Cours de S.E. – les Signaux

A. Dragut

Univ. Aix-Marseille, IUT Aix en Provence

2012

Plan

- Généralités
  - notion
  - noms et numéros
  - kill()
  - effets de l'arrivée des signaux
- Dynamique des signaux
  - sources des signaux
  - réception des signaux
  - déroutement
  - interruption, réentrance
  - signaux delivrés, bloqués, pendants
- Approfondissement
  - masques pour les signaux
  - bloquer : sigprocmask(), sigaction(), sigpending()
  - contrôle avancé : zones sensibles du programme, atomicité
- Multiplexage entrées-sorties
  - SIGALRM, minuterie
  - select()

Récapitulatif

- les processus
  - traitent (lisent, modifient, calculent) des données,
  - lues depuis et écrites sur des (descripteurs de) fichiers.
- les processus communiquent à l'aide des IPC
- les processus sont pilotés à l'aide des signaux.
- ce cours — SIGNAUX.
- paradigme : téléphone

Notion de signal

- Les signaux : des mécanismes du SE pour annoncer aux processus de manière asynchrone la production d'événements.
- asynchrone : entre l'émission du signal et sa réception il peut s'écouler un temps arbitraire. ⇒ le processus ne sait pas quand ni si tel ou tel signal va lui arriver

Noms et numéros des signaux

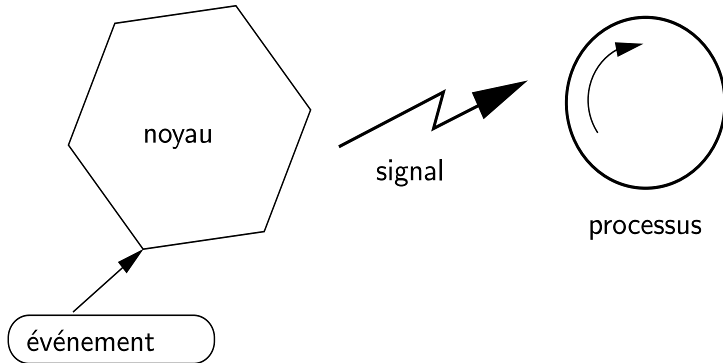
Signaux – 31 signaux de base en Unix/Linux

- Dans un programme → `<signal.h>`.
- Les noms descriptifs des signaux → un tableau *char * _sys_siglist[]*
- Il y a d'autres signaux que ceux-ci — « temps-réel », mais on ne les traite pas ici

- Les *noms* commencent avec SIG : `SIGALRM`, `SIGBUS`, `SIGINT`, `SIGFPE`, etc.
- Les numéros : de 1 à 31 → totalement déconseillé de les utiliser explicitement. Manque de portabilité.(man 7 signal)

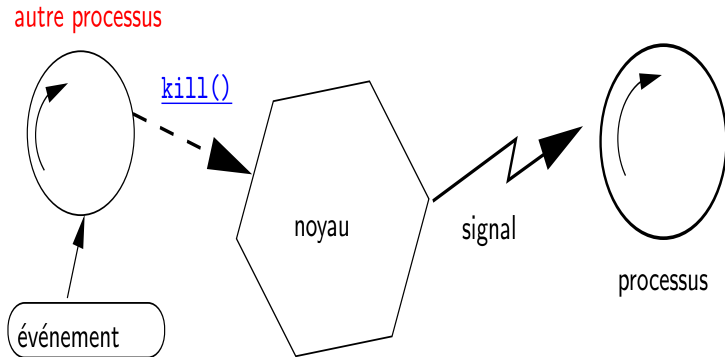
Sources des signaux

- le noyau prend connaissance de l'événement, génère et achemine le signal



Sources des signaux

- un autre processus, au moyen de l'appel système `kill()`, envoye le signal.



Génération des signaux

- Un signal peut être généré suite à
  - un problème matériel : division par zéro, problème d'addressage, défaillance d'alimentation électrique, etc.
  - l'appui de touches au clavier du terminal :
    - `Ctrl C` (envoi de `SIGINT`),
    - `Ctrl Z` (envoi de `SIGTSTP`), etc.
  - l'expiration de délai préprogrammé (fonction `alarm()`)
  - un événement logiciel des processus partenaires (lecteur absent sur pipe, processus fils terminé ou stoppé, etc.)
  - un besoin de stopper ou de terminer (plus ou moins brutalement) le processus
- C'est le noyau qui achemine le signal aux processus (identifiés par leur *pid*).

Envoi d'un signal : Fonction `kill()`

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t Pid, int NumSig);
```

Remarque

le nom de l'appel système `kill()` prête à confusion : pour la plupart des signaux, lors de leur réception, par défaut le noyau termine (« tue ») le processus — mais c'est reconfigurable.

Remarque

Il faut respecter leur signification.

`kill(12345,SIGKILL)`	Très bien : tue le processus 12345
`kill(12345,SIGFPE)`	Absurde : tue le processus 12345 sous le prétexte d'une erreur de calcul

Fin de processus

- la terminaison du processus (faite par le noyau) suite à la réception d'un signal est de deux types :
  - terminaison simple
  - terminaison avec dépôt de l'image mémoire du processus dans un fichier nommé *core*, à des fins de débogage.
    - `SIGFPE` — erreur de calcul (division par zéro, etc.)
    - `SIGSEGV` — addresse mémoire invalide (par ex. mauvais pointeur)
    - `SIGSYS` — appel système invalide (arguments invalides)
    - `SIGABRT` — fin anormale du programme — fonction `abort()`
- Par raison de sécurité,

  la taille du fichier est *core* à zéro.
  - pour changer ces limites, on a
    - les fonctions `setrlimit()` et `getrlimit()`.
    - les builtins du shell (par exemple, *ulimit* pour bash, *limit* pour csh, etc.) : *limit core 0*

Différents signaux, par défaut arrêtant le processus

- SIGTSTP — arrêt depuis un terminal — Ctrl Z
- SIGTTIN — tentative lecture depuis terminal alors que le processus est en arrière plan
- SIGTTOUT — tentative écriture sur terminal alors que le processus est en arrière-plan

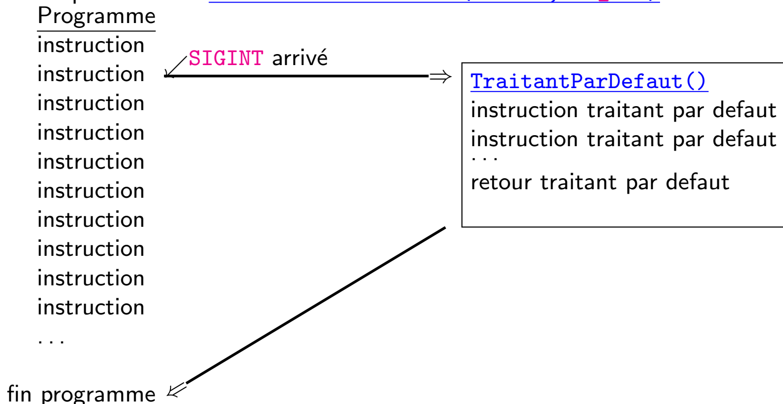
Remarque

- *Ces signaux, tout comme SIGSTOP, stoppent le processus, mais en revanche peuvent bien être déroutés au besoin.*
- *Ces signaux permettent l'implémentation du « job control ».*

```
allegro> a.out
Ctrl Z
[1]+ Stopped a.out
allegro> bg
[1]+ a.out &
allegro>
```

La réception des signaux : choix par défaut

laisser appliquer le traitement par défaut — pour la plupart des signaux, terminer le processus — DerouterVersTraitant(SIGINT,SIG_DFL)



Choix offert pour la réception des signaux : ignorer

On peut ignorer le signal, i.e. le faire « disparaître », le faire ne pas arriver effectivement au processus – `DerouterVersTraitant(SIGINT, SIG_IGN)`

Programme

instruction
`DerouterVersTraitant(SIGINT, SIG_IGN)`
instruction ↙ `SIGINT` arrivé
instruction
instruction
instruction
instruction
instruction
instruction
instruction
...

Différents signaux, par défaut ignorés

- `SIGCHLD` — changement d'état d'un processus fils —- sera discuté en détail dans la partie processus
- `SIGURG` — urgence (optionnellement lorsque des données hors-bande sont reçues depuis le réseau)
- `SIGWINCH` — changement taille fenêtre du terminal — le noyau maintient ces valeurs à jour, et en annonce les changements pour les applications qui affichent dedans — ainsi elles peuvent recalculer ce qu'elles affichent, nettoyer l'écran réorganiser l'information et la réafficher.

Choix pour la réception des signaux : traitant particulier

On peut dérouter le signal vers une fonction « à nous » `Derout()`

- execute une opération desirée par l'utilisateur
- profile imposé de type `sighandler_t` : un pointeur de fonction `int`→`void`

```
void Derout(int NumeroSignal);
```

```
typedef void (*sighandler_t)(int);
```

- si succès, la fonction rend le traitant précédent – pour le cas où l'on désire le sauvegarder pour le restaurer par la suite.
- si erreur : la valeur `SIG_ERR`
- constantes de type `sighandler_t` prédéfinies : `SIG_IGN` et `SIG_DFL`.

Remarque

Rappel syntaxe C :

- *pointeur de fonction – nom de la fonction*
- *appel de fonction – nom de la fonction avec ()*

Signal dérouté vers traitant particulier `Derout()`

Programme

instruction

`DerouterVersTraitant(SIGINT, Derout)`

instruction

instruction

instruction

instruction

instruction

instruction

instruction

instruction

...

`SIGINT` arrivé

`Derout()`

instruction traitant

instruction traitant

instruction traitant

instruction traitant

instruction traitant

instruction traitant

...

...

retour traitant

Exemple simple

```
void Derout(int NumeroSignal) {// notre traitant
   cout << "Recu␣signal␣" << NumeroSignal << endl;
}
....// et dans le programme, on l'installe
if(SIG_ERR == DerouterVersTraitant(SIGINT,Derout)) {
   cout<<"Probleme␣de␣deroutement␣de␣SIGINT..."<<endl;
}
else { .. // tout va bien, et SIGINT est maintenant
}          // deroute
```

- Si l'appel à `DerouterVersTraitant()` ok, par la suite, `Derout()` va être AUTOMATIQUEMENT appelé par le noyau à la réception d'un signal `SIGINT`.
- on peut attendre un signal, avec `sleep()` (temps fini) ou `pause()` (indéfiniment), les deux interrompues par l'arrivée d'un signal.

Signaux à traitant non modifiable

- trois signaux importants
  - `SIGKILL` — fin obligatoire (moyen sûr de terminer) *termine sans core*
  - `SIGSTOP` — arrêt « sur image » obligatoire (moyen sûr de stopper)
  - `SIGCONT` — si le processus était stoppé, alors il redémarre obligatoirement, sinon le signal est ignoré
- `SIGKILL` et `SIGSTOP` **ne peuvent pas être déroutés** ; si l'on tente leur déroutement, alors `DerouterVersTraitant()` revient en erreur ;
- pour `SIGCONT`, `DerouterVersTraitant()` ne revient pas en erreur, mais ne fait que rajouter l'éventuel traitant particulier.

Changements de dispositions en cours d'exécution

Programme

instruction

DerouterVersTraitant(SIGINT, SIG_IGN)

instruction

instruction

instruction

SIGINT arrivé

DerouterVersTraitant(SIGINT, SIG_DFL)

instruction

instruction

SIGINT arrivé

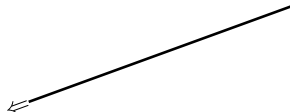
instruction

instruction

instruction

instruction

...



fin programme

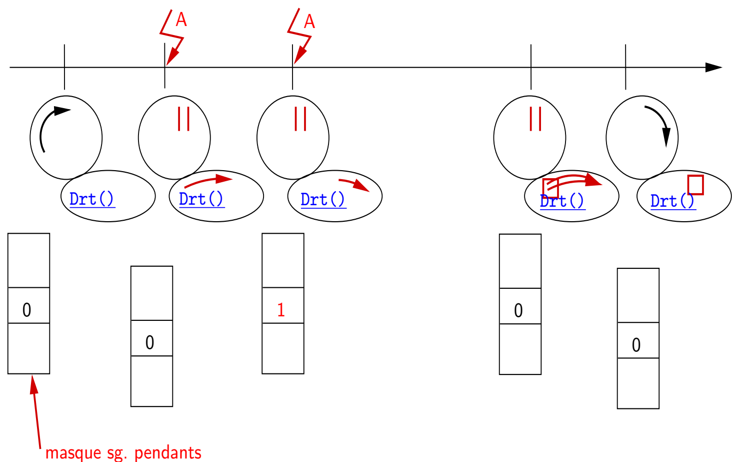
TraitantParDefaut()

instruction traitant par defa

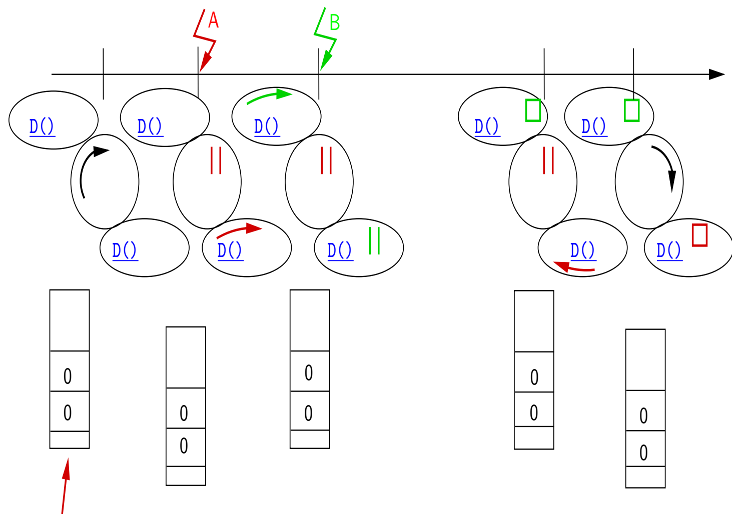
...

retour traitant par defaut

Arrivée deux fois du même signal : AA

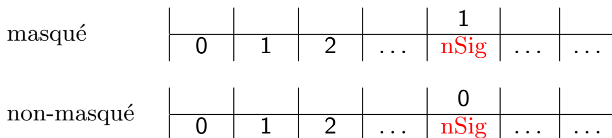


Arrivée alternée : AB



Comment manipule-t-on les signaux ?

- le type sigset_t : le masque des signaux — représentation ensembles signaux



- On dispose également des opérations
  - d'initialisation d'ensemble,
  - de rajout/enlèvement,
  - et de test d'appartenance

  fonctionnant sur ce type de données.

Manipulation ensembliste du masque des signaux

- les opérations sur sigset_t sont les suivantes

```
#include <signal.h>
int sigemptyset(sigset_t *EnsmblSig);// initialise
int sigfillset(sigset_t *EnsmblSig); // met tous les
                                       // signaux
int sigaddset(sigset_t *EnsmblSig, int NumSig);
                                      // ajoute
int sigdelset(sigset_t *EnsmblSig, int NumSig);
                                     // enleve
//ces quatre fonctions renvoient 0 si OK, -1 si erreur
int sigismember(const sigset_t *EnsmblSig,int NumSig);
//renvoie 1 si vrai, 0 si faux
```

DerouterVersTraitant : Fonction sigaction()

- contrôle fin de l'action à faire à la réception d'un signal

```
#include <signal.h>
int sigaction(int NumSig,
              const struct sigaction *Action,
              struct sigaction *AncienneAction);
```

- deux comportements distincts :
  - changement : si *AncienneAction* n'est pas nul, alors il est rempli avec l'action précédente
  - sauvegarde et test : si *Action* est nul, alors *AncienneAction* est rempli avec l'action courante pour le signal *NumSig*, et il n'y a pas de changement d'action pour ce signal.
- la structure sigaction indique
  - le traitant à appeler (une fonction, ou ignorer, ou ce qui se passe par défaut),
  - les autres signaux à bloquer pendant l'exécution de traitant
  - des options supplémentaires

DerouterVersTraitant : Structure sigaction

```
struct sigaction {
 void (*sa_handler)(int);//traitant, SIG_DFL ou SIG_IGN
 int sa_flags; //options supplementaires
 sigset_t sa_mask;         //autres signaux a bloquer
                           //pendant traitmnt
 ....                      // autre champs

};
```

Remarque

Le membre sa_mask est à manipuler avec les fonctions agissant sur les ensembles de signaux `sigset_t` *qu'on vient de voir :*

- sigemptyset()
- sigsetset()
- sigprocmask()
- sigismember()

DerouterVersTraitant : Notes sur `sigaction()`

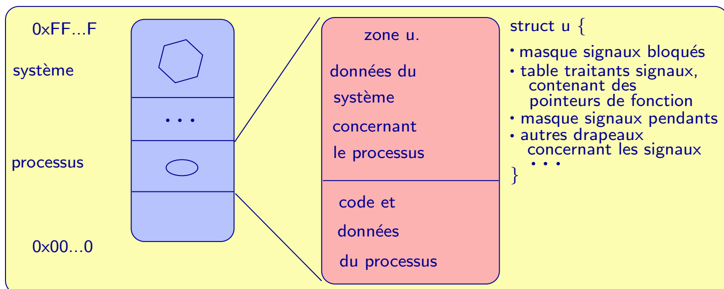
Parmi les options supplémentaires pour le membre sa_flags il y a

- `SA_NODEFER` – ne pas bloquer le signal pendant l'exécution de son propre traitant
- `SA_RESETHAND` – restauration à `SIG_DFL` dès l'entrée dans le traitant.
- `SA_RESTART` – redémarrage automatique de certains des appels système interrompus par le signal délivré

Remarque

Il existe également une fonction nommée `signal()`*, dont nous n'allons pas nous servir dans ce cours. Pour* `DerouterVersTraitant` *nous utilisons la fonction* `sigaction()` *qui nous permet de tout contrôler.*

Où trouve-t-on les renseignements pour le noyau ?



Notes sur l'interruption

Le signal est reçu par le processus (i.e. n'est pas ignoré ou bloqué).
Si le processus

- exécute du code utilisateur : interruption du code — exécution traitant signal
- attend pour qu'un appel système finisse : alors
  - certains appels système sont interrompus également, avec *errno*=`EINTR`, et
  - d'autres finissent – e.g. `read()` depuis un vrai fichier disque.
- Le *man* l'indique pour chaque fonction.

Notes sur le traitant de signal

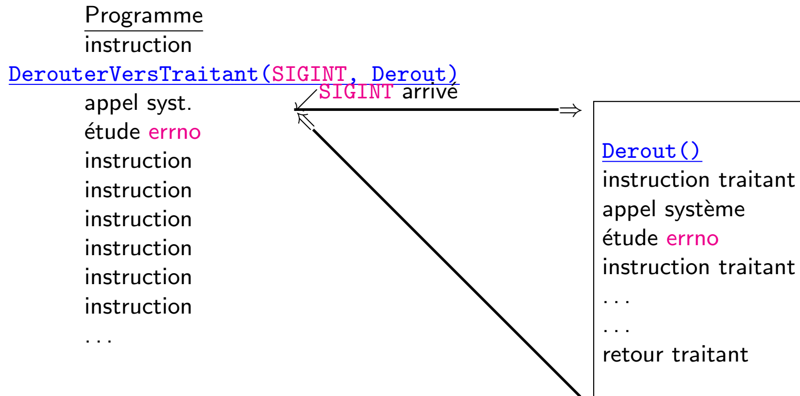
- il ne doit pas lever d'exceptions
- il doit être écrit avec soin → problèmes typiques
  - l'appel de fonctions unsafe
  - la sauvegarde des variables globales
- Les standards (comme POSIX) donnent les listes explicites des fonctions système safe, auxquelles il faut se tenir.

Fonctions safe et unsafe

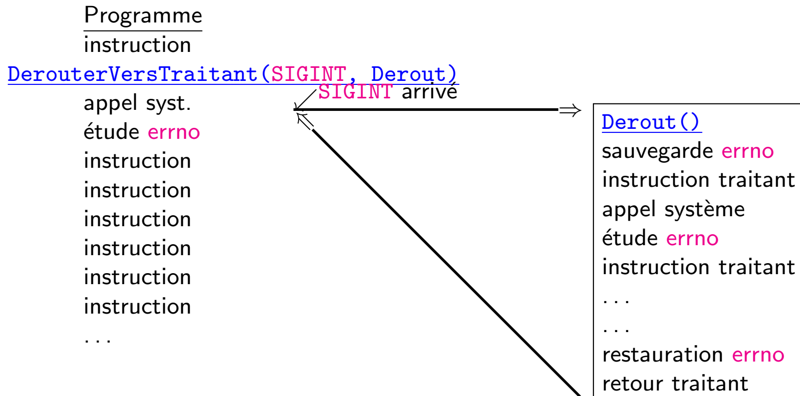
- variable globales ou static — un seul emplacement pour stockage en mémoire, pour tout le programme — commun à tous les appels de fonctions
- fonctions unsafe
  - utilisent des variables globales ou static, pour
    - rendre le résultat (comme `getlogin()`)
    - stocker des renseignments nécessaires d'un appel à l'autre (comme `rand()`)
  - si
    - appelées, puis
    - interrompues (arrivée d'un signal) et
    - appelées depuis le traitant du signal

  ⇒ incohérence des valeurs dans la zone globale ou static

Mauvais traitement de errno dans traitant particulier



Bon traitement de errno dans traitant particulier



Exemple de cas unsafe – strtok()

- strtok() sert à tokeniser (décomposer en lexèmes) une chaîne de caractères
- appelée une première fois ainsi avec une char *maChaine contenant du texte :

```
strtok(maChaine,",");
```

  - elle initialise un pointeur interne (variable locale, static)
  - qu'elle réutilise d'un appel à l'autre – static
  - pour parcourir maChaine de virgule en virgule
  - rendant les morceaux entre
- par exemple
  - si maChaine="pomme,poire,prune,cerise"
  - alors à chaque appel sucessif, strtok() rend un char * pointant au début de chaque nom de fruit (et remplace les virgules par des '\0')
  - (il faut l'appeler avec '0' pour les autres fois)

Exemple de cas unsafe – suite

- si le programme appelle `strtok()` lorsqu'un signal arrive
- et que le traitant appelle `strtok()` **AUSSI**, avec une chaîne
  `strtok(chaineDuTraitant,":");`
- alors `strtok()` **RÉINITIALISE** son pointeur interne
- et **OUBLIE** ce qu'elle était en train de faire pour le programme interrompu
- au retour du traitant, le programme, lors d'un rappel de `strtok()`, reçoit du n'importe quoi.
- même si le traitant appelle `strtok()` avec zéro il y a un problème :
  - il fait avancer le pointeur interne, sans que le programme ait un moyen de le savoir
- Comment bien faire les choses ?

POSIX safe

- appelé aussi réentrant
- pour les fonctions unsafe — définition des contreparties safe : rajout du _r au nom de la fonction
  - `strtok_r()`
  - `getlogin_r()`
  - `rand_r()`, etc.
- N'utilisent plus de variables globales ou static et rendent le résultat dans une zone mémoire fournie par le code appelant
- Exemple,
  - `strtok_r(char *s, const char *delim, char **ptrptr);` utilise `ptrptr` au lieu du pointeur interne
  - le programme **ET** le traitant peuvent appeler TOUS LES DEUX `strtok_r()`,
  - à l'appel, chacun donne un `ptrptr` déclaré localement
- si elles sont obligées d'utiliser des variables globales ou static, alors synchronisent l'accès (attente mutuelle, une seule instance y accède)

Signaux délivrés — étapes usuelles

1. production de l'événement
2. envoi du signal (appel de `kill()`, etc.)
3. acheminement du signal par le noyau (recherche du processus, examination dans la zone u. de sa table de traitants, verification qu'il n'est pas ignoré,),
4. on rajoute le signal dans le masque des signaux pendants du processus
5. examination si le signal est à délivrer ou à bloquer (examination du masque des signaux bloqués)

A. si le signal n'est pas bloqué, examination de l'action à faire lors de la réception

6. exécution de cette action effective — signal dit délivré

[Entre les étapes : des temps arbitraires — aspect asynchone.

- le signal est dit pendant de l'étape 1. à l'étape 5.

Signaux bloqués, pendants — étapes usuelles

1. production de l'événement
2. envoi du signal (appel de `kill()`, etc.)
3. acheminement du signal par le noyau (recherche du processus, examination de sa table de traitants, verification qu'il n'est ignoré)
4. rajout dans masque des signaux pendants du processus
5. examination si le signal est à délivrer ou à bloquer

B. constat par le noyau que le signal est bloqué pour ce processus

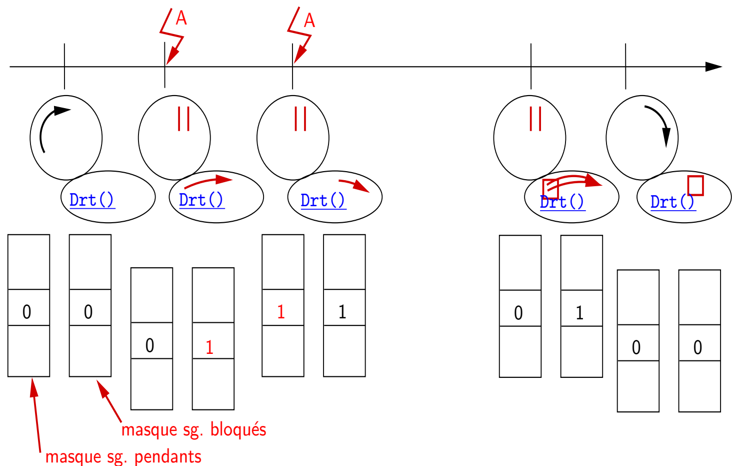
et le signal reste pendant.

- le processus peut changer les dispositions plus tard dans le programme

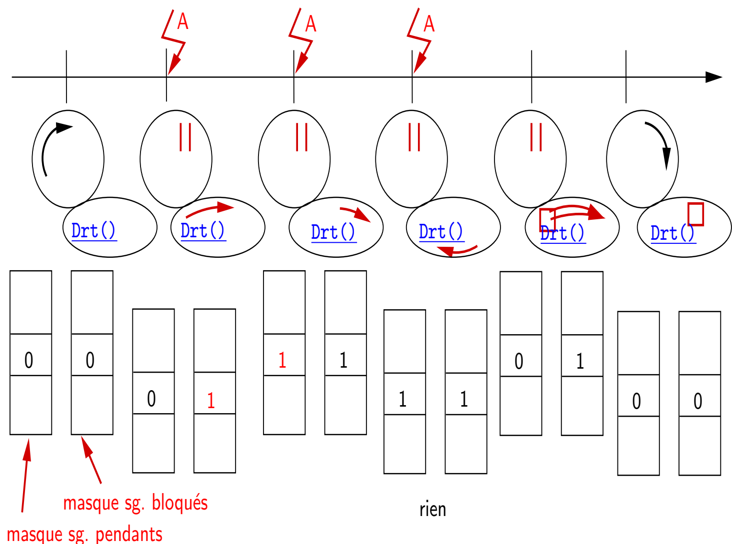
Remarques – signaux bloqués

- retrouver les signaux bloqués : la fonction `sigpending()`
- l'arrivée d'un signal peut être bloquée
  - par défaut par le noyau
  - par requête explicite de l'utilisateur
- Ignorer≠Bloquer
  - ignorer ⇒ perdre définitivement toute trace du signal
  - bloquer
    - noter son arrivée – étape 5 (marquage)
    - prendre en compte lors de l'éventuel déblocage – étape A.

Bloquer par défaut : arrivée deux fois du même signal : AA



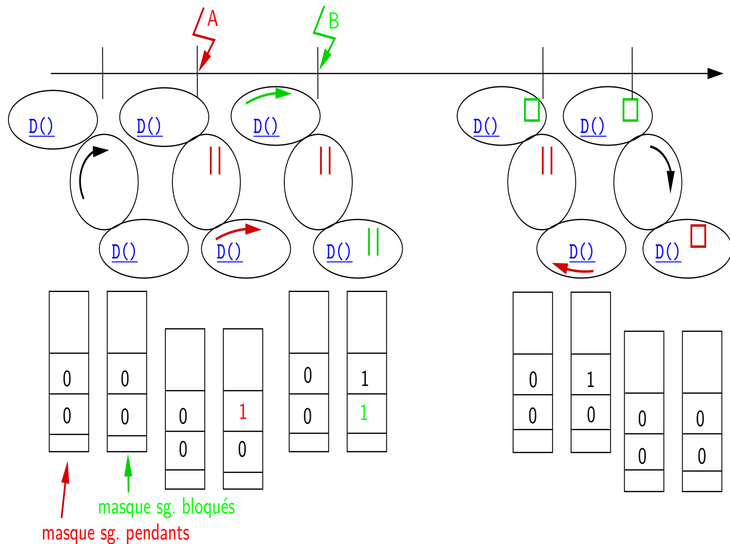
Le signal A arrive pendant l'exécution du traitant de A ! AAA= AA



Le signal A arrive pendant l'exécution du traitant de A ! AAA = AA

- le signal A est temporairement automatiquement bloqué par le noyau
- s'il arrive une fois pendant l'exécution (de son traitant `Derout()`)
  - il est enregistré dans le masque des signaux pendants
- s'il arrive encore une fois pendant l'exécution (de son traitant) qui n'a pas encore fini
  - il n'est empilé nulle part, donc AAA==AA
  - il est délivré dès la fin du traitant `Derout()`
  - provoquant ainsi la réexécution immédiate de `Derout()`
- ceci peut bien sûr se répéter si le signal arrive à nouveau pendant la réexécution du traitant du signal

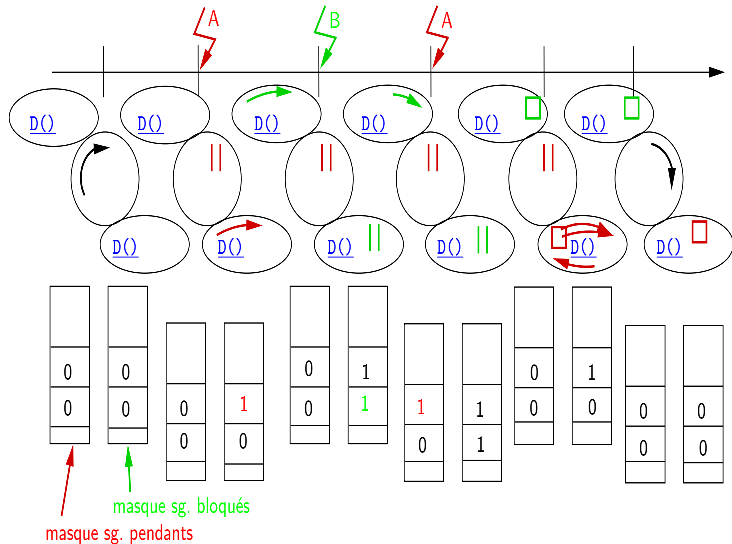
Bloquer par défaut : arrivée alternée : AB



Exemple arrivée alternée : AB ?

- un autre signal B, sauf blocage demandé explicitement,
  - est délivré et
  - interromp le traitant `Derout()` en cours
  - est traité à son tour (immédiatement, donc) selon ses dispositions prises auparavant
    - `SIG_DFL`, ou bien
    - `SIG_IGN`, ou bien
    - traitant particulier
  - à la fin du traitement du signal B, le traitant du signal A reprend (si l'action de B n'était pas de terminer le processus).

Exemple arrivée alternée répétée : ABA



Et si A arrive à nouveau, i.e. ABA ?

- pendant le traitement d'un signal A suite au déroutement vers un traitant particulier `Derout()`,
- un autre signal B, sauf blocage demandé explicitement,
  - est délivré et
  - interromp le traitant `Derout()` en cours
  - est traité à son tour (immédiatement, donc) selon ses dispositions prises auparavant
  - si le signal A arrive à nouveau pendant le traitement de B
  - il est noté comme pendant (car bloqué depuis le tout début — son traitant)
  - à la fin du traitement du signal B, le traitant du signal A reprend (si l'action de B n'était pas de terminer le processus).
  - et lorsque `Derout()` pour A finit également
  - `Derout()` pour A est réexécuté immédiatement, de nouveau

Et si l'on a ABAB ?

- Le traitant de B — également réexécuté en sortie. Ordre arbitraire de réexécution des traitants — signaux pendants sans priorités.

Bloquer pendant l'exécution d'un processus ?

- zone u. $\rightarrow$ explicitement
  - une table, appelée masque des signaux bloqués.
  - consultée à l'étape 5
  - un processus peut y accéder et la modifier avec la fonction `sigprocmask()`.
- implicitement – par défaut pendant l'exécution du traitant d'un signal, le noyau bloque le signal A

Remarque

Les trois signaux importants `SIGKILL`, `SIGSTOP` *et* `SIGCONT` *ne peuvent pas être bloqués.*

Bloquer pendant l'exécution d'un processus : sigprocmask()

```
#include <signal.h>
int sigprocmask(int quoiFaire, const sigset_t *EnsSig,
                sigset_t *EnsSigAncien)
// renvoie 0 si OK, -1 si erreur
```

- *quoiFaire* peut être
  - SIG_BLOCK – le nouveau masque est l'union entre l'ancien et *EnsSig*
  - SIG_UNBLOCK – le nouveau masque est l'intersection entre l'ancien et *EnsSig*
  - SIG_SETMASK – le nouveau masque est *EnsSig*.
- deux comportements distincts :
  - changement de masque
    - si *EnsSigAncien* n'est pas nul, alors il est rempli avec le masque précédent
  - sauvegarde et test sur le masque existant
    - si *EnsSig* est nul, alors *EnsSigAncien* est rempli avec le masque courant, et il n'y a pas de changement (et *quoiFaire* n'a pas d'importance).

Exemple

```
#include <signal.h>
#include <iostream>
int main() {
 sigset_t m,p; sigemptyset(&m); // initialisation
 Sigaddset(&m,SIGQUIT);Sigaddset(&m,SIGTSTP); // rajout
 Sigprocmask(SIG_SETMASK,&m,0); // bloquer
 Sigprocmask(0,0,&p); // interrog 'quel est le msq?'
 for (int k = 1; k < 32; ++k)  // balayage des signx
  if(Sigismember(&p, k)) // analyse du masque p
    std::cout<<"Blk: "<<k<<" "<<_sys_siglist[k]<<"\n";
 return 0;// a la fin de la boucle
}
/* affiche en sortie standard
 Blk: 3 Quit
 Blk: 20 Stopped
*/
```

Fonction sigpending()

- sigpending() donne la liste des signaux qui sont arrivés et pas encore délivrés — remplissant la zone mémoire pointée par son argument.

```
#include <signal.h>
int sigpending(sigset_t *SignauxPendants);
```

Pour des tests :

```
#include <signal.h>
int raise(int NumSignal);
```

envoie un signal au processus lui-même.

Exemple utilisation sigpending()

```
#include <signal.h>
#include <iostream>
int main() {
 sigset_t m,p; Sigemptyset(&m); // initialisation
 Sigaddset(&m,SIGQUIT);Sigaddset(&m,SIGTSTP); // rajout
 Sigprocmask(SIG_SETMASK,&m,0);
 raise(SIGQUIT);    // envoi SIGQUIT a soi-meme
 Sigpending(&p);    //'quels sgn en attnte?'
 for (int k = 1; k < 32; ++k)    //balayage des sgnx
  if(Sigismember(&p, k)) // analyse de l'ensemble p
    std::cout<< "Pnd: "<<k<<" "<<_sys_siglist[k]<<"\n";
 return 0;
}
/* affiche en sortie standard
 Pnd: 3 Quit
*/
```

Bloquer pendant l'exécution d'un traitant ?

- il y a aussi un masque des signaux bloqués pendant l'exécution du traitant
- on peut le changer lors de l'appel de sigaction()

```
struct sigaction {
 void (*sa_handler)(int);//traitant, SIG_DFL ou SIG_IGN
 int sa_flags; //options supplementaires
 sigset_t sa_mask;       //autres signaux a bloquer
                         //pendant traitmnt
 ...                     // autres champs opt.
};
```

- le champ sa_mask est à manipuler avec :sigemptyset(), sigsetset(), sigismember()
- pseudo-wrapper simplifie pour sigaction()

Blocage pendant un traitement sensible : sigprocmask()

- un signal arrive ⇒ exécution immediate de son traitant ⇒ interruption du programme
- protéger les zones de traitement sensible contre l'arrivée des signaux
- les signaux peuvent être deroutés vers un traitant particulier

```
main() {
 .... //   traitant de NumSig par d\'{e}faut
 Derouter(NumSig, DeroutSig);
 .... //   le traitant est DeroutSig

 Bloquer NumSig;
 .... //   traitement sensible=>protege

 Debloquer NumSig;
 ....//    si NumSig \'{e}tait arrive
 ....//    DeroutSig() est appele de suite
 ....//    sinon non
}
```

Blocage pendant un traitement sensible : sigprocmask()

```
void Derout (int n) { corps du traitant } // noter

main() {
    DerouterVersTraitant(Sig, Derout);
    sigset_t Masque;
    ... //prep Masque (initialisation, ajout Sig)
    ... //Sigemptyset(&Masque);Sigaddset(&Masque,Sig);

    Sigprocmask (SIG_SETMASK, &Masque, 0);

    .... //   traitement sensible=>protege
    ... //prep Masque (initialisation, enlevement Sig)
    ... //Sigemptyset(&Masque);Sigdelset(&Masque, Sig);
    Sigprocmask (SIG_SETMASK, &Masque, 0);
}
```

Comment noter l'arrivée d'un signal pour traiter plus tard

Remarque

Il faut une variable globale puisque la fonction traitant le signal n'a qu'un seul paramètre et ne rend rien.

- Les types usuels de données, l'accès — non-atomique, i.e.
  - les instructions (machine) du programme principal— en train de lire une variable globale déposée en mémoire,
  - les instructions machine du traitant — en train d'écrire une autre valeur dans la même variable
- $\Rightarrow$ conflit – incohérence des valeurs
- le type `sig_atomic_t` : l'accès à une variable globale de manière atomique du point de vue des interruptions par arrivée de signal

Comment noter l'arrivée d'un signal pour traiter plus tard

```
namespace {
 volatile sig_atomic_t NumSigArrive = 0;
}
 void Derout(int n) {  NumSigArrive = 1; } // noter
```

- le point de vue du compilateur qui ne comprend pas les signaux :
  - dans le `main()` *NumSigArrive* n'est pas modifée, seulement lue,
  - il n'y a pas d'appel explicite de `DeroutSig()`
- il optimise : *const NumSigArrive*
- pour l'empêcher et laisser `DeroutSig` s'exécuter si un signal arrive)— *volatile* dans la déclaration.

On note mais on attend pas

```
namespace { volatile sig_atomic_t NumSigArrive=0;}
void DeroutSig (int n) {  NumSigArrive = 1; } // noter
main() {
 Derouter(NumSig, DeroutSig);
 ...
 Bloquer NumSig;
 // traitement sensible=>protege
 .... //pas de NumSig delivre, pas d'appel de DeroutSig
 Debloquer NumSig;
 ...// traitement nonsensible
 if(NumSigArrive) {
 ...
 }
 ......
}
```

Attente de signal — naïve et fausse

```
namespace { volatile sig_atomic_t NumSigArrive=0;}
void DeroutSig (int n) {  NumSigArrive = 1; } // noter
main() {
 Derouter(NumSig, DeroutSig);
 ...
 Bloquer NumSig;
 ...    // traitement sensible=>protege

 Debloquer NumSig;
 ::pause() // attente signal
 if(NumSigArrive) {
 ...
 }
 ......
}
```

- Problème : Signal arrivé entre déblocage et pause() ...perdu
- Solution noyau : déblocage, attente (et reblocage) atomiquement — fonction système sigsuspend()

Fonction sigsuspend()

```
#include <signal.h>
int sigsuspend(sigset_t *MasqueTemporaire);
```

- remplace le masque du processus avec le *MasqueTemporaire*
- suspend le processus jusqu'à l'arrivee du signal
- remet le le masque du processus en place
- sort (en erreur — `EINTR`)

atomiquement — pas interruptible entre ses étapes.

Remarque

Ainsi on ne perd pas de signal qui se serait glissé entre un `Sigprocmask()` *et un* `pause()`.

Attente avec la fonction sigsuspend()

```
namespace { volatile sig_atomic_t NumSigArrive = 0; }
void DeroutSig (int n) {  NumSigArrive = 1; } // noter
main() {
sigset_t Masque, MasqueVide;
... // prep Masque (ajout NumSig), vidange MasqueVide
Sigprocmask (SIG_SETMASK, &Masque, 0);
DerouterVersTraitant(NumSig, DeroutSig);
 .......
Sigsuspend(&MasqueVide);// attente atomique
if(NumSigArrive == 0) {
    ....
}
......
}
```

A quoi sert d'attendre les signaux : Schéma MiniGestionaire

- *Sig*1 signale la fin, *Sig*2 signale une action spécifique
- tant que pas le signal *Sig*1 n'est pas arrivé
  - traitement répétitif sensible et protegé : mise à jour des processus déjà lancés
  - attente signal quelconque
  - traitement spécifique si *Sig*2 arrive : saisie de commandes, lancement des processus qui les exécutent, etc.
- ainsi : sortir si *Sig*1 arrive

Mise en œuvre

```
namespace { volatile sig_atomic_t Sig1Arrive = 0;
            volatile sig_atomic_t Sig2Arrive = 0; }
void DeroutSig1 (int n) { Sig1Arrive = 1; } // noter
void DeroutSig2 (int n) { Sig2Arrive = 1; } // noter
main() { sigset_t Masque, MasqueVide;
... //prep Masque (ajout Sig1,Sig2), vidange MasqueVide
Sigprocmask (SIG_SETMASK, &Masque, 0);
DerouterVersTraitant(Sig1, DeroutSig1);
DerouterVersTraitant(Sig2, DeroutSig2);
 .......
 while(!Fin) {
   ...
   ...//  traitement sensible
   Sigsuspend(&MasqueVide);// attente Sig1 ou Sig2
   if(Sig1Arrive) { Fin = vrai; }
   if(Sig2Arrive) { ...traitment sp\'{e}cifique␣}␣
   ....
 }
}
```

Stratégies gestion signaux

- traiter dès qu'il arrive
  - traitement par défaut
  - ignorer le signal
  - déroutement vers un traitant particulier : attention aux var. globales
- blocage pendant l'exécution d'un traitant :sigaction()
- blocage pendant traitement sensible dans le programme principal :sigprocmask()
  - le prog. bloque le signal, exécute le traitement sensible
  - le prog. débloque le signal et exécute le traitant
- attente d'un signal dans le programme principal :
  - le traitant ne fait que modifier une variable globale
  - le prog. deroute le signal A vers le traitant
  - le prog. bloque le signal, exécute le traitement sensible : sigprocmask()
  - le prog. attend le signal : sigsuspend()
  - le prog. teste la variable globale et si elle est modifiée, il exécute le traitement approprié

C'est l'heure

- la fonction système alarm()

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

  - met en marche une minuterie (*combienDeSecondes* > 0)
  - renvoie combien de secondes restaient avant l'arrêt de la minuterie
  - le signal SIGALRM est le signal envoyé automatiquement par le noyau lorsque la minuterie expire.
  - $\Rightarrow$ dérouter le signal SIGALRM sinon le processus fini lors de l'expiration du délai.
- il y a une seule minuterie par processus.

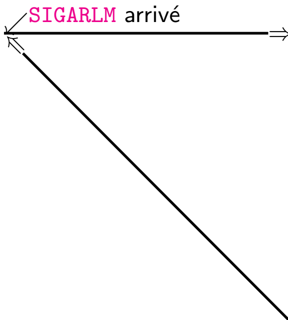
Minuterie avec alarm() et SIGALRM

Programme

DerouterVersTraitant(SIGALRM, Minuterie)

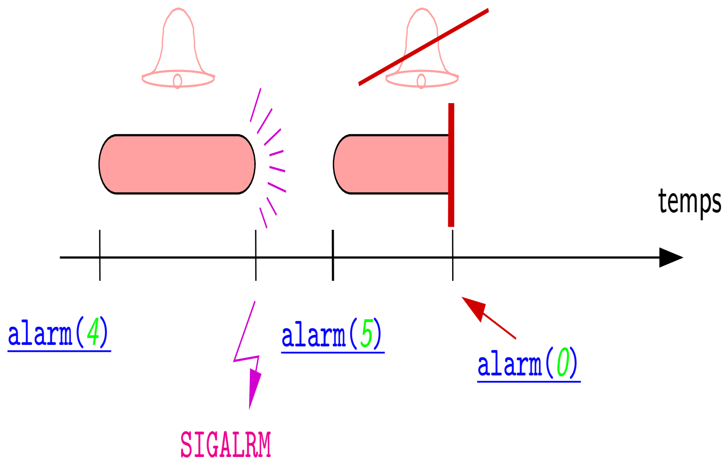
alarm(*intervalle*)

instruction
...
instruction
instruction
instruction
instruction
instruction
instruction
instruction
instruction
...
...

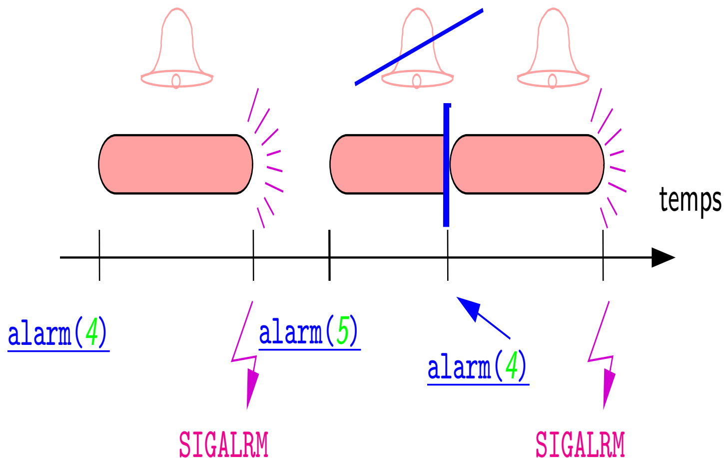


Minuterie()
instruction minuterie expirée
instruction minuterie expirée
instruction minuterie expirée
instruction minuterie expirée
instruction minuterie expirée
instruction minuterie expirée
...
...
retour traitant

Appels successifs à alarm() – annulation

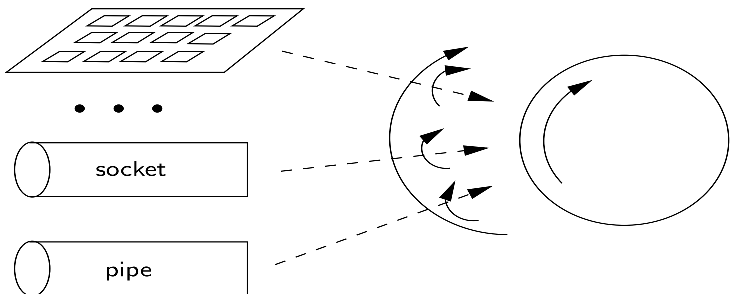


Appels successifs à `alarm()` – reconfiguration



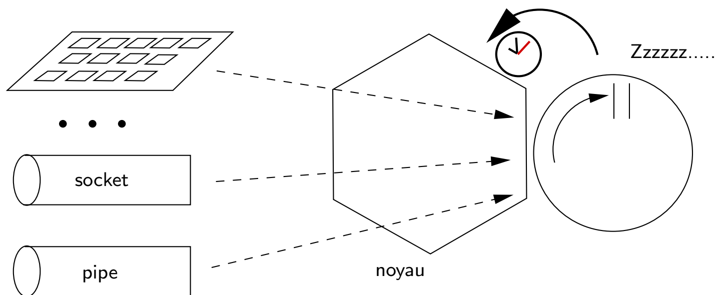
Multiplexage entrées-sorties

- un processus — plusieurs descripteurs de fichier :
  - le clavier, la souris
  - des pipes ou sockets, etc.
- besoin de les regarder périodiquement, sans bloquer
- **Solution lourde** : `read()` non-bloquant de chaque descripteur, dans une boucle — attente active, gaspillage CPU



Multiplexage entrées-sorties

- solution préférable : `select()`
  - laisse le système vérifier et « attendre »
  - le processus est endormi le temps
    - qu'au moins un descripteur devienne **prêt**, ou bien
    - maximum spécifié à `select()` comme délai de patience

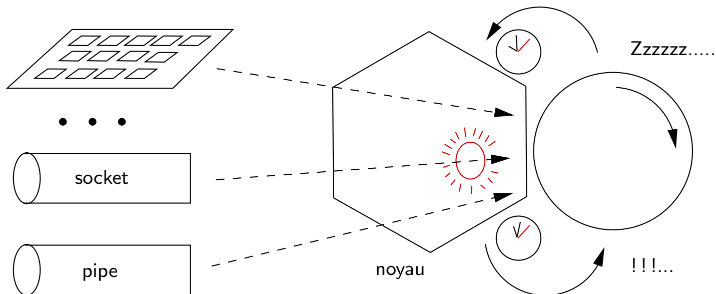


Multiplexage entrées-sorties – Descripteur prêt

- pour la lecture – un read() ne bloquerait pas
- pour l'écriture – un write() ne bloquerait pas

Remarque

Attention, select() *considère un descripteur comme étant prêt même si* EOF *est rencontré sur le descripteur (i.e. un* read() *ultérieur revient avec la valeur de retour zéro) — pas d'exception ou blocage.*

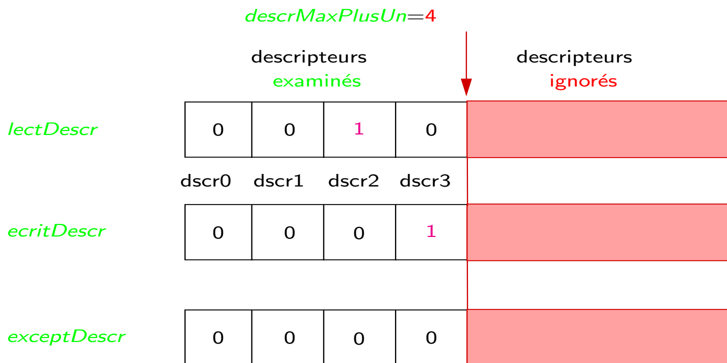


Fonctionnement appel `select()`

- pendant un délai le noyau guette trois ensembles de descripteurs de fichier pour des événements en :
  - lecture
  - écriture
  - exceptions
- le processus appellant s'endort, et quand
  - au moins un descripteur devient prêt
  - ou bien le délai expire
- le noyau reveille le processus et revient de `select()`, ayant marqué dans les ensembles les descripteurs prêts.

descrMaxPlusUn et les `fd_set`s

- *descrMaxPlusUn* donne le nombre de descripteurs à examiner



Fonction select()

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int descrMaxPlusUn, fd_set *lectDescr,
           fd_set *ecritDescr, fd_set *exceptDescr,
           struct timeval *delai);
```

- la structure timeval spécifie le délai d'attente ainsi

```
struct timeval {
  long    tv_sec;          /* secondes */
  long    tv_usec;         /* microsecondes */
};
```

- le type fd_set — manipulation des ensembles de descripteurs de fichier

Valeur de retour de select()

- si erreur (ou interruption par signal, etc.), alors -1 (et errno positionnée).
- si timeout expiré et aucun descripteur prêt alors 0
- si au moins un descripteur prêt alors
  - valeur positive (en principe la somme des bits à un)
  - les seuls bits encore à 1 sont ceux des descripteurs prêts

		dscr0	dscr1	dscr2	dscr3	
lectDescr	entrée	1	0	1	1	
lectDescr	sortie	0	0	1	0	

Remarque

- *La fonction* `select()` *modifie donc ses paramètres d'entrée* `fd_set*`.
- **certains** *noyaux modifient le paramètre* `timeval *` *en sortie — intervalle de temps restant*

Manipulation du `fd_set`

Un bit par descripteur :

- allocation, copie (comme toute variable)
- (re)mise à zéro — macro
  `FD_ZERO(fd_set *`*ensbleDescr*`)`
- mise du bit d'un descripteur à `un` — macro
  `FD_SET(int` *descr*`, fd_set *`*ensbleDescr*`)`
- mise du bit d'un descripteur à zéro — macro
  `FD_CLR(int` *descr*`, fd_set *`*ensbleDescr*`)`
- test du bit d'un descripteur s'il est `un` — macro
  `FD_ISSET(int` *descr*`, fd_set *`*ensbleDescr*`)`
  pouvant être utilisée dans un `if`.

Remarque

Attention à tester les descripteurs en retour de `select()` pour savoir s'ils sont prêts, uniquement si la valeur de retour de `select()` est **positive**.

Appels de select()

- si intérêt seulement certaines opérations, par exemple seulement lecture
- les autres `fd_set *` peuvent être nuls.

```
select(descrMaxPlusUn, &lectDescr, 0, 0, &delai);
```

- si tous les trois `fd_set *` sont nuls ?
- alors `select()` se comporte comme `sleep()` — précision plus grande.

```
select(0, 0, 0, 0, &delai);
```