# Chapitre 2: Les signaux

## Cours 3

# Plan

1) **Prise en compte d'un signal (suite)**
2) **Attacher un Handler à un Signal**
3) **Attente d'un signal**
4) **Armer une temporisation**
5) **Signaux et Héritage**
6) **Signaux vs Interruptions**

# Plan

1) **Prise en compte d'un signal (suite)**
2) Attacher un Handler à un Signal
3) Attente d'un signal
4) Armer une temporisation
5) Signaux et Héritage
6) Signaux vs Interruptions

# Prise en compte d'un signal

**2) Exécuter l'action par défaut :**

Le SE associe à chaque signal une action par défaut; ces actions par défaut sont de 5 natures :

1. Abort, abondon ou terminaison du processus et génération d'un fichier core contenant son contexte d'exécution, ce fichier core est exploitable par l'outil débogueur: SIGFPE, SIGQUIT… ;
2. Exit (terminaison du processus sans génération d'un fichier core) : SIGINT, SIGKILL, SIGUSR1, SIGUSR2 ;
3. Ignore (le signal est ignoré) : SIGCHLD ;
4. Stop (suspension du processus : passage à l'état bloqué) : SIGSTOP ;
5. Continue (reprendre l'exécution si le processus est suspendu sinon le signal est ignoré) : SIGCONT.

# Prise en compte d'un signal

**3) Exécuter une procédure spécifique:**

Le processus ne peut pas modifier l'action par défaut de SIGKILL et SIGSTOP par contre pour des signaux telles que SIGINT, SIGUSR1, SIGUSR2..., l'action par défaut peut être modifié, pour cela le processus doit associer au signal une autre procédure de gestion c'est le **Handler**. À la fin de l'exécution de la procédure le processus poursuit son exécution.

# Prise en compte d'un signal

**Remarques:**

- SIGINT est généré par la touche d'interruption Ctrl+C .Par défaut, ce signal arrête le processus. Le processus peut associer à ce signal un autre gestionnaire de signal.
- On peut associer un même gestionnaire (Handler) à des signaux différents.
- Les signaux SIGKILL, SIGSTOP ne peuvent être ni capturés (modifier leur actions par défaut), ni bloqués, ni ignorés.

# Plan

1) Prise en compte d’un signal (suite)
2) **Attacher un Handler à un Signal**
3) Attente d'un signal
4) Armer une temporisation
5) Signaux et Héritage
6) Signaux vs Interruptions

# Attacher un Handler à un Signal

La primitive qui permet d'attacher un Handler à un signal est l'appel système signal(), son prototype est :

```
#include <signal.h>
Void(signal(int sig , void(func)(int)))(int);
```

- `sig`: représente le numéro ou le nom du signal.
- `func`: c'est la fonction à exécuter, à l'arrivé du signal.

L'argument func peut prendre trois valeurs distinctes:

- func = SIGDFL, alors l'action par défaut est attaché au signal sig.
- func = SIGIGN, alors le signal sig est ignoré.
- func est un pointeur vers une procédure Handler

En cas d'échec, la fonction signal() renvoie la valeur SIG_ERR.

# Plan

1) Prise en compte d’un signal (suite)
2) Attacher un Handler à un Signal
3) **Attente d'un signal**
4) Armer une temporisation
5) Signaux et Héritage
6) Signaux vs Interruptions

# Attente d'un signal

L'appel système `pause` suspend l'appelant jusqu'au prochain signal. Son prototype est :

```
#include <unistd.h>

int pause (void);
```

L'appel système `sleep(v)` suspend l'appelant jusqu'au prochain signal ou l'expiration du délai (v secondes). Son prototype est :

```
#include <unistd.h>

void sleep(int);
```

# Attacher un Handler à un Signal

**Exemple 2: Que fait le programme signal2.c ?**

```
void bien_reçu() {
        printf("Signal reçu par le fils\n");
        }
void main(){
int pid=fork();
if(pid>0){
        sleep(1); //Pourquoi cette instruction ?
        printf("processus père=%d\n", getpid());
        kill(pid,SIGUSR1);
        printf("Envoie du signal au fils\n");
        wait(NULL); //Que va-t-il se passer si on enlève wait ?
        }
else{
        signal(SIGUSR1,bien_reçu);
        pause(); //Si on enlève pause, que va-t-il se passer ?
        printf("J'ai terminé\n");
        }
}
```

# Attente d'un signal

**Exemple 3: Que fait le programme signal3.c**

```
void fin_attente(){
        printf("Signal reçu");
        }
main(){
if(signal(SIGINT,fin_attente)== SIG_ERR)
        {
        perror("signal");
        exit(1);
        }
while(1)
        pause();
}
```

# Plan

# Armer une temporisation

La primitive `alarm` permet à un processus d'armer une temporisation. À l'issue de cette temporisation le signal SIGALRM est délivré au processus. Le comportement par défaut est d'arrêter l'exécution du processus. Le prototype de la fonction est :

```
#include <unistd.h>
Unsigned int alarm(unsigned int nb_sec);
```

L'opération `alarm(0)` annule une temporisation précédemment armée.

# Armer une temporisation

**Exemple 4: Que fait le programme signal4.c**

```
void fin_attente(int sig){
        if(sig == SIGINT)
                printf("Signal reçu, arrêt de la pause");
        else{
                printf("Trop tard, je n'attends plus");
                exit(0);}
        }
void main(){
    int x;
    signal(SIGINT,fin_attente);
    signal(SIGALRM,fin_attente);

    pause();
    alarm(5);

    printf("Saisir une valeur, puis taper Entrée SVP \n");
    scanf("%d",&x);
    printf("Vous avez saisi %d", x);
}
```

# Plan

1) Prise en compte d’un signal (suite)
2) Attacher un Handler à un Signal
3) Attente d'un signal
4) Armer une temporisation
5) **Signaux et Héritage**
6) Signaux vs Interruptions

# Signaux et Héritage

- Un processus fils n'hérite pas des signaux pendants de son père, mais il hérite les gestionnaires (Handler).
- En cas de recouvrement du code hérité du père (le fils exécute la fonction exec…), les gestionnaires par défaut sont réinstallés pour tous les signaux du fils.

# Plan

1) Prise en compte d’un signal (suite)
2) Attacher un Handler à un Signal
3) Attente d'un signal
4) Armer une temporisation
5) Signaux et Héritage
6) **Signaux vs Interruptions**

# Signaux vs Interruptions

Une interruption est soit:

- Une trappe (déroutement); Synchrone
- Appel Système; Synchrone
- Matérielle; Asynchrone