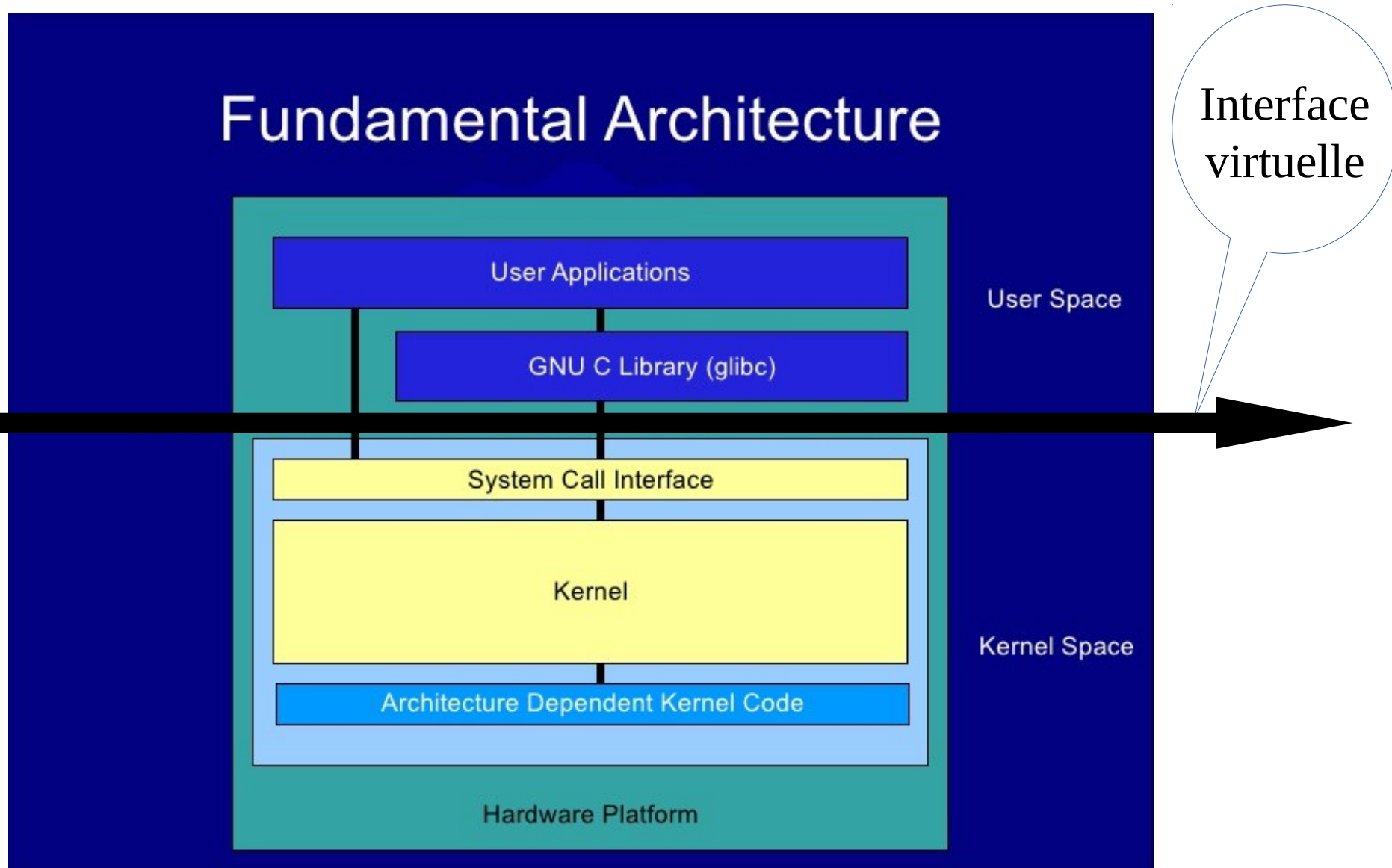
Definition Operating System (OS)

Un système d'exploitation, ou Operating System (OS), est un logiciel qui, dans un appareil électronique, pilote les dispositifs matériels et reçoit des instructions de l'utilisateur ou d'autres logiciels (ou applications).

Son but est d'offrir aux programmeurs et aux utilisateurs une interface virtuelle leur permettant de faire abstraction des problèmes de gestion des éléments matériels (Processeurs, Horloge, système de fichiers (HD,SD, CD,DVD,...), allocation mémoire RAM (mémoire virtuelle, swap,...), bus, ...)

De plus, l'OS comporte des utilitaires de gestion du système (exemple : interpréteur de commandes, ..., tournant dans le user space)

Architecture fondamentale OS Unix



Architecture fondamentale OS Unix

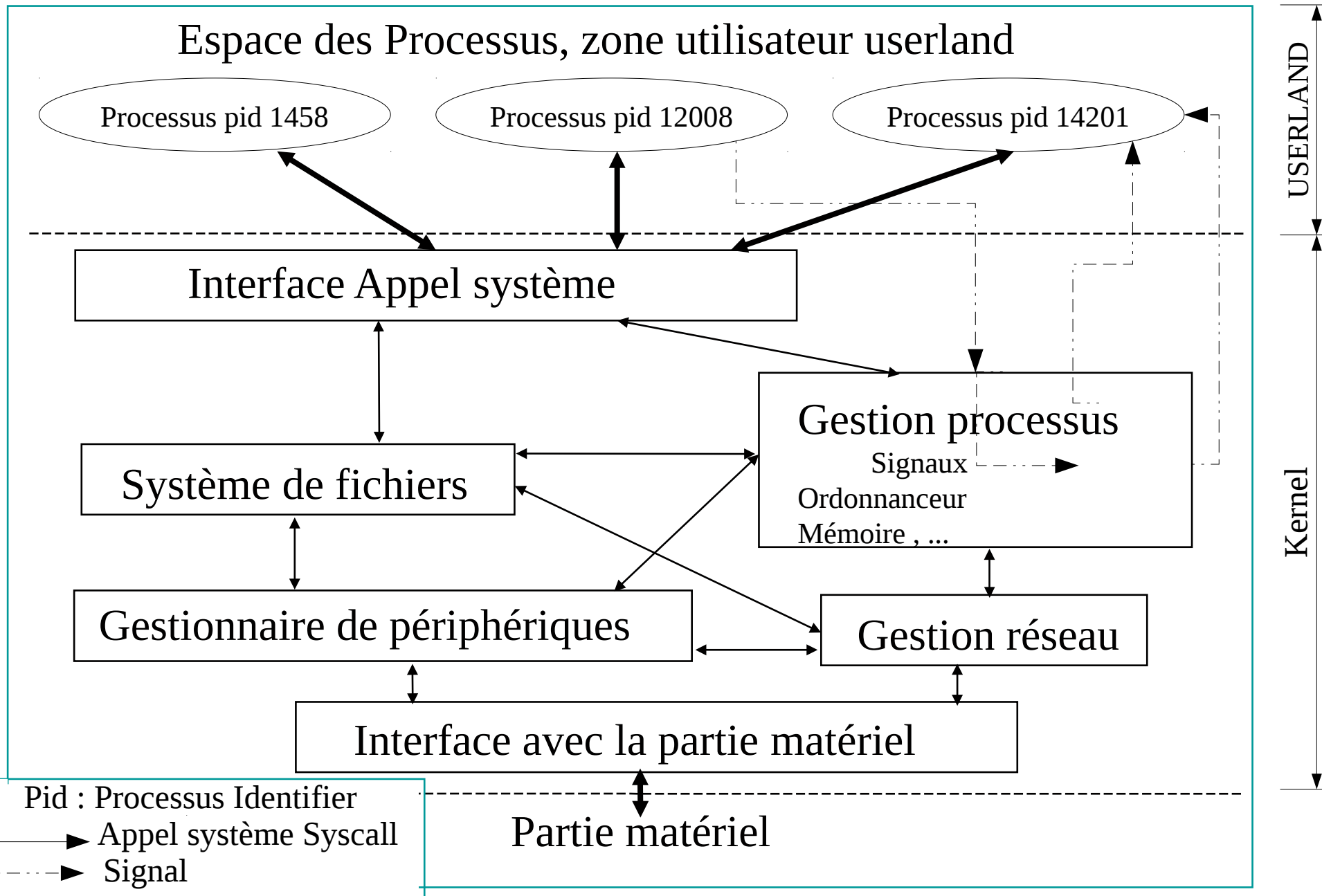
Dialogue entre le « user space »(espace utilisateur) et le « kernel space » (espace noyau) .

Ce dialogue s'effectue essentiellement par deux techniques :

- Les appels système («system call» ou «syscall» chapitre 2 du manuel Linux) :
Un appel système est une fonction fournie par le noyau d'un système d'exploitation et utilisée par les programmes s'exécutant dans l'espace utilisateur.
- Les signaux :
Les signaux sont des interruptions logicielles, ils interrompent le flot normal d'un processus¹ mais ils ne sont pas traités de façon synchrone comme les interruptions matérielles (Le processus en tient compte lorsqu'il est à l'état «running» et au temps d'ordonnancement près, signaux bloqués...).

¹ Entité active en mémoire possédant du code à exécuter et des données. Cette entité possède sa propre zone mémoire virtuelle.

Communication : Kernel \leftrightarrow Userland



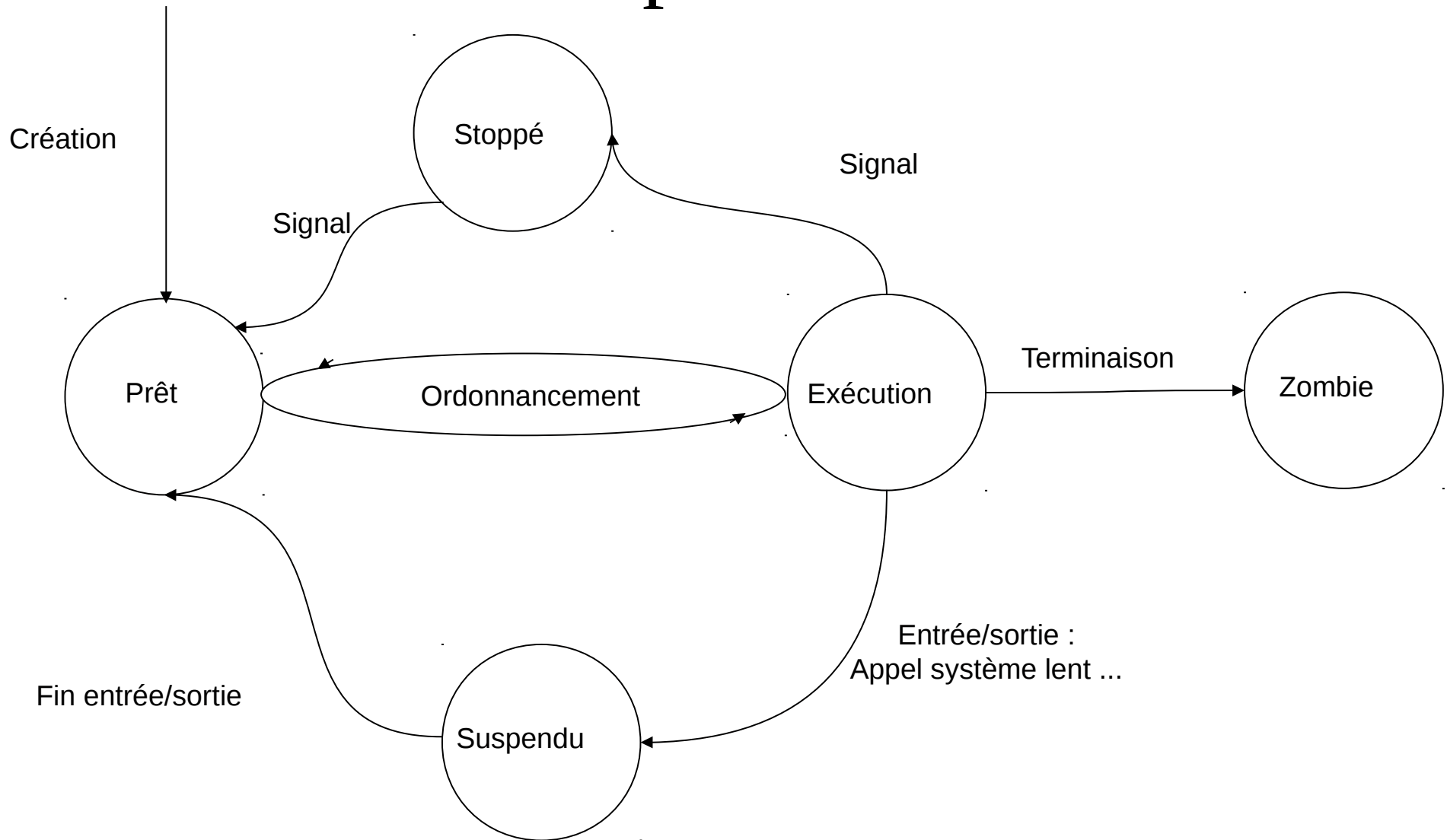
La bibliothèque C (glibc)

Dans le « user space », il existe une couche appelée « GNU C Library » qui vient compléter les appels système.

- Cette bibliothèque (Chapître 3 du manuel Linux) vient compléter les appels système du noyau.
Exemple : L'appel système qui permet d'allouer de la mémoire dans le tas (Heap) est « brk() », il est plus aisé et plus portable d'utiliser la fonction « malloc() » de la glibc.
- Les versions 1 à 4 de la « **libc** » étaient destinées aux programmes exécutables au format a.out ; La version 5 pour les exécutables au format « ELF » (Executable and Linkable Format). A partir du noyau 2.0, la bibliothèque a pris le nom de ¹« Glibc » car issue du projet « GNU ».

¹ Elle est parfois appelée abusivement « libc6 »

Etats d'un processus



Histoire d'UNIX

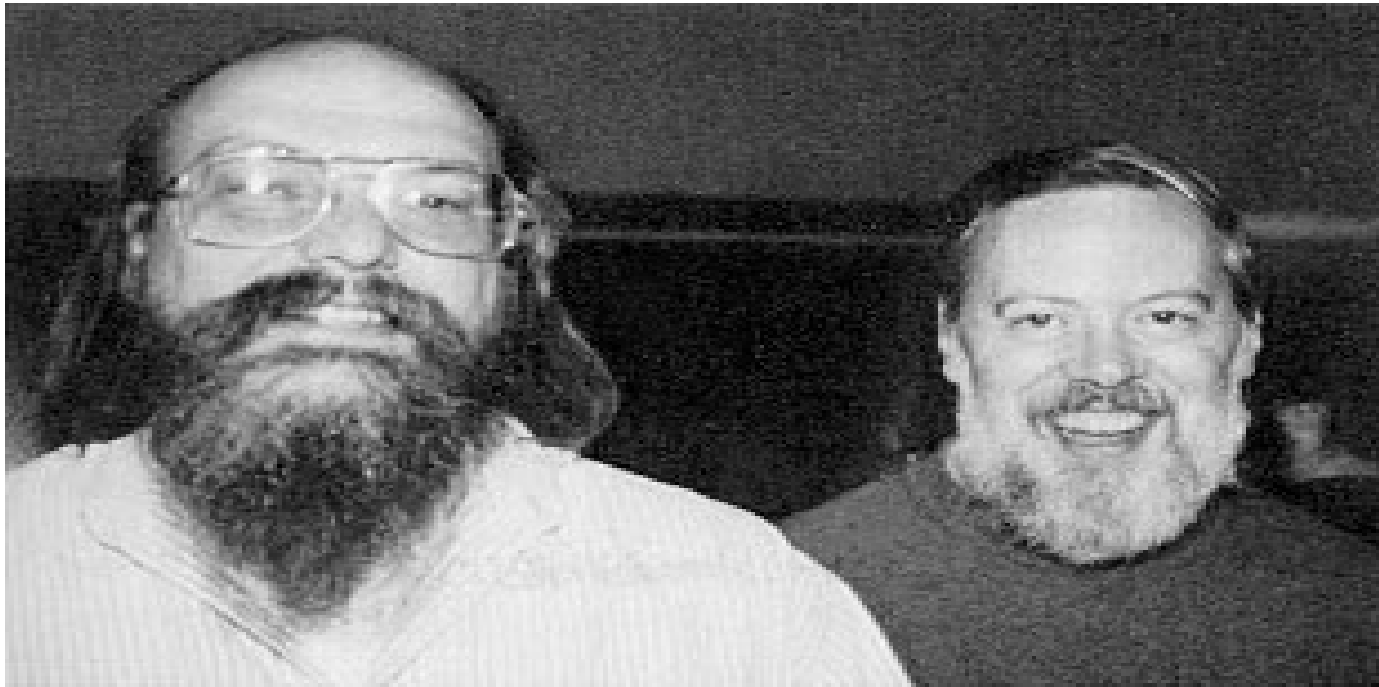
- Dès 1970 trois chercheurs « Ken Thompson » « Denis Ritchie » et « Brian Kernighan » de BELL LABS laboratoire de AT&T « American Telephone and Telegraph » créent un projet appelé UNIX (une seule personne) par opposition à MULTICS projet précédant ambitieux et jamais achevé.
- Première version écrite en C, diffusée en 1973 avec le code source aux universités, rendant les programmes moins dépendants des constructeurs.

Inventeurs d'UNIX

Dennis Ritchie : décédé le 12 octobre 2011 : c'est l'un des fondateurs de l'informatique moderne, il est l'un des créateurs du langage C et des système UNIX.

Ken Thompson : Un des concepteurs du système UNIX ainsi que du langage B prédécesseur du langage C (programmation OS).

Ken Thompson (gauche) et Dennis Ritchie (droite)



Histoire d'UNIX

- Deux branches d'UNIX
(Bell Labs AT&T) et
BSD

1970

1876 : L'Américain **Graham Bell** invente le **téléphone** et fonde la compagnie **Bell Telephone Company**

1975

1977

<http://www.bell-labs.com/history/unix/>

AT&T, *American Telephone & Telegraph* maison mère des laboratoires Bell

1979

1980

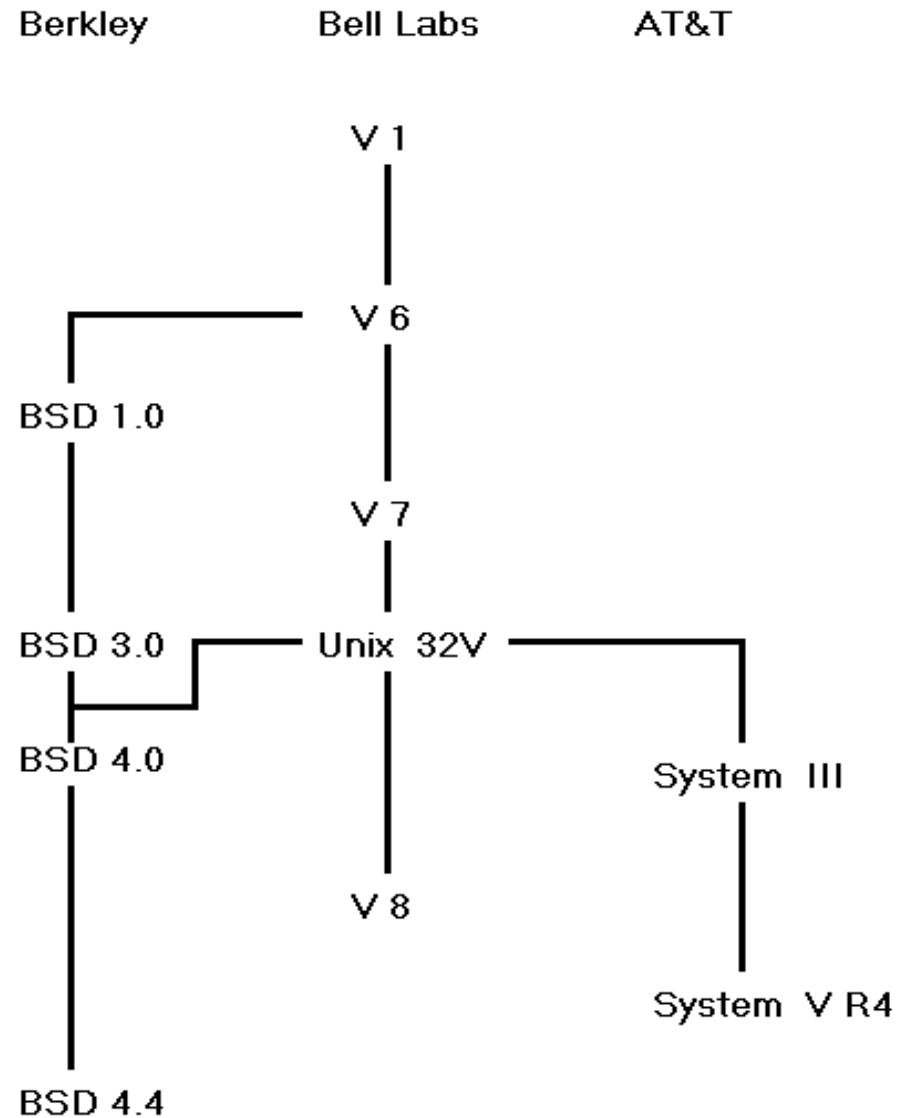
BSD, *Berkeley Software Development*

BSD Berkley Software Distribution
System V R4

1989

AT&T System V release 4
V8 Bell Labs Version 8

1992



Standardisation des OS

En 1988 les développeurs et les constructeurs vont alors se mettre d'accord sur un standard qui spécifie dans dix-sept documents différents les interfaces utilisateurs et les interfaces logicielles.

C'est la naissance de POSIX « Portable Operating System 'for Unix' » norme ¹IEEE 1003 « Institute of Electrical and Electronics Engineers » .
Le 'for UNIX' est donc historique mais concerne d'autres SE.

Ce standard même à l'heure actuelle n'est pas appliqué pour chaque développement et l'on parle de compatibilité POSIX.

La certification POSIX est coûteuse (conclusion les distributions OS libres sont dites compatibles et non certifiées).

¹IEEE Institute of Electrical and Electronics Engineers :
Organisation professionnelle fondée en 1963, regroupant les professionnels des télécommunications, de l'électronique, de l'électricité et de l'informatique.
<http://standards.ieee.org>

Standardisation des OS

Le respect de la normalisation est une garantie de portabilité et de pérennité.

Les standards importants (Sur LINUX « man standards ») :

- Posix.1 "Portable Operating System Interface for Computing Environments". IEEE 1003.1-1990 part 1, ratified by ISO in 1990 (ISO/IEC 9945-1:1990)
- Posix.1b IEEE Std 1003.1b-1993 describing real-time facilities for portable operating systems, ratified by ISO in 1996 (ISO/IEC 9945-1:1996).
- Posix.1c IEEE Std 1003.1c-1995 describing the POSIX threads interfaces.
- System V exemple : SVr4
- BSD exemple : 4.3BSDC
- ANSI (ISO C). Ce n'est pas un standard d'OS mais un standard de langage de programmation. Premier standard du langage C, ratifié par l'ANSI (American National Standards Institute) en 1989 (X3.159-1989). Ratifié par l'ISO en 1990 (ISO / IEC 9899: 1990) parfois appelé ISO C90.

Importance de la standardisation 1/3

Pourquoi les standards sont importants ?

Si l'on indique au compilateur de se conformer à une norme plutôt qu'à une autre cela affectera le comportement à l'exécution de certaines fonctions (démonstration en Tds) ou peut même empêcher la compilation.

Exemple : Soit le programme `display_signal.c` qui affiche le nom des signaux

```
//#include j'ai enlevé les « includes » pour l'intégrer dans la diapo  
int main (void)  
{ int i;  
  for (i = 1; i < NSIG; i ++)  
    fprintf(stdout, "signal %d (%s)\n",i, sys_siglist[i]);}
```

Importance de la standardisation 2/3

Compilation 1 réussie :

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
--> gcc -o display display_signal.c
--> █
```

Compilation 2 échouée :

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
--> gcc -D _XOPEN_SOURCE -o display display_signal.c
display_signal.c: In function 'main':
display_signal.c:10:18: error: 'NSIG' undeclared (first u
    for (i = 1; i < NSIG; i++)
                        ^
display_signal.c:10:18: note: each undeclared identifier
display_signal.c:11:40: error: 'sys_siglist' undeclared
    fprintf(stdout, "signal %d (%s)\n", i, sys_siglist[i]);
                                           ^
```

Dans la deuxième compilation on demande à « gcc » de prendre en compte la constante symbolique « _XOPEN_SOURCE » qui lui impose de compiler conformément aux normes POSIX. Or la constante symbolique « NSIG » et le tableau de caractères « sys_siglist[] » ne sont pas définis dans ces normes d'où échec de la compilation !!!!

Importance de la standardisation 3/3

Fichier Édition Affichage Rechercher Terminal Aide

```
--> gcc -o display display_signal.c -std=gnu11
--> gcc -o display display_signal.c -std=c11
display_signal.c: In function 'main':
display_signal.c:11:40: error: 'sys_siglist' undeclared (first use in this function)
    fprintf(stdout, "signal %d (%s)\n", i, sys_siglist[i]);
                                   ^
display_signal.c:11:40: note: each undeclared identifier is reported only once for ea
--\ ■
```

Par défaut gcc utilise le standard « gnu11 » ; ci-dessus on peut remarquer que la compilation faite avec « gnu11 » ne pose pas de problème, alors qu'avec la norme « ISO c11 » le tableau « sys_siglist[] » est inconnu !!!

Les signaux classiques

Rappels du module système 1

- Un signal peut être considéré comme une interruption (impulsion) du logiciel.
- Le processus le prend en compte immédiatement au temps d'ordonnancement près et à l'état running.
- Le processus qui le reçoit peut
 - L'ignorer.
 - Le « capturer »(!!! mal dit), exécution d'une partie de code appelée gestionnaire de signal.
 - Laisser le comportement par défaut (exemple le signal SIGSEGV termine le processus et crée un fichier image mémoire core)
- La plupart des signaux sont émis par le noyau mais les applicatifs peuvent demander à en émettre (vers eux même, ou vers d'autres processus).

Rappels de 1ère année

- Sous linux les signaux classiques conformes à la norme ANSI C sont au nombre de 32.
- 32 signaux temps réel (normes Posix.1 et Posix.1b) portent le nombre de signaux à 64.
- Chaque signal classique porte un nom symbolique commençant par SIG qui est associé à un numéro (Ex : SIGCONT 18).
- Il est recommandé d'utiliser les noms plutôt que les numéros pour des raisons de compatibilité des systèmes.
- Les fonctions relatives aux signaux sont définies dans le fichier <signal.h>

Constantes symboliques

- NSIG sur certains systèmes _NSIG = nombre de signaux
- SIGRTMIN (signal real time minimum) numéro du premier signal temps réel.
- SIGRTMAX (signal real time maximum) numéro du premier signal temps réel.
- _POSIX_REALTIME_SIGNALS définie dans `<unistd.h>` indique le nombre de signaux temps réel

Émission d'un signal sous C

- `kill`
 - `int kill (p_pid pid, int sig);`
 - 1er argument numéro pid du processus visé
 - 2ème argument numéro du signal.
 - Le signal numéro 0 permet de tester la présence d'un processus.
- `killpg`
 - `int killpg (int pgid, int sig);`
envoie un signal à un groupe de processus.

Attention : « `kill` » est un faux ami. « `kill` » = « envoyer un signal » qui parfois tue mais pas toujours.

Réception des signaux sous C

- Un processus peut demander explicitement au noyau soit d'ignorer un signal, soit d'appliquer le comportement par défaut soit d'installer un « gestionnaire de signal ».

Les signaux classiques

- SIGABRT, SIGIOT
 - Deux noms pour le même signal, utiliser plutôt le deuxième nom qui se trouve dans ANSI C et POSIX (premier System V)
 - Comportement par défaut : terminer le processus et créer un fichier core.
 - Indique la fin anormale d'un processus
 - Peut être ignoré mais attention à la gestion des flux d'entrée sortie (voir utilisation de la fonction abort()).
 - Comportement par défaut : terminaison du processus et création d'un fichier core

Les signaux classiques

- SIGALRM, SIGVTALRM et SIGPROF
 - SIGALRM est envoyé par le noyau à l'expiration d'un délai programmé (fonction alarm())
 - il permet :
 - de définir un délai d'attente pour les appels système bloquants (lecture depuis une socket par exemple).
 - De programmer des temporisations (setitimer())
 - Trois types de temporisations :
 - Fonctionne en temps réel et émet SIGALRM
 - Fonctionne que lorsque le processus s'exécute SIGVTALRM
 - Cumule le temps d'exécution du processus et ceux de ses appels noyaux puis déclenche SIGPROF
 - Comportement par défaut : termine le processus
 - Sleep () utilise SIGALARM donc éviter d'utiliser alarm() avec sleep()

Les signaux classiques

- SIGBUS et SIGSEGV
 - SIGBUS indique une erreur d'alignement sur le bus et SIGSEGV une violation de segmentation (pointeur mal utilisé).
 - SIGBUS indique une adresse mémoire invalide (ex : alignement sur une adresse non multiple de 4 pour un système d'adressage 32 bits).
 - SIGSEGV indique une adresse valide mais ne pointant pas sur un espace d'adressage permis au processus.
 - SIGBUS interrompt par défaut le processus.
 - SIGSEGV interrompt le processus et crée un fichier core
 - Il est préférable de ne pas ignorer ces signaux.

Les signaux classiques

- SIGCHLD et SIGCLD(synonymes CLD system V et CHLD posix.1)
 - SIGCHLD émis vers le père d'un processus dont le fils vient de se terminer, de passer à l'état stoppé ou qui vient de recevoir le signal « continue ».
 - Le père peut ignorer le signal ou invoquer wait() ou waitpid() de manière à récupérer le pid du fils et le code de terminaison.
 - SIGCLD est un synonyme, préférer la première écriture.
 - Comportement par défaut : le processus fils disparaît de la mémoire si SIGCHLD est volontairement ignoré (défaut).
 - Si SIGCHLD est ignoré **explicitement** le processus reste zombie jusqu'à un appel à wait() ou waitpid() par le père.
 - S'il est capturé par un gestionnaire le processus fils reste zombie jusqu'à un appel à wait() ou waitpid() par le père.

Les signaux classiques

- SIGFPE et SIGSTKFLT
 - SIGFPE « Floating Point Exception » : erreur de virgule flottante ou division par 0, si le processeur ne possède pas de coprocesseur et que le processus ne l'émule pas
 - Arrêt du processus plus core.
 - SIGSTKFLT non conforme à Posix n'est pas traité par le noyau Linux.

Les signaux classiques

- SIGHUP
 - Déconnexion du terminal de contrôle du processus
 - Le noyau envoie ce signal au processus leader de session associé au terminal (setpgid()).
 - Envoyé aussi à tous les processus d'un groupe quand le leader s'arrête. Il est suivi de SIGCONT pour remettre en route les processus arrêtés.
 - On utilise souvent cet appel système en destination de démons. Comme ceux-ci n'ont pas de terminaux de contrôle, ils déroutent le signal et relisent les fichiers de configuration (inetd ,named, httpd...).
 - Par défaut SIGHUP arrête le processus, nohup « commande shell » permet de lancer un processus en l'immunisant contre ce signal.

Les signaux classiques

- SIGILL
 - Détection d'une instruction assembleur illégale le signal est déclenché par le noyau recevant une interruption matériel ou une instruction invalide envoyée à l'émulateur de coprocesseur ou un débordement de pile.
 - Arrêt du processus et fichier core
 - On peut dérouter ce signal et adopter l'une des stratégies suivantes :
 - Arrêter le processus par un `exit()` : évite le fichier core
 - Utiliser un saut non local `longjmp()` sur un point défini du programme avec un contexte propre

Les signaux classiques

- SIGINT
 - Envoyé à tous les processus en avant plan par l'appui de touches particulières du clavier (CTRL-C sur PC).
 - Arrête proprement le processus en cours

Les signaux classiques

- SIGIO et SIGPOLL
 - Synonymes SIGIO norme BSD, SIGPOLL vient de Système V
 - Envoyé lorsqu'un descripteur de fichier change d'état et permet la lecture ou l'écriture en général pour des tubes des sockets ou des terminaux.
 - Permet l'utilisation de traitement d'entrée sortie de manière asynchrone.
 - Impossibilité de connaître le descripteur ayant déclenché le signal (voir select() alternative à SIGIO).

Les signaux classiques

- SIGKILL
 - Tout processus recevant ce signal est immédiatement stoppé sauf :
 - Si c'est le processus init.
 - Si le processus est dans l'état zombie.
 - SIGKILL est associé normalement à la valeur 9 (sous shell : `kill -9 xxx`). Lui préférer SIGQUIT ou SIGTERM
 - Un processus stoppé est remis en fonctionnement par SIGKILL puis terminé (attention les flux sont terminés anormalement)
 - **Ce signal ne peut être dérouté (c'est le seul avec SIGSTOP)**

Les signaux classiques

- SIGPIPE
 - Signal envoyé par le noyau lorsqu'un processus tente d'écrire dans un tube sur lequel il n'y a pas de lecteur au bout (Exemple socket dont le correspondant s'est déconnecté).
 - Par défaut arrête le processus

Les signaux classiques

- SIGQUIT
 - Comme SIGINT touches Ctrl-AltGr-\ au clavier (parfois inopérant en environnement graphique serveur X)
 - Différence de résultat entre SIGINT et SIGQUIT : le deuxième engendre un fichier core.

Les signaux classiques

- SIGSTOP, SIGCONT et SIGTSTP
 - SIGSTOP **ne peut être dérouté**, le processus n'est pas arrêté mais il est stoppé.
 - SIGCONT effectue l'opération de remise en route
 - SIGTSTP idem que SIGSTOP mais il peut être capturé il est émis par le terminal de contrôle (CTRL-Z)

Les signaux classiques

- SIGTERM
 - Termine le processus de manière moins brutale que SIGKILL
 - C'est le signal envoyé par défaut par la commande shell kill.

Les signaux classiques

- SIGTRAP
 - Émis par le noyau si un processus a atteint un point d'arrêt.
 - Utilisé pour déboguer un programme.

Les signaux classiques

- SIGTTIN SIGTTOU
 - Émis par le terminal vers un processus en arrière plan qui tente de lire depuis le terminal ou d'écrire sur le terminal.
 - Par défaut tous les processus du groupe reçoivent ce signal.
 - Ils sont arrêtés sans se terminer si l'attribut TOSTOP du terminal est activé.
 - Taper les lignes de commandes suivantes :
 - ls &
 - stty tostop
 - ls &
 - fg
 - stty -tostop

Les signaux classiques

- SIGURG
 - Détecte l'arrivée de signaux hors bande sur le réseau (voir `select()`).
 - Exemple cours réseau protocole « tcp ».

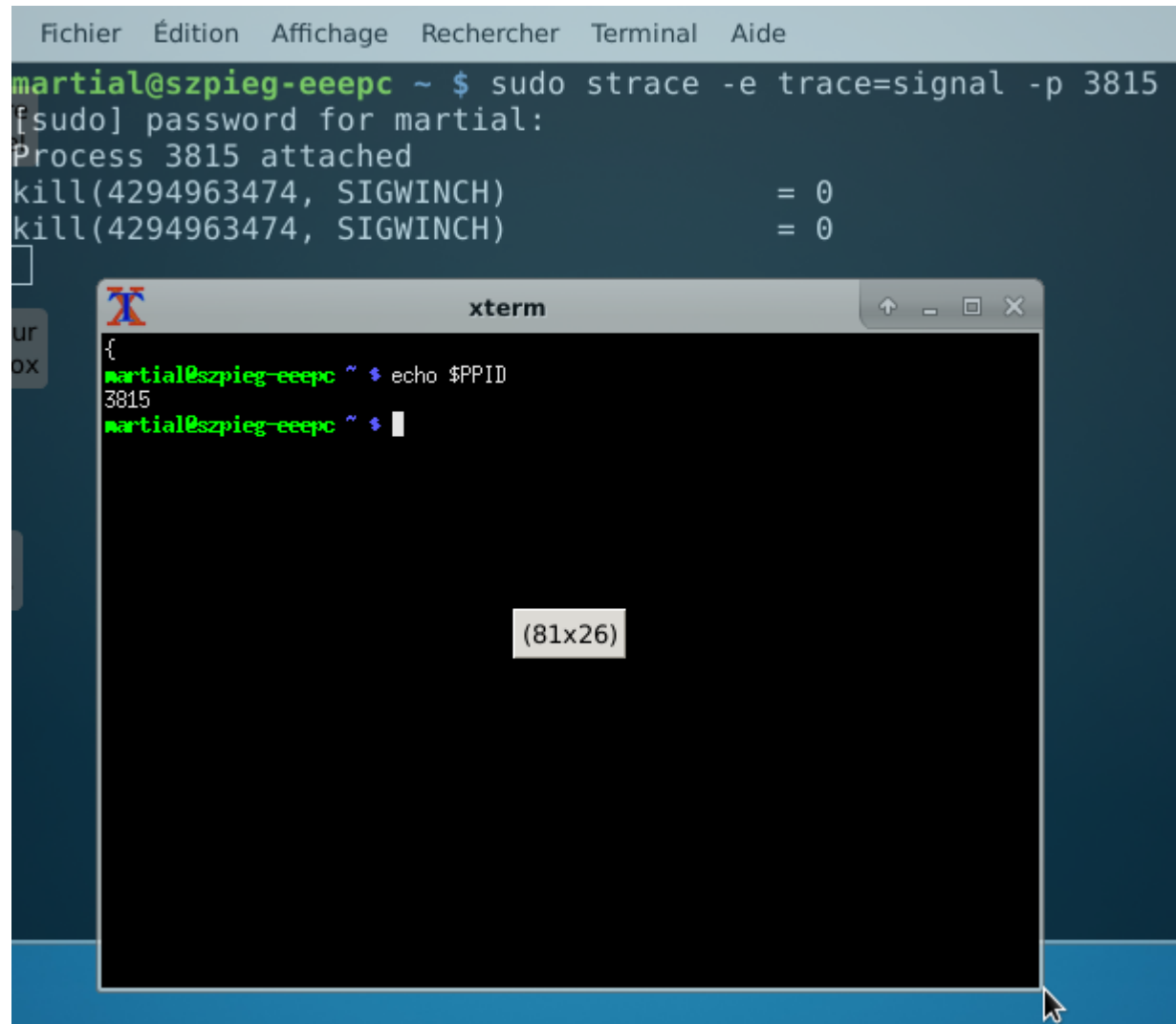
Les signaux classiques

- SIGUSR1 et SIGUSR2
 - A disposition du programmeur.
 - Par défaut termine le processus

Les signaux classiques

- SIGWINCH

Permet d'informer de la modification de la taille d'un terminal d'une application se déroulant en mode graphique (exemple : xterm) .



```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
martial@szpieg-eeepc ~ $ sudo strace -e trace=signal -p 3815
[sudo] password for martial:
Process 3815 attached
kill(4294963474, SIGWINCH)      = 0
kill(4294963474, SIGWINCH)      = 0

{
martial@szpieg-eeepc ~ $ echo $PPID
3815
martial@szpieg-eeepc ~ $
```


Les signaux classiques

- SIGXCPU et SIGXFSZ
 - Émis par le noyau si certaines limites des ressources systèmes sont atteintes :
 - Temps utilisation CPU
 - Taille maximale d'un fichier
 - Nombre maximal de fichiers ouverts
 - ...
 - Voir `getrlimit()` et `setrlimit()`.
 - Lorsqu'un processus a dépassé sa limite souple (définie par l'utilisateur) il reçoit toutes les secondes SIGXCPU qui arrête le processus par défaut
 - SIGXFSZ idem mais si le processus tente de créer un fichier trop grand en une fois.

Les signaux classiques

- Bloquer un signal.
 - Un processus peut bloquer temporairement un signal sauf SIGKILL et SIGSTOP.
 - L'ensemble des signaux en attente pour un processus constitue une table.
 - Un processus possède aussi un masque de bits indiquant les signaux bloqués.
 - Un signal reste bloqué tant que le processus ne le débloque pas.
 - Les signaux classiques ne sont pas empilés contrairement à un signal temps réel

Les signaux classiques

- Appels-système lents
 - Il existe des appels système rapides et in-interruptibles (atomique par exemple) et des appels système lents (open, read, write ...) et des appels système comme wait() qui peuvent attendre indéfiniment.
 - Si un signal arrive lors d'un appel système lent et qu'un gestionnaire de signal est activé, le noyau peut ne pas terminer l'appel système interrompu :
 - il faut alors tester les retours de fonction (read, write ...) et les relancer éventuellement
 - Ou demander au noyau de relancer automatiquement les appels. On parle d'appels système ré-entrants.

Les signaux classiques

- Gestion des appels système lents interrompus
 - La bibliothèque Glibc donne accès à une fonction `siginterrupt()`
prototype : `int siginterrupt (int numero,int bool);`
 - Cette fonction doit être appelée après l'installation du gestionnaire de signal.
 - Si le booléen est nul, alors l'appel système lent est relancé automatiquement

Les signaux classiques

- Installation d'un gestionnaire de signal
 - Définition : un gestionnaire de signal est une partie de code qui sera exécutée par un processus lors de l'arrivée d'un signal.
 - Une solution pour installer un gestionnaire de signal consiste à utiliser la fonction « signal() »
 - Prototype de signal
 - `void (*signal (int numero_signal, void (*gestionnaire) (int))) (int);`
 - Explication
- Si on pose : `typedef void (*gestion_t)(int);`
il vient : `gestion_t signal (int numero_signal, gestion_t gestionnaire);`
donc on passe comme premier argument le numéro du signal concerné
et en second l'adresse où se trouve le code à exécuter.
- La fonction retourne un pointeur sur l'ancien gestionnaire de signal ou un entier défini par les constantes symboliques `SIG_IGN` (ignore le signal) et `SIG_DFL` (comportement par défaut) on obtient un entier aussi si l'appel `signal()` échoue `SIG_ERR` ou `EFAULT` pointeur de gestionnaire invalide.

Signaux et Posix.1

- Installer un gestionnaire de signal avec « sigaction() »
 - Prototype : `int sigaction (int numero, const struct sigaction * nouvelle, struct sigaction * ancienne);`
 - Attention !!! : le terme « sigaction » est utilisé pour la fonction mais aussi pour la structure ne pas confondre les deux.
- # Le premier paramètre est le numéro du signal à capturer comme pour la fonction « signal »

Signaux et Posix.1

#Le deuxième paramètre est une structure sigaction qui contient :

- *sighandler_t sa_handler;*
pointeur sur le gestionnaire de signal (comme la fonction « signal »)
Il peut aussi prendre les valeurs SIG_IGN et SIG_DFL
- le prototypage du gestionnaire de signal sera:*
void gestionnaire_signal (int numero);
sauf si sa_siginfo activé (voir diapo suivante)
- *sigset_t sa_mask;*
liste des signaux bloqués pendant l'exécution du gestionnaire.
Remarque : le signal ayant déclenché le gestionnaire est bloqué par défaut.

Signaux et Posix.1

– Structure sigaction suite :

- `int sa_flags;`

peut contenir les valeurs suivantes combinées par un « ou » binaire:

- *SA_NOCLDSTOP pas d'appel au gestionnaire si le processus fils s'arrête temporairement, appel si arrêt définitif (que pour le signal SIGCHLD).*
- *SA_RESTART redémarrage des appels systèmes lents.*
- *SA_NODEFER empêche de bloquer les signaux de sa_mask.*
- *SA_ONESHOT remet le comportement par défaut à la fin du gestionnaire.*
- *SA_SIGINFO permet d'obtenir une structure donnant des renseignements complémentaires sur le signal déclencheur (origine, propriétaire du process..) au gestionnaire de signal temps réel (voir struct siginfo ci-dessous)*
- *SA_ONSTACK le gestionnaire utilise une pile différente du reste du processus*

- *Lorsque la valeur SA_SIGINFO est active, le prototypage du gestionnaire sera le suivant :*

*void gestionnaire_signal (int numero, struct siginfo * info, void *inutile);*

Signaux et Posix.1

- *Structure sigaction suite et fin : le dernier élément de la structure "sa_restorer" est un pointeur permettant de gérer la pile*
 - *Si SA_ONSTACK est activée la pile est alors sauvegardée à cette adresse.*
- # Le troisième et dernier paramètre de la "fonction" sigaction est une structure sigaction :
 - Permettant de restaurer le gestionnaire de départ.

Signaux et Posix.1

- Configuration de l'ensemble des signaux pour l'élément `sa_flag` de la structure `sigaction` :
 - Routines définies par Posix.1
 - `sigemptyset`
 - `sigfillset`
 - `sigaddset`
 - `sigdelset`
 - `sigismember`

Signaux et Posix.1

- Blocage des signaux
la fonction `sigprocmask` permet à un processus de bloquer des signaux (`SIGKILL` et `SIGSTOP` exclus).
 - Prototypage :
`int sigprocmask (int methode, const sigset_t *ensemble, sigset_t *ancien);`
 - `int methode` permet d'indiquer la manière d'utiliser l'ensemble passé en second argument :
 - `SIG_BLOCK` ajoute l'ensemble au masque existant.
 - `SIG_UNBLOCK` débloque l'ensemble du masque existant.
 - `SIG_SETMASK` utilise cet ensemble comme masque.

Signaux et Posix.1

- Consultation des signaux bloqués
L'appel-système sigpending permet à un processus de consulter la liste des signaux bloqués sans les débloquenter.
 - Prototypage :
 - `int sigpending(sigset_t * ensemble);`

Signaux temps réel

Les signaux Temps Réel

- Ces signaux sont appelés temps réel car le temps mis pour effectuer la tâche est le facteur prépondérant (Limites temporelles connues).
- Ces signaux sont réservés à l'utilisateur et ne sont pas générés par le noyau. C'est donc une extension de SIGUSR1 et SIGUSR2.
- Ces signaux sont ordonnés en fonction de leur priorité et fournissent des informations plus précises que les signaux classiques.

Les signaux Temps Réel

- Caractéristiques des signaux temps réel
 - Nombre plus important de signaux utilisateurs
 - Empilement des occurrences des signaux bloqués (Signaux classiques si 2 occurrences d'un même signal arrivent à un processus seule une est délivrée, ce qui n'est pas le cas pour les signaux TR, on peut empiler 1024 signaux maxi, utilisation de `sigqueue()` préférable à `kill()` pour émettre le signal).
 - Délivrance prioritaire des signaux (signaux de plus faible numéro en premier)
 - Pas de nom spécifique, on utilise leur numéro (préférer le positionnement relatif (Ex : `SIGRTMIN + 2`))
 - Informations supplémentaires fournies au gestionnaire de signal

Les signaux Temps Réel

Informations supplémentaires (suite)

- Caractéristiques des signaux temps réel
 - Le gestionnaire de signal peut, pour les signaux temps réel, récupérer des informations supplémentaires dans une union sigval (Dans la structure siginfo voir précédemment) dont la valeur peut être soit :
 - int utilisation de sigval_int;
 - void* utilisation de sigval_ptr;
 - Attention pour que ce gestionnaire soit installé il faut le placer dans le champ sa_sigaction d'une structure sigaction et non pas dans sa_handler, de plus le drapeau SA_SIGINFO doit être activé.
 - Pour émettre le signal, au lieu d'utiliser kill , on aura recours à la fonction sigqueue dont le prototypage est :
int sigqueue (pid_t,int numero, const union sigval valeur)
 - Attention lorsque valeur contient un pointeur passé entre deux processus, il est nécessaire que le pointeur passé en argument se trouve sur un espace mémoire partagé.

Les signaux Temps Réel

- Exemple d'initialisation d'un gestionnaire de signal temps réel :

```
struct sigaction action;
```

```
action.sa_sigaction=gestionnaire_de_signal;
```

```
sigemptyset(&action.sa_mask);
```

```
action.sa_flags=SA_SIGINFO;
```

```
sigaction (SIGRTMIN+1,&action,NULL);
```

Les signaux Temps Réel

- Émission du signal temps réel :
 - Pour émettre un signal temps réel avec des informations complémentaires, on utilise la commande "sigqueue" dont le prototypage est le suivant :
 - `int sigqueue (pid_t pid, int numero, const union sigval valeur)`
il y a donc un argument supplémentaire par rapport à "kill" c'est l'union sigval.
 - Cette union sigval est un des membres de la structure sigaction vue précédemment.

Les signaux Temps Réel

–Contenu de la structure `siginfo`

- `int si_signo` : numéro du signal (redondance avec le premier paramètre du gestionnaire de signal)
- `int si_value.sigval_int` : entier passé par `sigqueue`
- `int si_value.sigval_ptr` : pointeur passé par `sigqueue` si ce n'est pas `sigval_int`.
- `int err_no` : valeur `errno` lors du déclenchement du gestionnaire.
- `pid_t si_pid` : pid de l'émetteur si le signal est temps réel ou du fils terminé si `SIGCHLD`
- `uid_t si_uid` : UID réel de l'émetteur
- `int si_status` : code de retour avec `SIGCHLD`
- `int si_code` : Origine du signal « `SI_KERNEL` vient du noyau, `SI_USER` appel avec `kill` ou `raise`, `SI_QUEUE` appel avec `sigqueue`, `SI_ASYNCIO` terminaison d'une entrée sortie asynchrone, `SI_SIGIO` changement d'état d'un descripteur asynchrone. »

–Pour certains signaux classiques, l'argument `sa_flags` fournit des arguments utiles au débogage (sur LINUX mais pas portable)