

# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)

# Qu'est ce qu'un processus ?

## Définition formelle :

- ▶ Entité **dynamique** représentant l'exécution d'un programme sur un processeur

## Du point de vue du système d'exploitation :

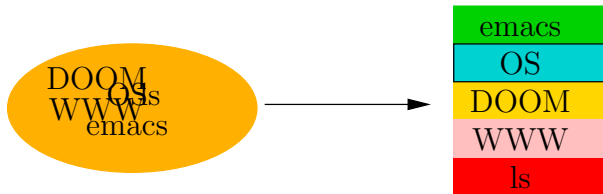
- ▶ Espace d'adressage (mémoire, contient données + code)
- ▶ État interne (compteur d'exécution, fichiers ouverts, etc.)

## Exemples de processus :

- ▶ L'exécution d'un programme
- ▶ La copie d'un fichier sur disque
- ▶ La transmission d'une séquence de données sur un réseau

# Utilité des processus : simplicité

- ▶ L'ordinateur a des activités différentes
- ▶ Comment les faire cohabiter simplement ?
  - ▶ En plaçant chaque activité dans un processus isolé  
L'OS s'occupe de chacun de la même façon, chacun ne s'occupe que de l'OS



- ▶ La décomposition est une réponse classique à la complexité

# Ne pas confondre processus et programme

## ▶ Programme :

Code + données (**passif**)

```
int i;  
int main() {  
    printf("Salut\n");  
}
```

## ▶ Processus :

Programme **en cours d'exécution**

Pile	
Tas	
Données	int i;
Code	main()

- ▶ Vous pouvez utiliser le même programme que moi, mais ça ne sera pas le même processus que moi
- ▶ Même différence qu'entre classe d'objet et instance d'objet

# Utilité des processus : efficacité

## ► Les communications bloquent les processus

(communication au sens large : réseau, disque ; utilisateur, autre programme)

⇒ **recouvrement des calculs et des communications**



## ► Parallélisme sur les machines multi-processeurs

# Parallélisme et pseudo-parallélisme

Que faire quand deux processus sont prêts à s'exécuter ?

- ▶ Si deux processeurs, tout va bien.
- ▶ Sinon, FCFS ? Mauvaise interactivité !



- ▶ Pseudo-parallélisme = chacun son tour
- ▶ Autre exécution pseudo-parallèle



## Le pseudo-parallélisme

- ▶ fonctionne grâce aux interruptions matérielles régulières rendant contrôle à OS
  - ▶ permet également de recouvrir calcul et communications
- On y reviendra.

# Relations entre processus

## Compétition

- ▶ Plusieurs processus veulent accéder à une ressource exclusive (*i.e.* ne pouvant être utilisée que par un seul à la fois) :
  - ▶ Processeur (cas du pseudo-parallélisme)
  - ▶ Imprimante, carte son
- ▶ Une **solution possible** parmi d'autres :  
FCFS : premier arrivé, premier servi (les suivants **attendent** leur tour)

## Coopération

- ▶ Plusieurs processus collaborent à une tâche commune
- ▶ Souvent, ils doivent se **synchroniser** :
  - ▶ p1 produit un fichier, p2 imprime le fichier
  - ▶ p1 met à jour un fichier, p2 consulte le fichier
- ▶ La synchronisation se ramène à :  
p2 doit **attendre** que p1 ait franchi un certain point de son exécution

# Faire attendre un processus

Primordial pour les interactions entre processus

## Attente active

Processus 1

```
while (ressource occupée)
{ };
ressource occupée = true;
```

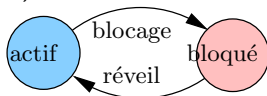
Processus 2

```
ressource occupée = true;
utiliser ressource;
ressource occupée = false;
```

- ▶ Gaspillage de ressource si pseudo-parallélisme
- ▶ Problème d'atomicité (*race condition* – on y reviendra)

## Blocage du processus

- ▶ Définition d'un nouvel état de processus : **bloqué**  
(exécution suspendue; réveil explicite par un autre processus ou par le système)



```
...
sleep(5); /* se bloquer pour 5 secondes */
...
```



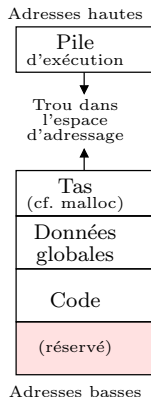
# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)

# Processus UNIX

- ▶ Processus = exécution d'un programme
  - ▶ Commande (du langage de commande)
  - ▶ Application
- ▶ Un processus comprend :
  - ▶ Une mémoire qui lui est propre (mémoire virtuelle)
  - ▶ **Contexte** d'exécution (pile, registres du processeur)
- ▶ Les processus sont identifiés par leur **pid**
  - ▶ Commande ps : liste des processus
  - ▶ Commande top : montre l'activité du processeur
  - ▶ Primitive getpid() : renvoie le pid du processus courant



# Environnement d'un processus

- ▶ Ensemble de variables accessibles par le processus (sorte de configuration)
- ▶ Principaux avantages :
  - ▶ L'utilisateur n'a pas à redéfinir son contexte pour chaque programme  
Nom de l'utilisateur, de la machine, terminal par défaut, ...
  - ▶ Permet de configurer certains éléments  
Chemin de recherche des programmes (PATH), shell utilisé, ...
- ▶ Certaines sont prédéfinies par le système (et modifiables par l'utilisateur)
- ▶ L'utilisateur peut créer ses propres variables d'environnement
- ▶ Interface (dépend du shell) :

Commande tcsh	Commande bash	Action
setenv	printenv	affiche toutes les variables définies
setenv VAR <valeur>	export VAR=<valeur>	attribue la valeur à la variable
echo \$VAR	echo \$VAR	affiche le contenu de la variable

- ▶ Exemple : `export DISPLAY=blaise.loria.fr:0.0` définit le terminal utilisé
- ▶ L'interface de programmation sera vue en TD/TP

# Vie et mort des processus

## Tout processus a un début et une fin

- ▶ **Début** : création par un autre processus
  - ▶ `init` est le processus originel : `pid=1` (`launchd` sous mac)  
Créé par le noyau au démarrage, il lance les autres processus système
- ▶ **Fin**
  - ▶ Auto-destruction (à la fin du programme) (par `exit`)
  - ▶ Destruction par un autre processus (par `kill`)
  - ▶ Destruction par l'OS (en cas de violation de protection et autres)
  - ▶ Certains processus ne se terminent pas avant l'arrêt de la machine
    - ▶ Nommés «démons» (disk and execution monitor → `daemon`)
    - ▶ Réalisent des fonctions du système (login utilisateurs, impression, serveur web)

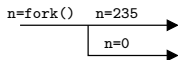
## Création de processus dans UNIX

- ▶ Dans le langage de commande :
  - ▶ Chaque commande est exécutée dans un processus séparé
  - ▶ On peut créer des processus en (pseudo-)parallèle :  
`$ prog1 & prog2 & # crée deux processus pour exécuter prog1 et prog2`  
`$ prog1 & prog1 & # lance deux instances de prog1`
- ▶ Par l'API : clonage avec l'appel système `fork` (cf. transparent suivant)

# Création des processus dans Unix (1/3)

## Appel système `pid_t fork()`

- ▶ Effet : clone le processus appelant
- ▶ Le processus créé (fils) est une copie conforme du processus créateur (père)  
Copies conformes comme une bactérie qui se coupe en deux
- ▶ Ils se reconnaissent par la valeur de retour de `fork()` :
  - ▶ Pour le père : le pid du fils (ou `-1` si erreur)
  - ▶ Pour le fils : `0`



## Exemple :

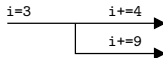
```
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
} else {
    printf("je suis le fils, mon PID est %d; mon père est %d\n",
        getpid(), getppid());
    /* en général exec(), (exécution d'un nouveau programme) */
}
```

## Création des processus dans Unix (2/3)

Duplication du processus père  $\Rightarrow$  duplication de l'espace d'adressage

```
int i=3;
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    i += 4;
} else {
    printf("je suis le fils, mon PID est %d; mon père est %d\n",
        getpid(), getppid());
    i += 9;
}
printf("pour %d, i = %d\n", getpid(), i);
```

```
je suis le fils, mon PID est 10271; mon père est 10270
pour 10271, i = 12
je suis le père, mon PID est 10270
pour 10270, i = 7
```

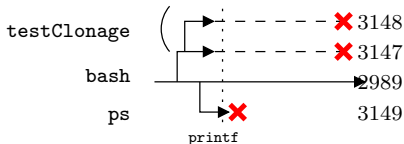


# Création des processus dans Unix (3/3)

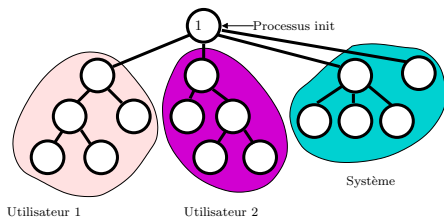
## testClonage.c

```
int main() {
    if (fork() != 0) {
        printf("je suis le père, mon PID est %d\n", getpid());
        sleep(10) /* blocage pendant 10 secondes */
        exit(0);
    } else {
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(10) /* blocage pendant 10 secondes */
        exit(0);
    }
}
```

```
$ gcc -o testClonage testClonage.c
$ ./testClonage & ps
je suis le fils, mon PID est 3148
je suis le père, mon PID est 3147
[2] 3147
  PID TTY          TIME CMD
2989 pts/0    00:00:00 bash
3147 pts/0    00:00:00 testClonage
3148 pts/0    00:00:00 testClonage
3149 pts/0    00:00:00 ps
$
```



# Hiérarchie de processus Unix



## ► Quelques appels systèmes utiles :

- `getpid()` : obtenir le numéro du processus
- `getppid()` : obtenir le numéro du père
- `getuid()` : obtenir le numéro d'utilisateur (auquel appartient le processus)



## Quelques interactions entre processus (1/2)

### Envoyer un signal à un autre processus

- ▶ En langage de commande, `kill <pid>` tue pid (plus de détails plus tard)

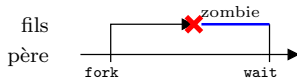
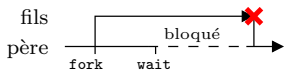
### Faire attendre un processus

- ▶ `sleep(n)` : se bloquer pendant n secondes
- ▶ `pause()` : se bloquer jusqu'à la réception d'un signal (cf. plus tard)

## Quelques interactions entre processus (2/2)

### Synchronisation entre un processus père et ses fils

- ▶ Fin d'un processus : `exit`(etat)  
etat est un code de fin (convention : 0 si ok, code d'erreur sinon – cf. `errno`)
- ▶ Le père attend la fin de l'un des fils : `pid_t wait`(int \*ptr\_etat)  
retour : pid du fils qui a terminé; code de fin stocké dans `ptr_etat`
- ▶ Attendre la fin du fils `pid` :  
`pid_t waitpid`(pid\_t pid, int \*ptr\_etat, int options)
- ▶ Processus zombie : terminé, mais le père n'a pas appelé `wait`().  
Il ne peut plus s'exécuter, mais consomme encore des ressources. À éviter.



# Faire attendre un processus

## Fonction `sleep()`

- ▶ Bloque le processus courant pour le nombre de secondes indiqué
- ▶ `unsigned int sleep(unsigned int seconds);`
- ▶ `usleep()` et `nanosleep()` offrent meilleures résolutions (micro, nanoseconde) mais interfaces plus compliquées et pas portables

somnole : affiche à chaque seconde le temps restant à dormir

```
void somnole(unsigned int secondes) {
    int i;
    for (i=0; i<secondes; i++) {
        printf("Déjà dormi %d secondes sur %d\n", i, secondes);
        sleep(1);
    }
}
```

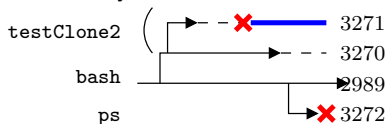
# Exemple de synchronisation entre père et fils

## testClone2.c

```
int main() {
    if (fork() != 0) {
        printf("je suis le père, mon PID est %d\n", getpid());
        while (1) ; /* boucle sans fin sans attendre le fils */
    } else {
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(2) /* blocage pendant 2 secondes */
        printf("fin du fils\n");
        exit(0);
    } }
}
```

```
$ gcc -o testClone2 testClone2.c
$ ./testClone2
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
->l'utilisateur tape <ctrl-Z> (suspendre)
Suspended
$ ps
  PID TTY          TIME CMD
 2989 pts/0    00:00:00 bash
 3270 pts/0    00:00:03 testClone2
 3271 pts/0    00:00:00 testClone2 <defunct>
 3272 pts/0    00:00:00 ps
$
```

Il y a un zombie



## Autre exemple de synchronisation père fils

### testClone3.c

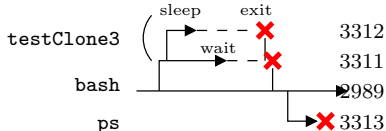
```
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    if (fork() != 0) {
        int statut; pid_t fils;
        printf("Le père (%d) attend.\n", getpid());
        fils = wait(&statut);
        if (WIFEXITED(statut)) {
            printf("%d : fils %d terminé (code %d)\n",
                getpid(), fils, WEXITSTATUS(statut));
        }
    };
    exit(0);
}
```

```
$ ./testClone3
je suis le fils, PID=3312
Le père (3311) attend
fin du fils
3311: fils 3312 terminé (code 1)
$ ps
  PID TTY          TIME CMD
 2989 pts/0    00:00:00 bash
 3313 pts/0    00:00:00 ps
$
```

### (suite de testClone3.c)

```
    } else { /* correspond au if (fork() != 0)*/
        printf("je suis le fils, PID=%d\n",
            getpid());
        sleep(2) /* blocage pendant 2 secondes */
        printf("fin du fils\n");
        exit(1);
    } }
}
```

Il n'y a pas de zombie



# Exécution d'un programme spécifié sous UNIX

## Appels systèmes `exec`

- ▶ Pour faire exécuter un nouveau programme par un processus
- ▶ Souvent utilisé immédiatement après la création d'un processus :  
`fork+exec` = lancement d'un programme dans un nouveau processus
- ▶ **Effet** : remplace la mémoire virtuelle du processus par le programme
- ▶ Plusieurs variantes existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement)
- ▶ C'est aussi une primitive du langage de commande (même effet)

### Exemple :

```
main() {
    if (fork() == 0) {

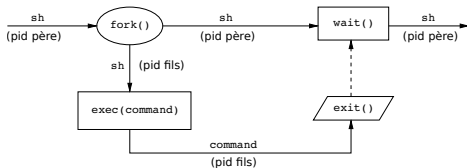
        code=execl("/bin/ls", "ls", "-a", 0); /* le fils exécute : /bin/ls -a .. */
        if (code != 0) { ... } /* Problème dans l'appel système; cf. valeur de errno */
    } else {
        wait(NULL); /* le père attend la fin du fils */
    }
    exit(0);
}
```

# L'exemple du shell

## Exécution d'une commande en premier plan

\$ commande

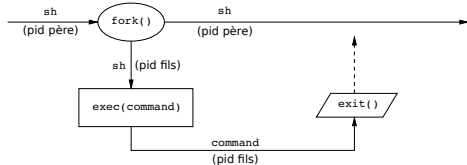
- ▶ 4 syscalls : fork, exec, exit, wait



## Exécution d'une commande en tâche de fond

\$ commande &

- ▶ Le shell ne fait pas wait()
- ▶ Il n'est plus bloqué



# Résumé du début du deuxième chapitre

- ▶ Utilité des processus
  - ▶ Simplicité (séparation entre les activités)
  - ▶ Sécurité (séparation entre les activités)
  - ▶ Efficacité (*quand l'un est bloqué, on passe à autre chose*)
- ▶ Interface UNIX
  - ▶ Création : `fork()`
    - ▶ `résultat=0` → je suis le fils
    - ▶ `résultat>0` → je suis le père (`résultat=pid` du fils)
    - ▶ `résultat<0` → erreur
  - ▶ Attendre un fils : deux manières
    - ▶ `wait()` : n'importe quel fils
    - ▶ `waitpid()` : un fils en particulier
  - ▶ Processus zombie : un fils terminé dont le père n'a pas fait `wait()`
  - ▶ Bloquer le processus courant : deux façons
    - ▶ Jusqu'au prochain signal : `pause()`
    - ▶ Pendant 32 secondes : `sleep(32)`
  - ▶ Appel système `exec()` : remplace l'image du process actuel par le programme spécifié



# Parenthèse sécurité : Shellshock/Bashdoor (2014)

- ▶ Série de bugs découverts dans le shell Bash : CVE-2014-6271, CVE-2014-6277, CVE-2014-6278, CVE-2014-7169, CVE-2014-7186, CVE-2014-7187 (CVE = *Common Vulnerabilities and Exposures* : base de données des vulnérabilités)
- ▶ Impact très important

The screenshot shows a news article from Courrier International. At the top, there is a navigation bar with categories like 'COURRIER EXPAT', 'LE MONDE', 'TELEGRAMA', 'LE MONDE DIPLOMATIQUE', 'LE HUFFINGTON POST', and 'LA VIE'. The main header features the 'Courrier international' logo and a small image of a person. Below the header, there is a sub-navigation bar with topics such as 'ENCEMOMENT', 'CORÉE DU NORD', 'ÉLECTIONS EN ALLEMAGNE', 'CATALOGNE', 'VENEZUELA', and 'EMMANUEL MACRON'. The article title is 'Technologie. Shellshock, la faille qui sème la panique'. To the left of the title are social media sharing icons (Facebook, Twitter, Google+) and buttons for 'PARTAGER', 'RÉAGIR', 'IMPRIMER', 'LECTURE ZEN', and 'NEWSLETTERS'. Below these is a yellow subscription box that says 'ABONNEZ-VOUS À PARTIR DE 1€'. The article text begins with 'ÉTATS-UNIS | COURRIER INTERNATIONAL - PARIS' and 'Publié le 25/09/2014 - 17:32'. A terminal window is shown with the command '\$ env x='() { : }; echo vulnérable''. To the right of the terminal is a red padlock icon. At the bottom of the terminal area, the word 'Shellshock' is written in large red letters. On the right side of the article, there is a section titled 'LES PLUS LUS' with three sub-articles: 'Etats-Unis De Kodak à Apple : comment les inégalités ont explosé', 'Franc CFA. Kérékou Soba, le sulfureux militant de la cause panafricaine', and 'Birmanie. Rohingya : les questions clés pour comprendre le conflit dans l'Arakan'. Each sub-article has a 'Partager' button. At the bottom right, there is another sub-article titled 'Travail. Les mensonges les plus courants au bureau (et ce qu'ils veulent dire)' and 'Vi du Royaume-Uni. Drones de Koz'.

## Shellshock/Bashdoor (2014) (suite)

- ▶ Au lancement, le shell `bash` analyse ses variables d'environnement, notamment pour définir des fonctions qui lui seraient transmises par un shell parent.
- ▶ À cause d'un bug (présent depuis 1989), du code shell présent après la définition de la fonction va être exécuté.

```
env X="() { :; } ; echo busted" bash -c "echo completed"
```

~> On peut faire exécuter n'importe quoi à bash en *polluant* ses variables d'environnement

- ▶ Exemple de vecteur d'exploitation : serveur web utilisant CGI
    - ▶ Moyen historique d'avoir des sites webs interactifs (exemple 1, 2)
    - ▶ Un programme exécutable (souvent un script Perl, Python, Ruby, Bash) est exécuté pour traiter la requête
    - ▶ Les paramètres de la requête et les en-têtes HTTP sont transmis via des variables d'environnement
    - ▶ Y compris `HTTP_USER_AGENT` : le navigateur utilisé
- ```
wget --user-agent="() { test; };/usr/bin/touch /tmp/VULNERABLE"  
http://server/script.cgi
```

# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- **Réalisation des processus UNIX**
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)

# Réalisation des processus

**Processus = mémoire virtuelle + flot d'exécution**

L'OS fournit ces deux ressources en allouant les ressources physiques

## Objectif maintenant :

En savoir assez sur le fonctionnement de l'OS pour utiliser les processus

- ▶ À propos de **mémoire**
  - ▶ Organisation interne de la mémoire virtuelle d'un processus Unix
- ▶ À propos de **processeur**
  - ▶ Pseudo-parallélisme : allocation successive aux processus par tranches de temps



## Objectifs repoussés à plus tard :

- ▶ Les autres ressources : disque (chapitre 3), réseau (seconde moitié du module)
- ▶ Détails de conception *sous le capot* (module RSA)

# Allocation du processeur aux processus

## Pseudo-parallélisme

### Principe

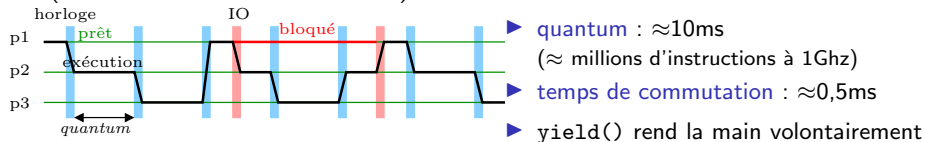
- ▶ Allocation successive aux processus par tranches de temps fixées (multiplexage du processeur par préemption)

### Avantages

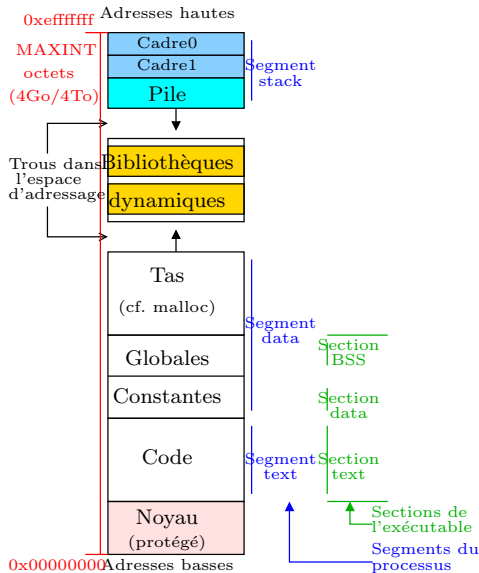
- ▶ Partage équitable du processeur entre processus (gestion + protection)
- ▶ Recouvrement calcul et communications (ou interactions)

### Fonctionnement

- ▶ Interruptions matérielles (top d'horloge, I/O, ...) rendent le contrôle à l'OS
- ▶ L'OS **ordonnance** les processus (choisit le prochain à bénéficier de la ressource)
- ▶ Il réalise la **commutation de processus** pour passer le contrôle à l'heureux élu
- ▶ (Comment ? Vous verrez en RSA !)



# Structure de la mémoire virtuelle d'un processus



- ▶ **Pile** : pour la récursivité
  - ▶ Cadres de fonction (*frame*)
  - ▶ Arguments des fonctions
  - ▶ Adresse de retour
  - ▶ Variables locales (non statique)
- ▶ **Bibliothèques dynamiques**
  - ▶ Code chargé ... dynamiquement
  - ▶ Intercalé dans l'espace d'adresses
- ▶ **Données**
  - ▶ **Tas** : `malloc()`
  - ▶ **globales** et **static** (modifiables)
  - ▶ Variables **constantes**
- ▶ **exec** lit l'exécutable et initialise la mémoire correspondante
- ▶ **Noyau** : infos sur le processus "Process Control Block" (pid, autorisations, fichiers ouverts, ...) (parfois au dessus de la pile)

# Sécurité : attaques par corruption de la mémoire

```
void fonction(char * chaine) {  
    char buffer[128];  
    strcpy(buffer, chaine);  
}
```

- ▶ Famille de failles très répandue (*buffer overflow*)
  - ▶ Récemment : ransomware WannaCry (mai 2017)
- ▶ Principe général : faire exécuter du code quelconque à un processus
- ▶ Si chaîne fait plus de 128 caractères, alors `strcpy` écrasera l'adresse de retour de la fonction
- ▶ Si chaîne contient du code machine, le processus exécutera ce code machine
- ▶ Plus de détails : <http://mdeloisson.free.fr/downloads/memattacks.pdf>
- ▶ Mécanismes de défense :
  - ▶ *W xor X* : interdire qu'une zone mémoire soit à la fois inscriptible et exécutable
  - ▶ *canaris* pour détecter les dépassements de pile
  - ▶ *ASLR (Address Space Layout Randomization)* : les adresses des fonctions et de la pile sont plus difficiles à prévoir
  - ▶ En assembleur/C/C++, utiliser des fonctions permettant de traiter des données de manière sécurisée (`strcpy`  $\leadsto$  `strncpy`)
  - ▶ Utiliser des langages de plus haut niveau
  - ▶ Utiliser des outils de test automatiques (*fuzzing*)

# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)



## Aspect central de la programmation système

### Moyens de communication entre processus sous Unix

- ▶ Signaux : suite de cette séance
- ▶ Fichiers et tubes (*pipes*, FIFOs) : partie 3.
- ▶ Files de messages : pas étudié dans ce module
- ▶ Mémoire partagée et sémaphores : partie 5.
- ▶ Sockets (dans les réseaux, mais aussi en local) : fin du semestre

# Signaux

## Définition : événement asynchrone

- ▶ Émis par l'OS ou un processus
- ▶ Destiné à un (ou plusieurs) processus

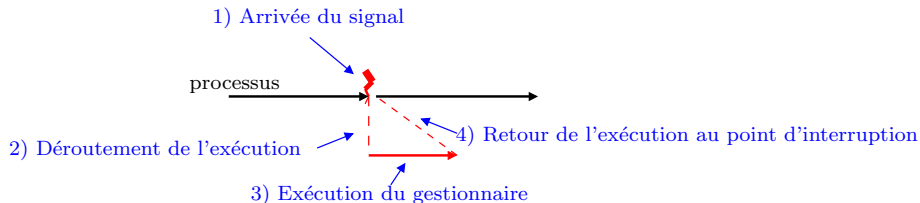
## Intérêts et limites

- ▶ Simplifient le contrôle d'un ensemble de processus (comme le shell)
- ▶ Pratiques pour traiter des événements liés au temps
- ▶ Mécanisme de bas niveau à manipuler avec précaution (risque de perte de signaux en particulier)

## Comparaison avec les interruptions matérielles :

- ▶ Analogie : la réception déclenche l'exécution d'un gestionnaire (*handler*)
- ▶ Différences : interruption reçue par processeur ; signal reçu par processus  
Certains signaux traduisent la réception d'une interruption (on y revient)

# Fonctionnement des signaux



## Remarques (on va détailler)

- ▶ On ne peut évidemment *signaler* que ses propres processus (même uid)
- ▶ Différents signaux, identifiés par un nom symbolique (et un entier)
- ▶ Gestionnaire par défaut pour chacun
- ▶ Gestionnaire vide  $\Rightarrow$  ignoré
- ▶ On peut changer le gestionnaire (sauf exceptions)
- ▶ On peut bloquer un signal : mise en attente, délivré qu'après déblocage
- ▶ Limites aux traitements possibles dans le gestionnaire (ex : pas de `signal()`)

## Quelques exemples de signaux

| Nom symbolique | Cause/signification                                         | Par défaut                        |
|----------------|-------------------------------------------------------------|-----------------------------------|
| SIGINT         | frappe du caractère <CTRL-C>                                | terminaison                       |
| SIGTSTP        | frappe du caractère <CTRL-Z>                                | suspension                        |
| SIGSTOP        | blocage d'un processus (*)                                  | suspension                        |
| SIGCONT        | continuation d'un processus stoppé                          | reprise                           |
| SIGTERM        | demande de terminaison                                      | terminaison                       |
| SIGKILL        | terminaison immédiate (*)                                   | terminaison                       |
| SIGSEGV        | erreur de segmentation<br>(violation de protection mémoire) | terminaison<br>+ <i>core dump</i> |
| SIGALRM        | top d'horloge (réglée avec alarm)                           | terminaison                       |
| SIGCHLD        | terminaison d'un fils                                       | ignoré                            |
| SIGUSR1        | pas utilisés par le système                                 | terminaison                       |
| SIGUSR2        | (disponibles pour l'utilisateur)                            | terminaison                       |

- ▶ KILL et STOP : ni bloquables ni ignorables ; gestionnaire non modifiable.
  - ▶ Valeurs numériques associées (ex : SIGKILL=9), mais pas portable
  - ▶ Voir man 7 signal pour d'autres signaux (section 7 du man : conventions)
- core dump* : copie image mémoire sur disque (premières mémoires : toriques → core; dump=vidanger)

# États d'un signal

## Signal **pendant** (*pending*)

- ▶ Arrivé au destinataire, mais pas encore traité

## Signal **traité**

- ▶ Le gestionnaire a commencé (et peut-être même fini)

## Pendant, mais pas traité? Est-ce possible?

- ▶ Il est **bloqué**, càd retardé : il sera délivré lorsque débloqué
- ▶ Lors de l'exécution du gestionnaire d'un signal, ce signal est bloqué

## **Attention** : au plus un signal pendant de chaque type

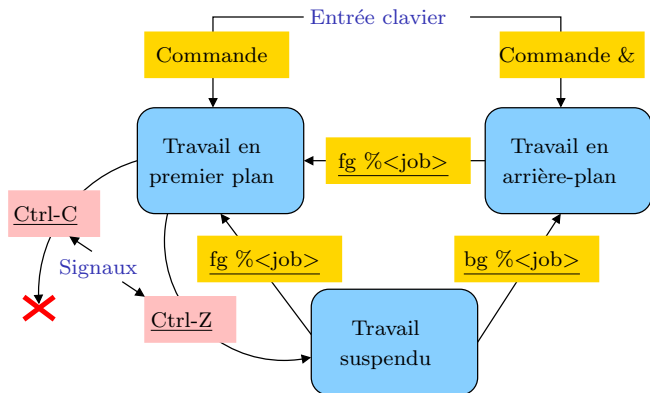
- ▶ L'information est codée sur un seul bit
- ▶ S'il arrive un autre signal du même type, le second est perdu

# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)

# États d'un travail



- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

# Terminaux, sessions et groupes en Unix

- ▶ Concept important pour comprendre les signaux sous Unix (détaillé plus tard)
- ▶ Une **session** est associée à un **terminal**, donc au login d'un utilisateur par shell  
Le processus de ce shell est le **leader de la session**.
- ▶ Plusieurs groupes de processus par session. On dit plusieurs **travaux** (*jobs*)
- ▶ Au plus un travail interactif (**avant-plan**, *foreground*)  
Interagissent avec l'utilisateur via le terminal, seuls à pouvoir lire le terminal
- ▶ Plusieurs travaux en **arrière plan** (*background*)  
Lancés avec & ; Exécution en travail de fond
- ▶ Signaux SIGINT (frappe de <CTRL-C>) et SIGTSTP (frappe de <CTRL-Z>) sont passés au groupe interactif et non aux groupes d'arrière-plan



# Exemple avec les sessions et groupes Unix

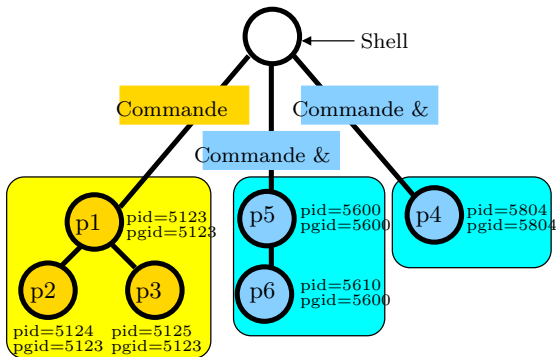
loop.c

```
int main() {
    printf("processus %d, groupe %d\n", getpid(), getpgrp());
    while(1) ;
}
```

```
$ loop & loop & ps
processus 10468, groupe 10468
[1] 10468
processus 10469, groupe 10469
[2] 10469
  PID TTY          TIME CMD
 5691 pts/0    00:00:00 bash
10468 pts/0    00:00:00 loop
10469 pts/0    00:00:00 loop
10470 pts/0    00:00:00 ps
$ fg %1
loop
[frappe de control-Z]
Suspended
$ jobs
[1] + Suspended          loop
[2] - Running            loop
```

```
$ bg %1
[1] loop &
$ fg %2
loop
[frappe de control-C]
$ ps
  PID TTY          TIME CMD
 5691 pts/0    00:00:00 bash
10468 pts/0    00:02:53 loop
10474 pts/0    00:00:00 ps
$ [frappe de control-C]
$ ps
  PID TTY          TIME CMD
 5691 pts/0    00:00:00 bash
10468 pts/0    00:02:57 loop
10475 pts/0    00:00:00 ps
$
```

## Exemple de travaux



- ▶ Signaux CTRL-C et CTRL-Z adressés à **tous** les processus du groupe jaune
- ▶ Commandes shell fg, bg et stop pour travaux bleus

# Deuxième chapitre

## Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion
- Évolution récente : groupes de processus (cgroups)

# Envoyer un signal à un autre processus

## Interfaces

- ▶ Langage de commande :

```
kill -NOM victime
```

- ▶ Appel système :

```
#include <signal.h>

int kill(pid_t victime, int sig);
```

## Sémantique : à qui est envoyé le signal ?

- ▶ Si  $victime > 0$ , au processus tel que  $pid = victime$
- ▶ Si  $victime = 0$ , à tous les processus du même groupe (pgid) que l'émetteur
- ▶ Si  $victime = -1$  :
  - ▶ Si super-utilisateur, à tous les processus sauf système et émetteur
  - ▶ Si non, à tous les processus dont l'utilisateur est propriétaire
- ▶ Si  $victime < -1$ , aux processus tels que  $pgid = |victime|$  (tout le groupe)

# Redéfinir le gestionnaire associé à un signal (POSIX)

Structure à utiliser pour décrire un gestionnaire

```
struct sigaction {
    void (*sa_handler)(int);           /* gestionnaire, interface simple */
    void (* sa_sigaction) (int, siginfo_t *, void *); /* gestionnaire, interface complète */
    sigset_t sa_mask;                 /* signaux à bloquer pendant le traitement */
    int sa_flags;                     /* options */
}
```

- ▶ **Gestionnaires particuliers** : SIG\_DFL : action par défaut ; SIG\_IGN : ignorer signal
- ▶ Deux types de gestionnaires
  - ▶ sa\_handler() connaît le numéro du signal
  - ▶ sa\_sigaction() a plus d'infos

Primitive à utiliser pour installer un nouveau gestionnaire

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

- ▶ Voir man sigaction pour les détails

Autre interface existante : ANSI C

- ▶ Peut-être un peu plus simple, mais bien moins puissante et pas thread-safe

## Exemple 1 : traitement d'une interruption du clavier

- ▶ Par défaut, Ctrl-C tue le processus ; pour survivre : il suffit de redéfinir le gestionnaire de SIGINT

test-int.c

```
#include <signal.h>
void handler(int sig) {                /* nouveau gestionnaire */
    printf("signal SIGINT reçu !\n");
    exit(0);
}
int main() {
    struct sigaction nvt,old;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = &handler;
    sigaction(SIGINT, &nvt, &old);    /*installe le gestionnaire*/
    pause ();                          /* attend un signal */
    printf("Ceci n'est jamais affiché.\n");;
}
```

```
$ ./test-int
[frappe de CTRL-C]
signal SIGINT reçu !
$
```

Exercice : modifier ce programme pour qu'il continue après un CTRL-C

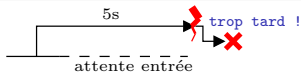
Note : il doit rester interruptible, *i.e.* on doit pouvoir le tuer d'un CTRL-C de plus

## Exemple 2 : temporisation

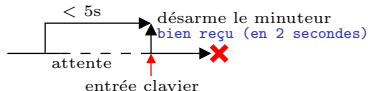
`unsigned int alarm(unsigned int nb_sec)`

- ▶ `nb_sec > 0` : demande l'envoi de SIGALRM après environ `nb_sec` secondes
- ▶ `nb_sec = 0` : annulation de toute demande précédente
- ▶ Retour : nombre de secondes restantes sur l'alarme précédente
- ▶ Attention, `sleep()` réalisé avec `alarm()` ⇒ mélange dangereux

```
#include <signal.h>
void handler(int sig) {
    printf("trop tard !\n");
    exit(1);
}
int main() {
    struct sigaction nvt,old;
    int reponse,restant;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = handler;
    sigaction(SIGALRM, &nvt, &old);
```



```
printf("Entrez un nombre avant 5 sec:");
alarm(5);
scanf("%d", &reponse);
restant = alarm(0);
printf("bien reçu (en %d secondes)\n",
        5 - restant);
exit (0);
}
```



## Exemple 3 : synchronisation père-fils

- ▶ Fin ou suspension d'un processus  $\Rightarrow$  SIGCHLD automatique à son père
- ▶ **Traitement par défaut** : ignorer ce signal
- ▶ **Application** : wait() pour éviter les zombies mangeurs de ressources  
C'est ce que fait le processus init (celui dont le pid est 1)

```
#include <signal.h>
void handler(int sig) {                               /* nouveau gestionnaire */
    pid_t pid;
    int statut;
    pid = waitpid(-1, &statut, 0); /* attend un fils quelconque */
    return;
}
int main() {
    struct sigaction nvt,old;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = &handler;
    sigaction(SIGCHLD, &nvt, &old); /* installe le gestionnaire */
    ... <création d'un certain nombre de fils> ...
    exit (0);
}
```



# Résumé de la fin du deuxième chapitre

- ▶ Quelques définitions
  - ▶ pgid : groupe de processus ; un par processus lancé par shell (et tous ses fils)
  - ▶ Travail d'avant-plan, d'arrière-plan : lancé avec ou sans &
- ▶ Communication par signaux
  - ▶ Envoyer un signal  $\langle NOM \rangle$  à  $\langle victime \rangle$  :
    - ▶ Langage commande : kill -NOM victime
    - ▶ API : kill(pid\_t victime, int sig)
      - victime > 0 : numéro du pid visé
      - victime = 0 : tous les process de ce groupe
      - victime = -1 : tous les process accessibles
      - victime < -1 : tous les process du groupe  $abs(victime)$
    - ▶ Astuce : envoyer le signal 9 à tous les processus : kill -9 -1
  - ▶ Changer le gestionnaire d'un signal : sigaction (en POSIX)
  - ▶ Exemples de signaux
    - ▶ SIGINT, SIGSTOP : Interaction avec le travail de premier plan (ctrl-c, ctrl-z)
    - ▶ SIGTERM, SIGKILL : demande de terminaison, fin brutale
    - ▶ SIGALARM : temporisation
    - ▶ SIGCHLD : relations père-fils

# Évolution récente : groupes de processus (cgroups)

- ▶ Souvent, les applications ou services sont composés de plusieurs processus
  - ▶ Navigateur web Chrome : un processus par onglet
  - ▶ Serveur web Apache : plusieurs processus pour traiter les différentes requêtes
- ▶ Des limitations par processus n'ont pas beaucoup de sens
- ▶ cgroups (*control groups*)
  - ▶ Groupes de processus sur lesquels s'appliquent diverses propriétés
  - ▶ Spécifique à Linux ( $\approx$  Solaris Zones, FreeBSD Jails)
  - ▶ Complémentaire des *namespaces* : limite l'accès et la visibilité des ressources pour un *cgroup* donnéContainers (Docker, LXC) = *cgroups* + *namespaces* + interface
- ▶ Également utilisé dans *systemd* pour isoler chaque service du système et chaque session utilisateur : `systemd-cgls`, `systemd-cgtop`