Interface de communication asynchrone : signaux UNIX

Plan

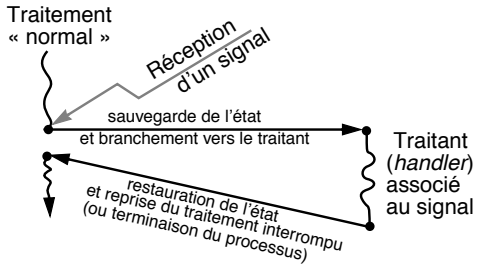
1 – La communication entre processus

UNIX fournit un ensemble de services permettant à un processus de communiquer avec son environnement ou avec d'autres processus, selon diverses formes et modalités :

- Communication asynchrone d'événements (signaux) :
  - ◊ **Schéma publier/s'abonner**
    - le récepteur manifeste son intérêt pour l'occurrence d'événements à venir/de données à produire en *s'abonnant*
    - chaque nouvelle occurrence est *transmise par l'émetteur* au récepteur
    - en réaction à cette transmission, le récepteur interrompt (provisoirement) son comportement courant pour traiter l'événement
  - ◊ Mécanisme analogue aux interruptions matérielles
  - ◊ Utilisation : communication d'événements asynchrones par le système (erreurs, interactions avec le matériel) ou l'utilisateur
- Communication synchrone
  - ◊ *le récepteur décide de l'instant de la réception*
  - ◊ communication explicite
    - flots d'E/S : fichiers, tubes
    - files de messages (IPC System V et POSIX)
    - sockets: canaux virtuels entre processus quelconques, éventuellement distants (vu plus tard)
  - ◊ communication implicite : mémoire (virtuelle) partagée et sémaphores (vus plus tard)

2 – Signaux : utilisation et réalisation

- un signal traduit l'occurrence d'un événement « observé » par l'environnement du processus qui reçoit le signal :
  - ◊ erreur liée à l'exécution du processus récepteur (accès errroné…)
  - ◊ certains événements matériels (frappe de caratères particuliers…) transmis par le système
  - ◊ événements applicatifs transmis par d'autres processus utilisateurs
- fonctionnement analogue au traitement des interruptions matérielles
  - ◊ différence : un signal ne correspond pas forcément à un événement matériel
  - ◊ transfert de contrôle ≈ appel de procédure *non contrôlé par le processus récepteur*



Quelques signaux (<signal.h>)

mnémonique	*événement correspondant*	*traitant par défaut*
SIGHUP	terminaison du leader	terminaison
SIGINT	control-C au clavier	terminaison
SIGQUIT	control-\ au clavier	terminaison+core
SIGTSTP	control-Z au clavier	suspension
SIGCONT	continuation d'un processus stoppé	reprise
SIGKILL	terminaison	terminaison
SIGPIPE	écriture dans un tube sans lecteur	terminaison
SIGFPE	erreur arithmétique (overflow…)	terminaison
SIGCHLD	terminaison d'un fils	vide (SIG_IGN)
SIGALRM	interruption horloge	terminaison
SIGTERM	terminaison normale	terminaison
SIGUSR1	laissé à l'utilisateur	terminaison
SIGUSR2	laissé à l'utilisateur	terminaison

- un traitant par défaut est associé à chaque signal
- le traitant par défaut peut être redéfini par l'utilisateur, sauf pour SIGKILL et SIGSTOP
- les mnémoniques sont communs à tous les UNIX
  (mais pas les numéros : SGCHLD vaut 17 pour Linux, 18 pour Solaris, 20 pour FreeBSD)
- un traitant vide permet d'ignorer un signal

Mise en œuvre

Dans chaque descripteur de processus :

				1 2 NSIG
1	0/1	0/1	void (*)(int)	1 0/1 ... 0/1
2	0/1	0/1	void (*)(int)	0/1 1 ... 0/1
...				
NSIG	0/1	0/1	void (*)(int)	0/1 0/1 ... 1

numéro de signal — masqué — en instance (pendant) — adresse traitant — masque durant l'exécution du traitant

- un signal reçu, mais non pris en compte est *en instance (ou : pendant)*
- lorsqu'un signal *masqué* est reçu, son traitement est mis en attente, jusqu'à ce que ce signal soit démasqué
- toute nouvelle occurrence d'un signal pendant est perdue
- lorsque le traitant associé au signal *S* est exécuté, *S* est masqué

Exemple

```
#include <signal.h>
void message(int sig) { /* traitant */
 printf("signal %d reçu\n",sig);
 exit(0);
}
int main() {
 signal(SIGINT, message); /* installe le traitant */
 signal(SIGQUIT, SIG_IGN); /* ignorer SIGQUIT*/
/* SIGINT et SIGQUIT sont interceptés */
 ...
/* on rétablit la terminaison par SIGINT et SIGQUIT */
 signal(SIGINT, SIG_DFL);
 signal(SIGQUIT, SIG_DFL);
 ...
}
```

Commentaires

- L'entier paramètre du traitant est le numéro du signal ayant provoqué l'exécution du traitant
  -> possibilité d'identifier l'évènement déclencheur dans le traitant
- 2 traitants sont définis par défaut
  - ◊ SIG_DFL : traitant par défaut associé au signal
  - ◊ SIG_IGN : traitant vide, permettant d'ignorer un signal
- En POSIX (et BSD), l'association définie par signal/sigaction est permanente

3 – Primitives de manipulation des signaux

Un des services où l'on observe le plus de divergences entre UNIX :

interfaces ≠, mécanismes ≠ } pour { BSD, POSIX, System V

Présentation centrée sur la définition POSIX

1) Association traitant/signal

Définition POSIX

```
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

avec

```
struct sigaction {
 void (*sa_handler)(int);/* pointeur sur traitant */
 sigset_t sa_mask;       /* signaux à masquer durant l'exécution du traitant */
 int sa_flags;           /* options (souvent spécifiques aux implantations) */
}
```

Définition C standard

```
#include <signal.h>
typedef void (*handler_t)(int); /* procédure prenant un paramètre entier */
handler_t signal (int sig, handler_t traitant)
/* signal renvoie l'adresse du traitant précédent */
```


Héritage du traitement des signaux

- Après `fork()` : oui
- Après `exec()` :
  - ◊ le masque est conservé
  - ◊ les signaux ignorés (associés à `SIG_IGN`) le restent
  - ◊ les autres signaux reprennent leur traitant par défaut (`SIG_DFL`)

2)Emission

```
int kill(pid_t pid, int sig)
```

- désignation du destinataire
  - ◊ pid > 0 → signal envoyé au processus de numéro pid
  - ◊ pid = 0 → signal envoyé à tous les processus du même groupe que l'émetteur
  - ◊ pid = -1 → non défini
  - ◊ pid < -1 → signal envoyé à tous les processus du *groupe* |pid|
- le destinataire doit avoir le même propriétaire que l'émetteur

```
unsigned int alarm(unsigned int sec);
```

entraîne l'envoi du signal `SIGALRM` au processus appelant après un délai de `sec` secondes

3)Contrôle de la réception

Attente d'un signal

```
int pause();
```

Attente d'un signal quelconque

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

positionne le masque courant à `masque` et attend un signal.
Le masque courant est restauré au retour de `sigsuspend`

Définition des masques

Opérations ensemblistes

```
int sigemptyset(sigset_t *set);       /*   *set = {}          */
int sigfillset(sigset_t *set);        /*   *set = {1..NSIG}   */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Affectation du masque courant :

```
int sigprocmask(int op, const sigset_t *set, sigset_t *oldSet);
```

- `op = SIG_BLOCK` → `set` est ajouté au masque courant ;
- `op = SIG_UNBLOCK` → `set` est retiré du masque courant ;
- `op = SIG_SETMASK` → `set` remplace le masque courant.

Ensemble des signaux masqués pendants

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


4 – Gestion des signaux pour les sessions interactives

Buts

- faciliter contrôle de sessions interactives
- factoriser la gestion de processus "liés" (démon + fils)

Organisation

- session = { groupes } = {{processus}}
- les groupes, ainsi que les sessions, sont disjoints
- groupes et sessions sont identifiés par leur créateur(*leader*)
- par défaut, groupes et sessions s'héritent
- un périphérique *peut* être associé à une session. Alors :
  - ◊ ce périphérique est le *terminal de contrôle* de la session ;
  - ◊ un unique groupe (groupe en *premier plan*) au plus *peut* interagir avec le terminal :
    - lire/écrire sur le terminal
    - capter *(*signaux) la frappe de : *intr, quit, susp (Ctrl-C,\,Z)*
  - ◊ les autres groupes (en arrière plan)
    - ignorent les signaux précédents, et
    - sont suspendus en cas de demande d'accès au terminal
  - ◊ lorsque le leader d'une session se termine, *SIGHUP* est diffusé aux membres de la session.
  - ◊ un périphérique peut être attaché à une session au plus
  - ◊ seul le leader peut définir le terminal de contrôle

Opérations

- gestion (création, test, affectation) des groupes et sessions : **`getpgrp, setpgid, getsid, setsid`**
- manipulation du groupe en avant plan : **`tcgetssid, tcgetpgrp, tcsetpgrp`**

5 – Transfert de contrôle asynchrone

But : offrir au programmeur un mécanisme (service) logiciel de commutation de contexte

Opérations de base

- sauvegarde du contexte du traitement courant
- remplacement du contexte courant par un contexte préalablement sauvegardé

Exemple (API système - UNIX -**) : bibliothèque *setjmp.h***

- *Sauvegarder* le contexte courant dans 1 zone mémoire (`sv_cntxt`) → `cr := setjmp(sv_cntxt)`
- Restaurer (et commuter avec) un contexte (sauvé dans `sv_cntxt`) → `longjmp(sv_cntxt, cr)`

Remarques

- Pour des raisons d'efficacité, la sauvegarde et la restauration ne portent que sur une partie du contexte des processus (pile d'appel, registres, et une partie du mot d'état programme). En particulier, les variables globales et le tas ne sont ni sauvegardés ni restaurés.
- En cas de restauration (appel à *longjmp*), le programme reprend son exécution comme s'il venait d'exécuter *setjmp*. La valeur (`cr`) renvoyée par *setjmp* permet de distinguer la sauvegarde (`0` pour l'appel à *setjmp*) de la restauration (valeur (≠`0`) renvoyée par *longjmp*).
- L'API POSIX (fonctions *sigsetjmp* et *siglongjmp*) est similaire (mais pas identique)

Exemple

```
#include <setjmp.h>
int val;
jmp_buf env;
...
/* sauvegarde d'un point de reprise */
val = setjmp(env);
if (val==0) {
   ...
   /* Cascade d'appels procéduraux  */
       ....
       /* Détection d'un problème et retour au point de reprise */
       longjmp(env, 1);
   ...
} else {
   /* traitement après longjmp  */
}
```


6 – Signaux et temps réel

1) Signaux et primitives de temporisation (`<sys/time.h>`)

Il est possible de programmer des temporisations avec un grain plus fin que ce que permet **`alarm()`**

Structures de données

```
struct timeval {
       long    tv_sec;         /* seconds */
       long    tv_usec;        /* microseconds */
};
```

```
struct  itimerval {
       struct  timeval it_interval; /* période */
       struct  timeval it_value; /* instant de départ (0 = jamais)*/
};
```

Horloges et signaux

- **`ITIMER_REAL`**temps physique (réel)émet **`SIGALRM`**
- **`ITIMER_VIRTUAL`**temps d'exécution en mode utilisateurémet **`SIGVTALRM`**
- **`ITIMER_PROF`**temps d'exécution totalémet **`SIGPROF`**

Primitives

```
int getitimer(int horloge, struct itimerval *val)
int setitimer(int horloge, struct itimerval *val, struct itimerval *oldval)
```


2)Signaux temps réel

- définis dans la norme POSIX 1.b
- les signaux temps réel
  - ◊ sont mémorisés (conservés dans des files)
  - ◊ ont une priorité, correspondant à leur numéro
  - ◊ peuvent être accompagnés de données spécifiques