# Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Cité
L3 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2023-2024

# Signaux, suite et fin

# Comment ça marche ?

chaque processus a sa *table des signaux*, qui inclut (entre autres) :

- une table des signaux pendants, *i.e.* émis mais non encore délivrés ; c'est une *bitmap* – il n'y a ***pas de possibilité de décompte*** des occurrences d'un signal donné reçues
- une table des signaux masqués – cf plus loin
- une table des *handlers* : `SIG_IGN`, `SIG_DFL` ou un pointeur de fonction (nécessairement définie ***dans la zone de texte du processus***)

la table des signaux pendants est a priori incluse dans la table des processus, et mise à jour lors de l'envoi d'un signal (donc alors que le processus cible n'est pas actif), puis lors du traitement

à certaines occasions (non spécifiées, mais souvent lors de la bascule entre mode noyau et mode utilisateur), le processus actif prend connaissance des signaux pendants et en traite un (pas de critère de choix spécifié)

# Comportement vis-à-vis de `fork` et `exec`

Lors d'un clonage avec `fork`, sont hérités en particulier :

- la table des gestionnaires de signaux
- la table des signaux masqués

mais ***pas celle des signaux pendants*** : le nouveau processus n'en était pas la cible

# Comportement vis-à-vis de `fork` et `exec`

Lors d'un clonage avec `fork`, sont hérités en particulier :

- la table des gestionnaires de signaux
- la table des signaux masqués

mais ***pas celle des signaux pendants*** : le nouveau processus n'en était pas la cible

Lors d'un recouvrement avec `exec`, tout est écrasé à l'exception de

- la table des signaux pendants
- la table des signaux masqués *(oubli dans la version originale des slides)*
- la liste des signaux ignorés

mais les ***gestionnaires*** autres que `SIG_IGN` ou `SIG_DFL` sont écrasés : pour les signaux concernés, `SIG_DFL` est rétabli

## Que mettre dans un *handler* ?

le processus peut avoir été interrompu absolument n'importe quand, en plein milieu d'une opération complexe agissant sur des variables globales par exemple – voire des variables « cachées », comme les structures utilisées pour la gestion des zones allouées ou les entrées/sorties de haut-niveau

# Que mettre dans un *handler* ?

le processus peut avoir été interrompu absolument n'importe quand, en plein milieu d'une opération complexe agissant sur des variables globales par exemple – voire des variables « cachées », comme les structures utilisées pour la gestion des zones allouées ou les entrées/sorties de haut-niveau

on ne peut pas mettre n'importe quoi dans un *handler*, au risque d'interférer et de rendre ces structures incohérentes

# Que mettre dans un *handler* ?

le processus peut avoir été interrompu absolument n'importe quand, en plein milieu d'une opération complexe agissant sur des variables globales par exemple – voire des variables « cachées », comme les structures utilisées pour la gestion des zones allouées ou les entrées/sorties de haut-niveau

on ne peut pas mettre n'importe quoi dans un *handler*, au risque d'interférer et de rendre ces structures incohérentes

il faut se limiter aux fonctions dites *async signal-safe*, cf

```
man 7 signal-safety
```

# Que mettre dans un *handler* ?

le processus peut avoir été interrompu absolument n'importe quand, en plein milieu d'une opération complexe agissant sur des variables globales par exemple – voire des variables « cachées », comme les structures utilisées pour la gestion des zones allouées ou les entrées/sorties de haut-niveau

on ne peut pas mettre n'importe quoi dans un *handler*, au risque d'interférer et de rendre ces structures incohérentes

il faut se limiter aux fonctions dites *async signal-safe*, cf
`man 7 signal-safety`

***en particulier, ni `malloc`, ni `free`, ni `(d/f)printf` ne sont sûres***

# Que mettre dans un *handler* ?

il faut se limiter aux fonctions dites *async signal-safe*

*en particulier, ni `malloc`, ni `free`, ni `printf` ne sont sûres*

# Que mettre dans un *handler* ?

il faut se limiter aux fonctions dites *async signal-safe*

*en particulier, ni `malloc`, ni `free`, ni `printf` ne sont sûres*

une bonne pratique peut être de limiter le rôle du *handler* à la modification d'une variable globale dédiée de type `volatile sig_atomic_t`, pour déléguer le traitement réel au programme prinicipal

# Que mettre dans un *handler* ?

il faut se limiter aux fonctions dites *async signal-safe*

*en particulier, ni `malloc`, ni `free`, ni `printf` ne sont sûres*

une bonne pratique peut être de limiter le rôle du *handler* à la modification d'une variable globale dédiée de type `volatile sig_atomic_t`, pour déléguer le traitement réel au programme prinicipal

```
volatile sig_atomic_t signal_recu = 0;
void handler () { signal_recu = 1; }
int main () {
  ... /* mise en place du handler */
  while (1) {
    if (signal_recu == 1) { ... }
    ...
  }
}
```

# Appels systèmes interrompus

sauf si `sa_flags` inclut `SA_RESTART`, un appel système (bloquant) interrompu par la réception d'un signal ne reprend pas : il retourne -1 et `errno=EINTR`

# Appels systèmes interrompus

sauf si `sa_flags` inclut `SA_RESTART`, un appel système (bloquant) interrompu par la réception d'un signal ne reprend pas : il retourne -1 et `errno=EINTR`

il faut donc en tenir compte tout au long d'un programme susceptible de recevoir des signaux

# Appels systèmes interrompus

sauf si `sa_flags` inclut `SA_RESTART`, un appel système (bloquant) interrompu par la réception d'un signal ne reprend pas : il retourne -1 et `errno=EINTR`

il faut donc en tenir compte tout au long d'un programme susceptible de recevoir des signaux

Exemple :

```
do {
  rc = write(...);
} while(rc < 0 && errno == EINTR);
```

## Masquage de signaux

une solution pour améliorer la fiabilité des signaux en retardant le moment où un signal est délivré (pour éviter les sections critiques)

# Masquage de signaux

une solution pour améliorer la fiabilité des signaux en retardant le moment où un signal est délivré (pour éviter les sections critiques)

masque = ensemble de signaux (temporairement) masqués/bloqués

# Masquage de signaux

une solution pour améliorer la fiabilité des signaux en retardant le moment où un signal est délivré (pour éviter les sections critiques)

masque = ensemble de signaux (temporairement) masqués/bloqués

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- l'ancien masque est stocké dans `oldset` (si non `NULL`)
- si `set` est `NULL`, pas de changement, sinon le nouveau masque est déterminé à partir de `set` et `how`
- `how` vaut `SIG_BLOCK` (ajout au masque), `SIG_UNBLOCK` (suppression) ou `SIG_SETMASK` (remplacement du masque)

# Manipulation des ensembles

```c
/* initialisation */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

/* modification */
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

/* test d'appartenance */
int sigismember(const sigset_t *set, int signum);
```

# Masquage durant l'exécution d'un *handler*

sauf si `sa_flags` inclut `SA_NODEFER`, le signal en cours de gestion est masqué

`sa_mask` permet de définir un masque additionnel (*i.e.* en plus du masque défini par `sigprocmask`)

# Relâchement du masquage

attention à ce genre de situation (*à ne surtout pas reproduire ! !*) :

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
... /* section critique protégée des interruptions par le masquage */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();      /* pb si le signal attendu arrive avant la pause */
```

# Relâchement du masquage

attention à ce genre de situation (*à ne surtout pas reproduire ! !*) :

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
... /* section critique protégée des interruptions par le masquage */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();      /* pb si le signal attendu arrive avant la pause */
```

il est *indispensable* de procéder au relâchement + attente de manière *atomique* $\Longrightarrow$ il faut un appel système spécifique

# Relâchement du masquage

attention à ce genre de situation (*à ne surtout pas reproduire ! !*) :

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
... /* section critique protégée des interruptions par le masquage */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();      /* pb si le signal attendu arrive avant la pause */
```

il est ***indispensable*** de procéder au relâchement + attente de manière ***atomique*** $\Longrightarrow$ il faut un appel système spécifique

```
int sigsuspend(const sigset_t *mask);
```

- de manière ***atomique***, remplace *temporairement* le masque par `mask` et met le processus en attente
- (si le processus ne termine pas) replace l'ancien masque avant de retourner -1, avec `errno=EINTR`

# Récapitulatif concernant le masquage

pour modifier le masque

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

pour consulter les signaux pendants

```
int sigpending(sigset_t *set);
```

pour modifier le masque *et* attendre de manière atomique

```
int sigsuspend(const sigset_t *mask);
```

pour supprimer un signal pendant masqué (mais pas ignoré) – ou bloquer jusqu'à ce qu'un des signaux de l'ensemble soit pendant – sans exécuter le handler éventuel

```
int sigwait(const sigset_t *restrict set, int *restrict sig);
```