

***Interruptions  
Et signaux***

### *Plan*

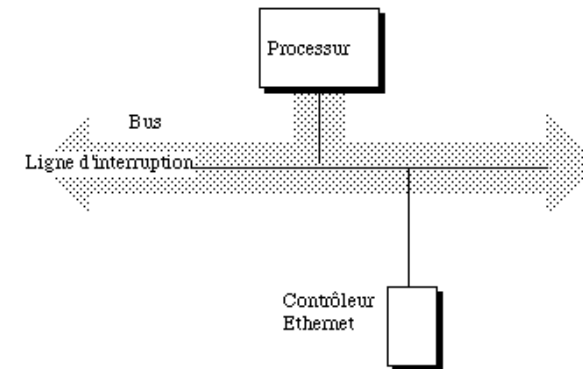
- **Introduction**
- Interruptions
- Signaux
  - Emission
  - Réception
  - Traitement
  - Exemples
    - SIGSEGV
    - SIGALRM
    - SIGCHLD

### *Introduction*

- Nous présentons ici les moyens qui permettent de gérer les événements qui arrivent **parallèlement** au déroulement d'un processus.
- Ces événements ont différentes origines:
  - matérielles : arrivée d'une trame réseau, saisie clavier, ...
  - logicielles : erreur d'adressage, dépassement de capacité de la pile, ...
- Ces événements peuvent être destinés au processus lui-même,
- Ces événements peuvent être destinés à un autre processus.

### *Introduction*

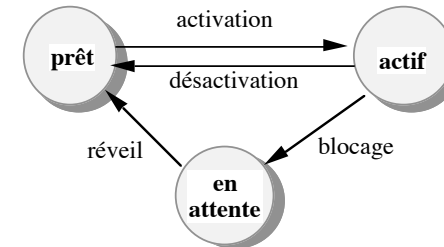
- La prise en compte matérielle de ces événements est faite sous forme de variation d'état d'une des lignes du bus, appelée ligne d'interruption :



- Le processeur détecte cette variation, le système d'exploitation la prend en charge et la répercute vers le processus concerné,
- Sous Unix l'outil utilisé pour « remonter » les interruptions vers un processus s'appelle *signal*

### *Retour sur les transitions entre états*

- Rappel : graphe d'état des processus :



- Le processeur exécute le programme associé au processus actif :
  - sur une machine monoprocesseur, cadre de notre étude, un seul programme est exécuté à la fois,
- Question : comment va se faire la transition vers un autre état ?
  - on ne voit pas comment : le processeur exécute un programme, le compteur ordinal va donc rester à l'intérieur de ce programme,
  - il faut donc introduire un mécanisme matériel qui indique au processeur d'arrêter le traitement courant,
  - ce mécanisme s'appelle une **interruption**,

### *Plan*

- Introduction
- **Interruptions**
- Signaux
  - Emission
  - Réception
  - Traitement
  - Exemples
    - SIGFPE
    - SIGALRM
    - SIGCHLD

### ***Interruptions: émission***

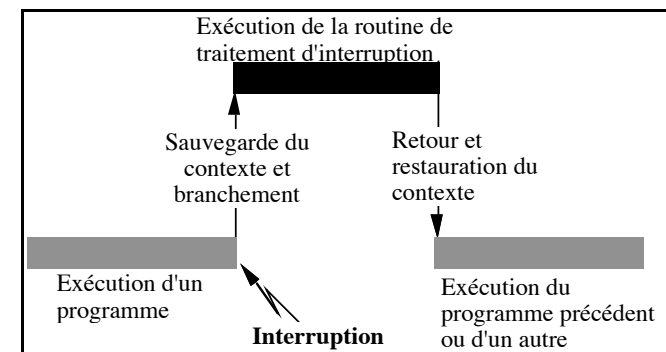
- **Qui** peut émettre une interruption ?
  - un périphérique, pour indiquer une terminaison d'E/S, qu'elle soit correcte ou non,
  - l'horloge, pour indiquer l'échéance d'une alarme,
  - etc,
  
- **Comment** émettre une interruption ?
  - pour simplifier: on envoie une interruption sous forme d'un changement d'état sur une des « pattes » du processeur ; ce dernier vérifiera, avant chaque exécution, si cette ligne a changé d'état,

### *Interruptions : fonctionnement*

- Modification **de principe** du cycle de fonctionnement du processeur :
  - avant d'exécuter une instruction, vérifier si une interruption n'est pas arrivée,
  - si oui, la traiter, c'est-à-dire aller au point 2 du paragraphe suivant,
  
- Si une interruption arrive pendant l'exécution d'une instruction :
  - 1 l'instruction en cours se termine (**ATOMICITE** d'une instruction),
  - 2 l'adresse de l'instruction suivante et le contenu des registres sont **sauvegardés** dans une pile spécifique : la pile d'interruption (*interrupt stack*),
  - 3 les interruptions de niveau inférieur ou égal sont masquées,
  - 4 le processeur consulte une table qui contient l'adresse de la **procédure** à exécuter suivant le type d'interruption qu'il a reçu,

### *Interruptions : traitement*

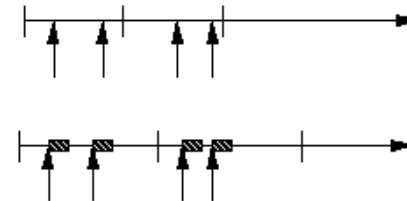
- Le schéma suivant indique comment le processeur traite une interruption.



- Conséquence capitale pour l'ordonnancement :
  - les interruptions sont plus prioritaires que n'importe lequel des processus,

### *Interruptions et ordonnancement*

- Le schéma suivant indique la prise en compte des interruptions dans un scénario d'ordonnancement du type *round robin* :



- le séquençage de l'ordonnancement reste inchangé, mais l'attribution de  $n$  quantum de durée  $q$  se fera dans une fenêtre de temps de taille **supérieure** à  $(n \cdot q)$ .
  - La taille de la fenêtre sera  $(n \cdot q)$  augmenté du temps de traitement de toutes les interruptions reçues,
- Remarques :
    - L'exécution du programme traitant l'interruption peut conduire à l'arrêt du travail courant pour exécuter une **autre tâche** (si celle-ci est plus prioritaire que le travail courant),
    - La fin du quantum est provoquée par une interruption,
    - Le changement de contexte ne se fait pas en temps nul, comme sur les schémas de principe,

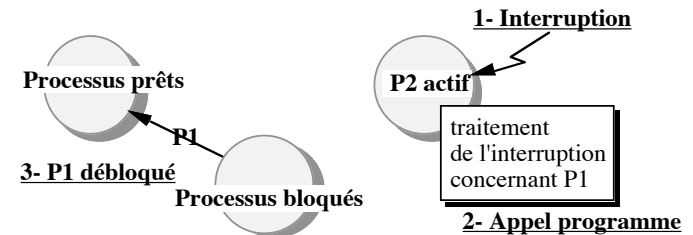
### *Interruptions : gestion par le système*

- Gestion de l'interruption par le système :
  - L'appel du programme traitant l'interruption est géré comme un appel de procédure classique.
  - **Mais** cet appel se fait généralement pendant l'exécution d'un AUTRE processus, par exemple :
    - l'interruption concernant un processus bloqué arrive pendant qu'un autre est actif !
- Les interruptions permettent de traiter plusieurs activités « en même temps »

***Interruptions et  
Grphe d'état des processus***

• Exemple de scénario :

- 1 une interruption (fin de lecture disque) concernant le processus P1 arrive pendant que le processus P2 est actif (1),
- 2 la procédure qui est associée à cette interruption est exécutée,
- 3 ce traitement provoque le passage de P1 de l'état bloqué à l'état prêt (3), si la priorité de P1 est supérieure à celle de P2, P2 devient prêt et P1 actif,



### *Plan*

- Introduction
- Interruptions
- **Signaux**
  - Emission
  - Réception
  - Traitement
  - Exemples
    - SIGFPE
    - SIGALRM
    - SIGCHLD

## Interruptions et signaux

---

Voici un extrait du fichier `/usr/include/sys/signal.h` :

```
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction(not reset when caught)*/
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT 6 /* IOT instruction */
#define SIGABRT 6 /*used by abort,replace SIGIOT in the future */
#define SIGEMT 7 /* EMT instruction */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD 18 /* child status change */
#define SIGCHLD 18 /* child status change alias (POSIX) */
#define SIGPWR 19 /* power-fail restart */
#define SIGWINCH 20 /* window size change */
#define SIGURG 21 /* urgent socket condition */
#define SIGPOLL 22 /* pollable event occured */
#define SIGIO SIGPOLL /* socket I/O possible (SIGPOLL alias) */
#define SIGSTOP 23 /* stop (cannot be caught or ignored) */
#define SIGTSTP 24 /* user stop requested from tty */
#define SIGCONT 25 /* stopped process has been continued */
#define SIGTTIN 26 /* background tty read attempted */
#define SIGTTOU 27 /* background tty write attempted */
#define SIGVTALRM 28 /* virtual timer expired */
#define SIGPROF 29 /* profiling timer expired */
#define SIGXCPU 30 /* exceeded cpu limit */
#define SIGXFSZ 31 /* exceeded file size limit */
#define SIGWAITING 32 /* process's lwps are blocked */
#define SIGLWP 33 /* special signal used by thread library */
#define SIGFREEZE 34 /* special signal used by CPR */
#define SIGTHAW 35 /* special signal used by CPR */
#define SIGCANCEL 36 /*thread cancel signal used by libthread */
#define _SIGRTMIN 37 /*first(highest-priority) realtime signal*/
#define _SIGRTMAX 44 /*last (lowest-priority) realtime signal */
```

## Interruptions et signaux

---

### ***Les signaux : présentation***

#### • Définition

un signal sert à indiquer l'occurrence d'un événement, cet événement peut être d'origine :

- Interne, c'est-à-dire provoqué par le processus lui-même :
    - Involontairement : division par zéro, dépassement de capacité de la pile, ...
    - volontairement : gestion d'une alarme, gestion d'E/S, ...
  - Externe, provoqué par :
    - Un autre processus : fin d'un processus fils, ...
    - Le système : fin d'E/S, ...
- Dans le cas des événements d'origine matérielle, les signaux sont le moyen dont dispose le système pour remonter les interruptions du matériel vers les processus concernés
- L'ensemble des signaux disponibles est décrit dans `signal.h`

## Interruptions et signaux

---

### Remarques :

Un seul bit indique l'occurrence d'un signal de type donné. Si un signal arrive avant que le traitement du précédent de même type soit terminé, il est perdu !

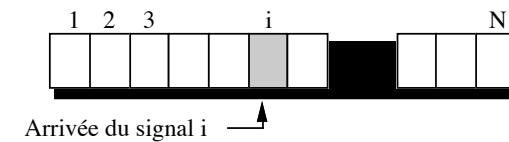
Seuls les processus du « super-utilisateur (root) » et les processus appartenant au même utilisateur peuvent envoyer des signaux vers un processus donné.

## Interruptions et signaux

---

### *Les signaux :émission*

- **Qui** peut émettre un signal ?
  - le noyau, à la suite d'une division par zéro, d'une tentative d'exécution d'une instruction interdite, à la fin d'un processus (émission de SIGCHLD), ...
  - un utilisateur : en utilisant le clavier (<CTRL C>), par exemple) ou la commande `kill` du shell,
  - un processus : appel à la fonction `kill`, à la fonction `alarm`,
- L'envoi d'un signal à un processus est matérialisé sous forme d'un bit positionné par le système dans un tableau associé à ce processus :



- Remarque :

un émetteur ne peut pas savoir si le processus destinataire a **reçu** ou non le signal qu'il a émis.

### ***Les signaux : Outils d'émission***

- Utilisation de la commande kill du shell :

```
kill -nom-du-signal pid  
kill -numéro-du-signal pid
```

kill -INT 567	Envoi du signal SIGINT au processus 567.
kill -9 7890	Envoi du signal SIGKILL au processus 7890.

- Utilisation de la fonction kill en C:

```
#include <signal.h>  
int kill (pid_t pid, int sig);
```

kill (pid, SIGINT)	Envoi du signal SIGINT au processus pid.
kill (getppid(), SIGUSR1)	Envoi du signal SIGUSR1 au processus père.

- Utilisation de la fonction alarm en C:

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

alarm (2)	Le processus courant recevra le signal SIGALRM dans deux secondes
-----------	---

### Cas des processus bloqués :

Un processus bloqué dans un appel système (*system call*) ne va pas se comporter de la même façon suivant qu'il a fait appel à un :

- appel système standard
- appel système dit "slow system call".

Exemple de *slow system calls* (les appels système qualifiés de *slow* ne sont pas les mêmes sur tous le système Unix...) :

- Saisie depuis un clavier ,
- Lecture sur un fichier tube (pipe),
- Attente d'une trame TCP.

Donc la sortie d'un appel bloquant peut être provoquée par un événement **différent** de celui attendu parce que celui qui arrive est plus prioritaire que celui attendu. L'exemple suivant illustre ce cas : SIGCHLD est plus prioritaire que l'E/S clavier.

Un appel système dont on sort en recevant un événement différent de celui qui était attendu retourne -1 et la variable `errno` est positionnée à `EINTR` (cf. exemple suivant).

### Réception d'un signal (1)

- La gestion de l'occurrence d'un signal dépend de l'état dans lequel se trouve le processus destinataire à cet instant. On distingue donc les cas :
  - Réception d'un signal par un processus actif,
  - Réception d'un signal par un processus bloqué,
- **1- Réception d'un signal par un processus actif :**
  - si le processus exécute un programme utilisateur : traitement immédiat du signal reçu,
  - s'il se trouve dans une fonction du système (ou *system call*) : le traitement du signal est différé jusqu'à ce qu'il revienne en mode *user*, c'est à dire lorsqu'il sort de cette fonction,

## Interruptions et signaux

### Exemple de slow system call :

```
int main (void) {
    void fils (void);
    void pere (void);
    int Ret_Fork;
    Ret_Fork = fork ();
    if (Ret_Fork == 0) fils ();
    pere();
    return 0;
}

void fils (void) {
    long i;
    for (i=0; i<100000000; i++);
    printf (" Pid %d fils de %d : fin \n", (int)getpid(), (int)getppid());
    exit (5);
}

void pere (void){
    char Carac;
    int Ret_Read;
    void fonc (void);
    signal (SIGCHLD, fonc);

    printf ("pere \n");
    Ret_Read = read(0, &Carac, 1);
    if ( Ret_Read == -1 && errno == EINTR) printf ("EINTR\n");
    printf ("Carac = %c\n", Carac);
}

void fonc (int NumSig) {
    printf (" fonc : Pid %d a recu %d\n", (int)getpid(), (int) NumSig );
    signal (NumSig, fonc);
}
```

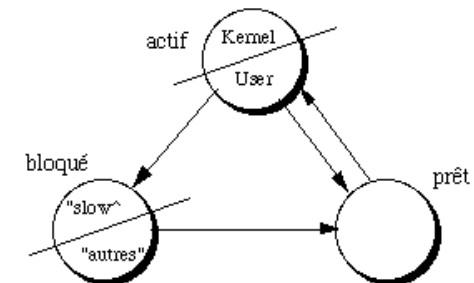
Résultat :

```
-----
pere
Pid 14770 fils de 14769 : fin
fonc : Pid 14769 a recu 18
EINTR
Carac = !
```

## Interruptions et signaux

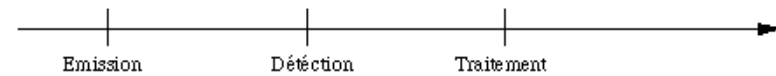
### Réception d'un signal (2)

- 2- Le processus destinataire est bloqué (donc dans un *system call*) :
  - Dans la plupart des cas, le signal est traité en sortant de l'appel système bloquant,
  - Dans les autres cas (*slow system calls*) le signal est traité dès réception et le processus **sort** de la fonction bloquante, il passe donc à l'état **prêt**. La valeur de retour de la fonction est positionnée à -1 et errno à EINTR. (Exemple : arrivée de SIGCHLD pendant qu'un processus est bloqué sur une lecture réseau ou clavier)



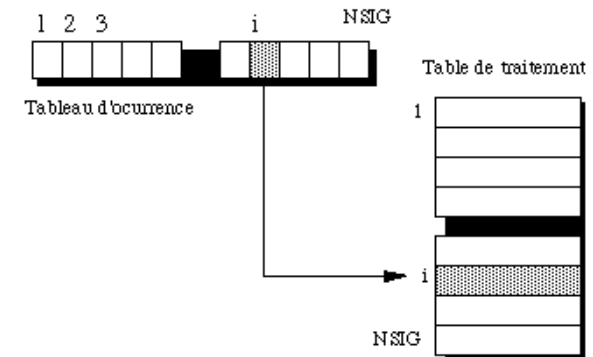
### *De la réception au traitement*

- Une fois détectée son occurrence, **comment traiter** un signal ? Pour chaque signal, un processus peut :
  - ignorer ce signal, si le système l'y autorise,
  - se contenter du traitement par défaut (en général fin du processus),
  - exécuter une fonction spécifique, si le système l'y autorise,



### *De la réception au traitement*

- Pour traiter le signal  $i$  qui a été délivré à un processus de pid  $P$ , le système consulte un tableau associé à  $P$  qui indique l'attitude de  $P$  vis à vis de chacun des signaux :
  - ce tableau comporte une entrée par signal, chacune initialisée à la valeur SIG\_DFL (traitement par défaut) à la création du processus,
  - le processus peut modifier les entrées de ce tableau pour **indiquer le comportement** qu'il adoptera s'il reçoit le signal correspondant



- REMARQUE : Un processus hérite, par copie, de la table construite par son père.

### ***Traitement des signaux***

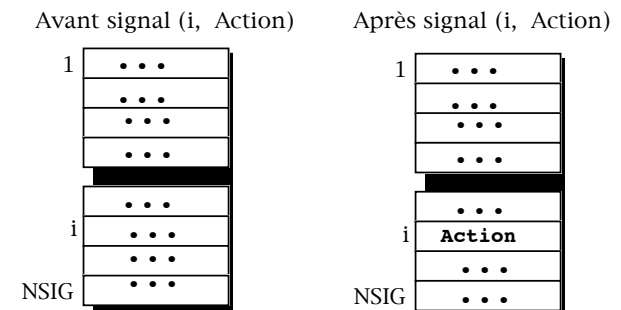
- Un processus peut adopter les comportements suivants lors de la réception d'un signal :
  - se satisfaire du traitement standard,
  - ignorer ce signal (si le système le permet),
  - lui associer un traitement spécifique (si le système le permet),
- Traitement standard en réception d'un signal :
  - dans la plupart des cas, fin du processus destinataire avec, ou sans, production d'un fichier appelé `core` (dump mémoire)
  - cas particuliers, citons :
    - `SIGCHLD`, émis par le `exit` d'un processus fils, pour indiquer sa fin au processus père.

### **Modifier le Traitement par défaut**

- Pour ce faire, un processus utilise la fonction

**signal**(Num\_Sig, Action)

- Elle **modifie** l'entrée numéro Num\_Sig dans le tableau. L'entrée Num\_Sig indique ce que doit faire le processus si il reçoit le signal de numéro Num\_Sig.



### **Traitement d'un signal : Les 3 options**

**1• Ignorer le signal :** `signal (Num_Sig, SIG_IGN)`

effet : *mise à la valeur « ignorer » de l'entrée associée au signal Num\_Sig dans la tableau,*

**2• (re)prendre le traitement par défaut** (en général `exit`) :

`signal (Num_Sig, SIG_DFL)`

effet : *mise à la valeur « traitement par défaut » de l'entrée associée au signal Num\_Sig dans la tableau,*

**3• adopter un traitement spécifique** (défini dans fonction)

`signal (Num_Sig, fonction)`

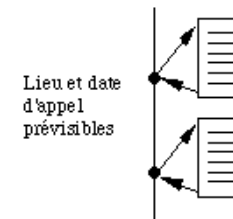
effet : *l'adresse de la fonction fonction est rangée dans l'entrée associée au signal Num\_Sig dans la tableau.*

**Remarque :** `signal` **n'émet pas de signal** (la fonction qui émet un signal est la fonction `kill ...`),

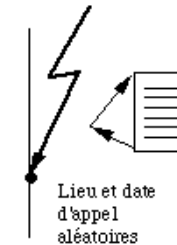
### **Traitement d'un signal : Exécution de la fonction associée**

- Comme on l'a vu, si une fonction est associée à un signal, elle est exécutée lorsque le système constate l'arrivée du signal,
- Différence entre un appel de fonction classique (synchrone) et un appel à celle qui traite un signal (asynchrone) :

Appel de fonctions classique



Appel de la fonction associée à un signal



- L'adresse de retour de la fonction de traitement du signal est celle de l'instruction où a été détecté le signal.

### ***Ignorer un signal : exemple***

- Voici un exemple de programme exécuté par un processus qui veut ignorer tous les signaux :

```
#include <signal.h>
#include <stdio.h>
int main(void){
    short int  Num_Signal;
    long      Ret_Sig;

    /*
     La fonction signal indique le comportement à adopter
     si le signal numéro Num_Sig arrive.
     Contrairement à son nom, elle n'envoie pas de signal !
     Si on n'a pas le droit d'ignorer le signal Num_Sig,
     la fonction signal renvoie -1.
    */

    for (Num_Signal = 1; Num_Signal < NSIG ; Num_Signal ++){
        Val_Sig= signal(Num_Signal, SIG_IGN);
        printf("Valeur renvoyée pour: %d %d\n",Num_Signal, Val_Sig);
    }
    ...
}
```

### *Traitement spécifique: exemple*

- Si un signal lui arrive, ce programme exécute une fonction de traitement :

```
#include <stdio.h>
#include <signal.h>

int main (void){

    void fonc (int Num);
    int NumSig;

    /* Si un signal arrive , on appelle fonc */
    for (NumSig = 1 ; NumSig <= NSIG ; NumSig++)
        signal (NumSig, fonc);

    while (1);
}

/* fonction de traitement des signaux */
void fonc (int Num){
    printf("Recu signal %d\n", Num);
    signal (Num, fonc);
}
```

- Remarque : dans la fonction de traitement, on **réarme** ce traitement. En effet certains système Unix repositionnent le traitement par défaut après avoir effectué le traitement spécifique.

### **Exemple : le signal SIGSEGV**

- Le signal SIGSEGV est émis lors d'un accès à la mémoire interdit.

- Soit le programme :

```
#include <stdio.h>
int i, Tab[100];

int main(void) {
    ...
    /* La fonction Mise_A_Jour calcule des
       valeurs de i et met le tableau Tab à jour */
    Mise_A_Jour() ;
    ...
}
```

- Si une des valeurs calculées pour i provoque une erreur mémoire, le processus est arrêté et on reçoit le message suivant (l'exécutable s'appelle a.out) :

```
zsh: segmentation fault ./a.out
```

### *Ignorer le signal*

- Modification du programme pour ignorer ce signal :

```
int i, Tab[100];  
int main(void) {  
    signal(SIGSEGV, SIG_IGN);  
    ...  
    /* Cette fonction calcule des valeurs de i  
       et met le tableau Tab à jour */  
    Mise_A_Jour();  
    ...  
}
```

- Inconvénient : on n'est pas averti si l'un des valeurs calculées est erronée!

### *Traitement spécifique d'un signal*

- Modification du programme pour adopter un traitement spécifique, c'est à dire l'exécution d'une fonction donnée par l'utilisateur . Cette fonction s'appelle **Traite\_Sig**.

```
int i, Tab[100];

/***** main *****/
int main(void) {
    void Traite_Sig();

    signal(SIGSEGV, Traite_Sig);
    ...
    /* Cette fonction calcule des valeurs de i
       et met le tableau Tab à jour */
    Mise_A_Jour() ;
    ...

}

/***** traitement du signal *****/
void Traite_Sig (int Num_Sig ){
    printf("Erreur sur adresse : %x, i = %d\n", (int)&Tab[i], i);
    signal(Num_Sig, Traite_Sig);
}
}
```

- Inconvénient : on continue après l'instruction qui a provoqué l'erreur. Il pourrait être plus intéressant de prévoir un point de reprise en cas d'erreur, c'est ce qu'on va voir.

### ***Traitement spécifique d'un signal (2)***

- On modifie le programme précédent pour y introduire un point de reprise.
  - Ce point est défini par la fonction `setjmp`
  - Après exécution de la fonction de traitement du signal, on reviendra au point de reprise en utilisant `longjmp` .

```
#include <setjmp.h>
int i, Tab[100];
jmp_buf contexte;

/***** main *****/
int main(void) {
    int Retour;
    void Traite_Sig();
    signal(SIGSEGV, Traite_Sig);
    ...
    /* point de reprise */
    Retour = setjmp (contexte);
    ...
    /* Cette fonction calcule des valeurs de i
    et met le tableau Tab à jour */
    Mise_A_Jour() ;
    ...
}
/***** traitement du signal *****/
void Traite_Sig (int Num_Sig ){
    printf("Erreur sur adresse : %x, i = %d\n", (int)&Tab[i], i);
    signal(Num_Sig, Traite_Sig);
    longjmp(contexte, Num_Sig);
}
}
```

**Commentaire:** `longjmp` va extraire le contexte rangé dans `contexte` et le restaurer. Ainsi, après avoir exécuté `longjmp` le programme retourne au `setjmp` qui avait sauvé le contexte.

### *Problèmes d'implémentation*

- Sur certains systèmes UNIX (famille System V), après exécution de la fonction spécifique définie par l'utilisateur, l'option par défaut est rétablie. Pour conserver l'option spécifique il faut réarmer le traitement dans la fonction de traitement.

- Exemple :

```
/****** traitement du signal SIGFPE *****/
void Traite_FPE (int NumSig) {
    printf("Pid %d recoit signal %d\n", getpid(), NumSig);
    signal(SIGFPE, Traite_FPE);
    longjmp(Contexte, 1);
}
```

- Remarque :

le paramètre de type int passé à la fonction qui gère un signal est mis sur la pile par le système. Il contient le numéro du signal reçu.

Ce type d'événement s'appelle une alarme ou un *timer*

### ***Le signal SIGALRM***

- Cet exemple va montrer les problèmes que peut poser la gestion du temps dans un système d'exploitation.
- On va utiliser la fonction `alarm(t)` qui demande au système d'envoyer au processus courant le signal SIGALRM dans **au moins t** secondes.
- Le programme suivant reçoit SIGALRM toutes les 5 secondes, il exécute alors la fonction `Traite_Alarme`

```
#define MAX (1024*1024*1024)
#define ALARME 5
unsigned long  Compteur ;

int main(void){
    void Traite_Alarme (int Signal);
    signal (SIGALRM, Traite_Alarme);
    alarm(ALARME);

    for (Compteur=0; Compteur< MAX; Compteur++);

    return 0;
}

/*****/
void Traite_Alarme(Signal){
    static int Nb_Alarmes = 1;
    printf ("Alarme num %d Compteur %ld\n", Nb_Alarmes, Compteur);
    signal (Signal, Traite_Alarme);
    Nb_Alarmes = Nb_Alarmes + 1;
    alarm(ALARME);
}
```

### ***SIGALRM Exemple – suite 1***

- Résultats sur une machine dotée d'un processeur cadencé à 800 Mhz :

```
Alarme num 1 Compteur 114834311
Alarme num 2 Compteur 232834363
Alarme num 3 Compteur 350383937
Alarme num 4 Compteur 463890247
Alarme num 5 Compteur 577742244
```

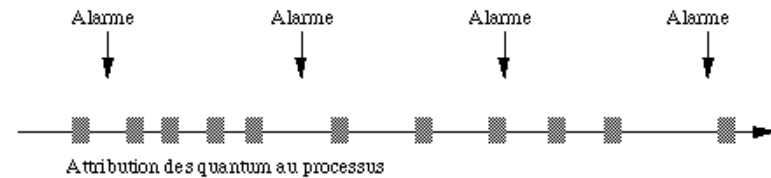
- Donc, après 5 alarmes, c'est à dire 25 secondes, on a exécuté environ  $577 \cdot 10^6$  incréments alors qu'on pensait exécuter environ  $25 \cdot (800 \cdot 10^6)$  instructions, le processeur étant cadencé à 800 MHz !
- D'où vient cette différence ? A cet instant la commande ps donne ceci pour le programme étudié :

UID	PID	PPID	CPU	PRI	NI	VSZ	STAT	TIME	COMMAND
501	775	769	0	9	5	27448	RN	0:10.16	prog

- en fait, le processus a consommé seulement environ 10 secondes de temps cpu pendant ces 25 secondes. Pourquoi ? On va le voir dans ce qui suit.

### **SIGALRM Exemple – suite 2**

- Dans une fenêtre de temps de 25 secondes, le processus n'est **actif** que lorsque l'ordonnanceur lui accorde le quantum, comme l'indique le schéma suivant:



- Ceci explique pourquoi le processus n'a consommé que 10 secondes de temps cpu pendant 25 secondes de temps écoulé,
- Mais on aurait donc du exécuter :  $10 \cdot 800 \cdot 10^6$  instructions au lieu des  $577 \cdot 10^6$  incréments comptés,
- Pourquoi cette différence ?
  - une instruction de langage de haut niveau correspond à **plusieurs instructions** machine,
  - les changements de contexte ne se font pas en temps nul !

### ***SIGALRM Exemple – suite 3***

- Commentaires généraux sur l'exemple précédent :
  - différence entre temps de service et temps d'exécution (d'où le temps de réponse important sur les machines chargées),
  - en compilant avec des options d'optimisation, on peut gagner un facteur 2, mais il faut garder à l'esprit que les performances annoncées pour un processeur concernent les instructions machines,
  - les traces peuvent perturber l'application : au lieu de tracer avec printf, il faudrait ranger les traces dans un tableau affiché en fin de programme,
  - on a vu ici les problèmes que peut poser la gestion du temps dans les systèmes temps partagé :
    - on ne peut pas prédire quand un processus sera actif,
    - une alarme pour la date D est délivrée **au plus tôt à cette date D,**

### ***Le signal SIGCHLD***

- On rappelle que SIGCHLD est émis par un processus lorsqu'il fait appel à `exit`. Le processus passe alors dans un état transitoire appelé *zombie*, en attendant de recevoir un acquittement de son père. Cet acquittement peut se faire de plusieurs façons, en général par un appel à `wait`.
- On va montrer ici les effets de cet état transitoire.

### *Le signal SIGCHLD : exemple*

• Exemple :

```
int main (void){
    ...
    ret_fork = fork ();
    if (ret_fork == 0) fils ();
    pere();
    return 0;
}

/***** ce que fait le fils *****/
void fils (void){
    ...
    printf ("Pid %d fils de %d : debut\n", getpid(), getppid());
    ...
    printf ("Pid %d fils de %d : fin\n", getpid(), getppid());
    exit (5);
}

/***** ce que fait le pere *****/
void pere (void){
    ...
    signal (SIGCHLD, fonc);
    while (1) ;
}

/***** traitement de SIGCHLD *****/
void fonc (int NumSig){
    printf ("fonc : Pid %d a recu %d\n", getpid(), NumSig);
}
```

### ***SIGCHLD (suite)***

- La trace de l'exécution est la suivante :

```
Pid 943 fils de 942 : debut
Pid 943 fils de 942 : fin
fonc : Pid 942 a reçu 20
ici le shell ne reprend pas la main (on ne voit pas l'invite)
```

- La commande ps donne ceci pour les deux processus précédents :

```
942 45.7 0.2 27448 808 p1 R 5:42PM 0:03.81
943 0.0 0.0 0 0 p1 Z 1Jan70 0:00.00
```

- Que s'est-il passé ?
  - création du processus 943 par le processus 942,
  - exécutions entrelacées de 942 et 943, qui passent alternativement de l'état actif à l'état prêt, suivant l'attribution du quantum,
  - 943 fait exit, envoie donc SIGCHLD à 942 et passe dans l'état Z (zombie) en attendant l'acquittement de ce signal,
  - 942 reçoit le signal, exécute fonc et continue son activité, **sans avoir acquitté** (pas de wait) le SIGCHLD émis par son fils qui reste zombie. Il le restera jusqu'à ce que son père se termine.

