Les signaux

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2021-2022

Cours n°7

Rappels sur les signaux

Programmation des signaux

Version 11 janvier 2022

Table des matières

Les signaux

- Messages envoyés par le noyau à un ou plusieurs processus
- Permettent de communiquer entre processus ou entre le système et les processus :
  ↪ Pas de donnée associée, juste un numéro
- Sous *Windows* : les "*events*"
- Les effets à la réception dépendent du signal :
  ↪ Peut arrêter, stopper ou redémarrer le processus
  ↪ Peut être ignoré
- L'arrêt peut être géré différemment :
  ↪ Arrêt "brutal"
  ↪ Génération d'un *core* (copie de la mémoire du processus)
  ↪ Actions définies par l'utilisateur

PCB et signaux

- `signal` (type `sigset_t`) :
  ↪ Signaux reçus pas le processus
  ↪ Codé sur 64 octets, 1 bit par signal (1 pour reçu, 0 sinon)
- `blocked` (type `sigset_t`) :
  ↪ Signaux bloqués dont la prise en compte est retardée
- `sigpending` : drapeau indiquant si un signal au moins est bloqué en attente
- `gsig` : pointeur vers l'action associée à chaque signal

Un seul bit est utilisé par signal : pas de gestion de réceptions multiples !

Numéro de signal

- Signaux numérotés de 1 à 64 :
  ↪ 1 à 32 : signaux dits “classiques”
  ↪ 34 à 64 : signaux dits “temps réel”
- Associés à des constantes :
  ↪ Préfixe `SIG` suivi du nom
  ↪ Ne pas utiliser directement les numéros !
- En C : constantes donc majuscules
- Sous le terminal : en minuscules ou majuscules (ou les deux)
  ↪ Dépend de l’implémentation

La description et/ou la définition de certains signaux n’est pas POSIX !

Liste des signaux et description (1/2)

Numéro	Nom	Description
1	SIGUP	Terminaison du processus leader de la session
2	SIGINT	Interruption du clavier (CTRL+C)
3	SIGQUIT	Caractère QUIT depuis le clavier (CTRL+\)
4	SIGILL	Instruction illégale
5	SIGTRAP	Point d'arrêt pour le débogage
6	SIGABRT	Terminaison anormale
7	SIGBUS	Erreur de bus
8	SIGFPE	Erreur mathématique virgule flottante
9	SIGKILL	Terminaison forcée du processus
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Accès mémoire invalide
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Écriture dans un tube sans lecteur
14	SIGALRM	Fin de temporisation alarme
15	SIGTERM	Terminaison du processus
16	SIGSTKFLT	Erreur de pile du coprocesseur

Liste des signaux et description (2/2)

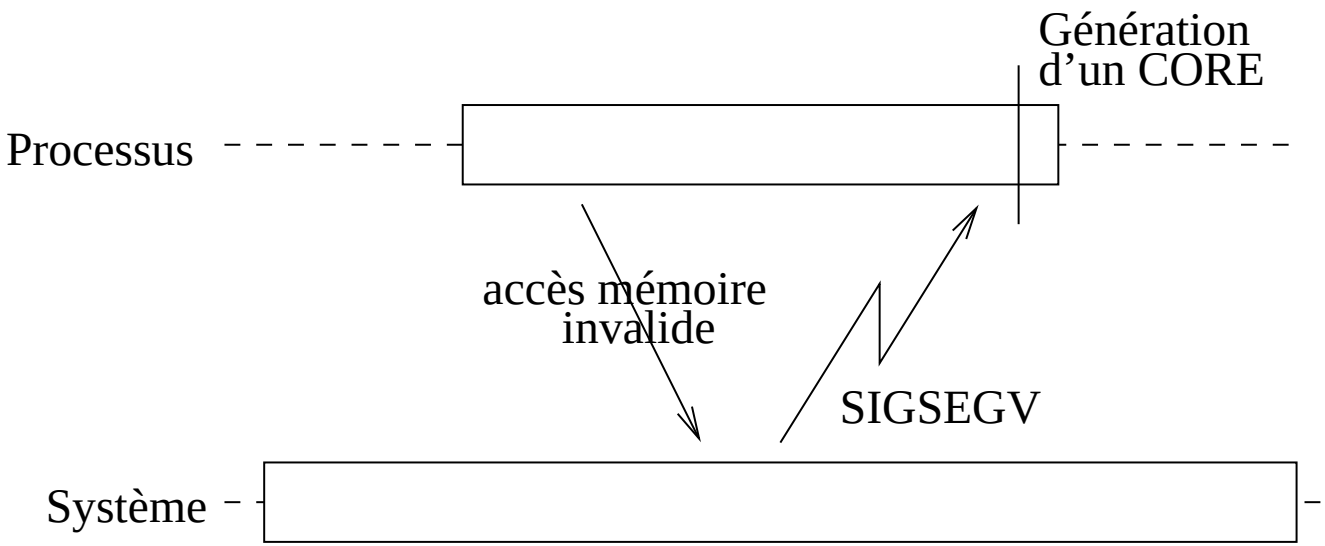
Numéro	Nom	Description
17	SIGCHLD	Terminaison du processus fils
18	SIGCONT	Reprise de l'exécution d'un processus stoppé
19	SIGSTOP	Stoppe l'exécution d'un processus
20	SIGSTP	Caractère suspend depuis le clavier (CTRL+Z)
21	SIGTTIN	Lecture par un processus en arrière-plan
22	SIGTTOU	Écriture par un processus en arrière-plan
23	SIGURG	Données urgentes dans une socket
24	SIGXCPU	Limite de temps processus dépassée
25	SIGXFSZ	Limite de taille de fichier dépassée
26	SIGVALRM	Alarme virtuelle
27	SIGPROF	Alarme du profileur
28	SIGWINCH	Fenêtre redimensionnée
29	SIGIO	Arrivée de caractères à lire
30	SIGPOLL	Équivalent à SIGIO
31	SIGPWR	Chute d'alimentation
32	SIGUNUSED	Signal non utilisé

Actions par défaut

Nom des signaux	Action par défaut
SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALARM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALARM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGUNUSED	Abandon
SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIGSEGV	Abandon + fichier *core*
SIGCHLD, SIGURG, SIGWINCH	Ignoré
SIGSTOP, SIGSTP, SIGTTIN, SIGTTOU	Processus stoppé
SIGCONT	Processus redémarré

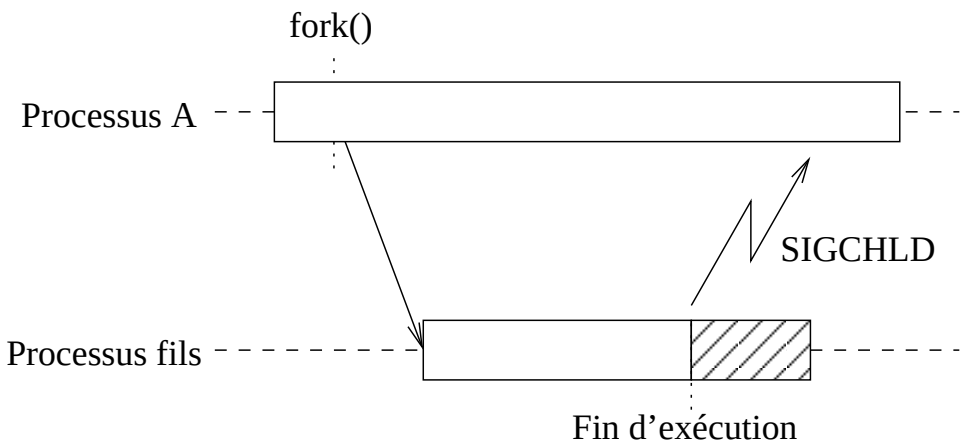
Violation d'accès mémoire : le signal "SIGSEGV"

- Si un processus accède à une zone mémoire *interdite*, le système d'exploitation empêche cet accès :
  - Envoi d'un signal "SIGSEGV" au processus fautif
  - Copie mémoire du processus réalisée dans un fichier (*core*)
  - Processus tué



La fin d'exécution d'un fils : le signal "SIGCHLD"

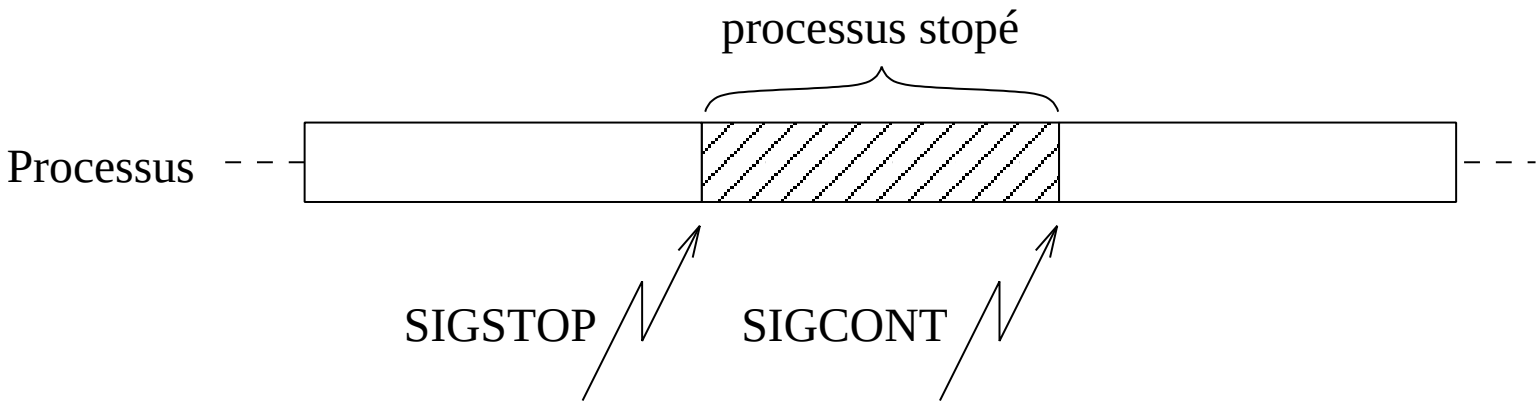
- Lorsqu'un fils termine son exécution, signal "SIGCHLD" envoyé à son père
  ↪ Le fils devient un processus *zombie* (*cf* cours sur les processus)
- Lorsque le père capte le signal, le fils est définitivement détruit.



Une application qui crée des processus (exemple d'un serveur multi-clients) doit capturer les signaux "SIGCHLD" pour éviter une saturation mémoire.

Stopper un processus (et redémarrage)

- Stopper l'exécution d'un processus :
  ↪ Signal "`SIGSTOP`" ou "`SIGSTP`" (CTRL+Z depuis le clavier)
- Reste en mémoire mais ne consomme plus de CPU
- Pour continuer son exécution, il faut lui envoyer le signal "`SIGCONT`"



Envoyer des signaux depuis le terminal

- Envoi de signaux avec la commande `kill` :

  ```
  kill -sigstop 1234
  ```

  ↪ Le processus dont l'identifiant est 1234 est stoppé
- Pour lister les signaux disponibles : `kill -l`
- Pour vérifier la présence d'un processus : `kill -0 pid`

  ↪ Si le processus n'existe pas, une erreur est générée

Terminer un processus

- Fermer la fenêtre correspondant à l'application, si elle existe
- Si le shell est en attente de la fin de l'exécution, presser CRTL + C.
- Si le processus est exécuté en tâche de fond :
  `kill -sigkill pid` (avec le PID du processus)

- Quelle que soit la méthode utilisée (exécution en tâche de fond ou non), la fermeture de l'invite de commandes implique l'arrêt de tous les processus en cours d'exécution
- La fermeture d'un père ne tue pas ses fils :
  $\hookrightarrow$ Leur PPID devient 1 (leur père devient le processus `init`)

Stopper et reprendre un processus

- Si le processus n'est pas exécuté en tâche de fond :
  - Presser CRTL + Z (ou `kill -sigstop pid` depuis un autre terminal)
  - L'utilisateur reprend la main dans le shell mais le processus est stoppé
  - Pour que le processus soit exécuté à nouveau :
    - Taper la commande `bg` (pour *background*)
    - *Ou* taper la commande `kill -sigcont pid` avec le PID du processus
- Si le processus est exécuté en tâche de fond :
  - `kill -sigstop pid` avec le PID du processus :
    ↪ Le processus est stoppé
  - `kill -sigcont pid` avec le PID du processus :
    ↪ Le processus reprend son exécution

Gestion des signaux

- La réception d'un signal induit un comportement par défaut :
  ↪ Dépend du signal
- Possible de modifier ces comportements :
  ↪ Spécifier une procédure à appeler : un gestionnaire (`handler`)
  ↪ Ignorer le signal
  ↪ Replacer le comportement par défaut
- Utilisation de la fonction `sigaction`
- Pour envoyer un signal à un processus/groupe, fonction `kill`

Il est impossible de bloquer les signaux "`SIGKILL`" et "`SIGSTOP`".
Certaines opérations sont déconseillées (comportements indéfinis).

Le positionnement «fiable» des signaux

- `signal` permet de placer un gestionnaire pour un signal
- Problème : comportements non normalisés
  ↪ Possibilité d'avoir des comportements différents suivant les systèmes
- Exemples :
  ↪ Selon certains systèmes, gestionnaire replacé par défaut à la réception d'un message
  ↪ Que se passe-t-il si deux signaux sont reçus ?
- Utilisation d'une fonction normalisée : `sigaction`
  ↪ Permet de spécifier les signaux bloqués pendant l'appel du gestionnaire

Il est formellement interdit d'utiliser la fonction `signal` dans vos codes !

Gestion d'ensembles de signaux

- Possibilité de gérer des ensembles de signaux
- Utilisation de la structure `sigset_t` (qui est un ensemble)
- Ensemble de fonctions pour gérer les ensembles :
  ↪ Ajout, suppression...
- Intérêts :
  ↪ Bloquer des ensembles de signaux
  ↪ Placer un gestionnaire pour un ensemble de signaux
  ↪ Conserver le positionnement de tous les signaux avant modification

Gestion de `sigset_t` : en-têtes (1/2)

- `int sigemptyset(sigset_t *set)` :
  ↪ Vide l'ensemble (tous les signaux sont désélectionnés)
- `int sigfillset (sigset_t *set)` :
  ↪ Remplit l'ensemble (tous les signaux sont sélectionnés)
- `int sigaddset(sigset_t *set, int signum)` :
  ↪ Ajoute le signal "`signum`" à l'ensemble "`set`"
- `int sigdelset(sigset_t *set, int signum)` :
  ↪ Supprime le signal "`signum`" de l'ensemble "`set`"
- `int sigismember(const sigset_t *set, int signum)` :
  ↪ Indique si le signal "`signum`" appartient à l'ensemble "`set`
- *Inclusion* : `signal.h`

Gestion de `sigset_t` : en-têtes (2/2) : retour et erreurs

- Pour `sigismember` :
  ↪ 1 si le signal est dans l'ensemble, 0 sinon et -1 en cas d'erreur
- Pour les autres :
  ↪ 0 en cas de réussite et -1 en cas d'erreur
- Seule erreur :
  ↪ `EINVAL` : numéro de signal invalide

Fonction `sigaction` (1/2)

En-tête de la fonction (S2)

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`
- *Inclusion* : `signal.h`

Paramètre(s)

- `signum` : le signal concerné (sauf "`SIGKILL`" et "`SIGSTOP`")
- `act` : action associée au signal (si non nul)
- `oldact` : ancienne action à sauvegarder (si non nul)

Fonction `sigaction` (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EINVAL` : signal indiqué invalide (par exemple "`SIGKILL`", "`SIGSTOP`")
  - `EINTR` : appel système interrompu par un signal

Structure sigaction (1/2)

```
struct sigaction {
  void      (* sa_handler) (int);
  void      (* sa_sigaction) (int, siginfo_t *, void *);
  sigset_t  sa_mask;
  int       sa_flags;
}
```

Champs

- sa_handler : gestionnaire à placer (fonction, SIG_IGN ou SIG_DFL)
- sa_sigaction : utilisé pour les signaux temps réel (voir dans la suite)
- sa_mask : ensemble de signaux à bloquer pendant l'exécution du gestionnaire (signal concerné aussi sauf attribut contraire)

Sur certains systèmes, il s'agit d'une union donc ne pas utiliser simultanément les champs sa_handler et sa_sigaction

Structure sigaction (2/2)

```
struct sigaction {
  void      (* sa_handler) (int);
  void      (* sa_sigaction) (int, siginfo_t *, void *);
  sigset_t  sa_mask;
  int       sa_flags;
}
```

Champs (suite)

- sa_flags : attributs permettant de modifier le comportement du gestionnaire
  - SA_RESETHAND : rétablir action par défaut une fois le gestionnaire appelé (SA_ONESHOT est obsolète)
  - SA_NODEFER : ne pas empêcher un signal d'être reçu pendant l'exécution de son gestionnaire (SA_NOMASK est obsolète)
  - SA_SIGINFO : champ sa_sigaction à utiliser plutôt que sa_handler
  - SA_RESTART : relance l'appel système interrompu par l'exécution du gestionnaire

Exemple d'utilisation

```
void gestionnaire(int signum) {
    printf("J'ai␣reçu␣un␣signal\n");
}

int main() {
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = gestionnaire;

    if(sigaction(SIGINT, &action, NULL) == -1) {
        perror("Erreur␣lors␣du␣positionnement␣du␣gestionnaire␣");
        exit(EXIT_FAILURE);
    }

    printf("Pressez␣CRTL+C\n");
    pause();
    printf("Je␣suis␣dans␣le␣main\n");

    exit(EXIT_FAILURE);
}
```

Fonction `kill`

En-tête de la fonction (S2)

- `int kill(pid_t pid, int num_sig)`
- *Inclusions* : `signal.h` pour la fonction, `sys/types` pour les types

Paramètre(s)

- `pid` : *pid* du processus ou d'un groupe
- `num_sig` : numéro du signal à envoyer

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EINVAL` : numéro de signal invalide
  - `ESRCH` : processus ou groupe inexistant

Fonction `kill` (suite)

Autres valeurs possibles pour `pid`

- 0 : signal envoyé à tous les processus du groupe
- -1 : signal envoyé à tous les processus accessibles (sauf le 1)
- $<$ -1 : signal envoyé au groupe dont l'identifiant est -`pid`
- Dans tous les cas :
  ↪ Le processus doit avoir les privilèges nécessaires
  ↪ Ou le même UID (sauvé ou réel) que le processus cible
  ↪ Pour "`SIGCONT`" : les deux processus doivent être dans la même session

Vérifier la présence d'un processus/groupe de processus

- Utilisation du signal numéro 0 :
  ↪ Aucun signal envoyé...
  ↪ ...mais gestion des erreurs réalisée

Exemple (1/2) : attente des signaux

```
int cpt = 0; /* Une variable globale */

void gestionnaire(int signum) {
    if(signum == SIGUSR1) cpt = cpt | 1;
    if(signum == SIGUSR2) cpt = cpt | 2;
}

int main() {
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = gestionnaire;

    sigaction(SIGUSR1, &action, NULL);
    sigaction(SIGUSR2, &action, NULL);

    printf("Prêt à recevoir des signaux, PID=%d\n", getpid());
    while(cpt != 3)
        sleep(1);

    return EXIT_SUCCESS;
}
```

Exemple (2/2) : envoi des signaux

```
int main(int argc, char *argv[]) {
    pid_t pid;

    /* Vérification des arguments */
    pid = atoi(argv[1]);

    if(kill(pid, 0) == -1) {
        if(errno == ESRCH) {
            fprintf(stderr, "Le␣processus␣avec␣le␣PID␣%d␣n'existe␣pas\
                n", pid);
            exit(EXIT_FAILURE);
        }
        else {
            perror("Erreur␣lors␣de␣l'envoi␣du␣signal␣");
            exit(EXIT_FAILURE);
        }
    }
    kill(pid, SIGUSR1);
    kill(pid, SIGUSR2);

    return EXIT_SUCCESS;
}
```

Pause et alarmes

- Pause :
  - Possible d'endormir un processus jusqu'à réception d'un signal :
    ↪ `pause`
  - Ou pendant au moins un certain temps :
    ↪ `sleep` et `nanosleep`
- Alarme :
  - Possible de placer une alarme avec `alarm` :
    ↪ Un signal "`SIGALARM`" est reçu après le délai spécifié
  - Annulation d'une alarme `alarm(0)`

Il est déconseillé d'utiliser à la fois `alarm` et `sleep`
car l'implémentation de `sleep` peut utiliser les alarmes.

Fonction `sleep`

En-tête de la fonction (S3)

- `unsigned int sleep(unsigned int nbS)`
- *Inclusion* : `unistd.h`

Explications

- Endort le processus jusqu'à ce que `nbS` secondes soient écoulées
- Interrompu lorsqu'un signal (non ignoré) est reçu

Paramètre(s)

- `nbS` : nombre de secondes à attendre

Valeurs retournées et erreurs générées

- Retourne 0 si toutes les secondes sont écoulées, ou le nombre de secondes restantes s'il est interrompu

Fonction `nanosleep`

En-tête de la fonction (S2)

- `int nanosleep(const struct timespec *req, struct timespec *rem);`
- *Inclusion* : `time.h`

Explications

- Endort le processus jusqu'à ce que le temps indiqué dans `*req` soit écoulé

Fonction `nanosleep` (suite)

Paramètre(s)

- `req` : nombre de secondes / nanosecondes à attendre
- `rem` : temps restant si interrompu

```
struct timespec {
    time_t tv_sec;        /* secondes */
    long   tv_nsec;       /* nanosecondes */
}
```

Valeurs retournées et erreurs générées

- Retourne 0 si le temps est écoulé, -1 sinon
- Erreurs possibles :
  - `EINTR` : interruption par un signal
  - `EINVAL` : valeurs du paramètre `req` incorrectes

Exemple : sleep et nanosleep

```
int main() {
    struct timespec temps = { 1, 500000000 };

    printf("Je me mets en pause pendant 1s...\n");
    sleep(1);

    printf("Je me mets en pause pendant 1.5s...\n");
    if(nanosleep(&temps, NULL) == -1) {
        perror("Erreur lors de l'initialisation de la pause ");
        exit(EXIT_FAILURE);
    }

    return EXIT_FAILURE;
}
```

Fonction `pause`

En-tête de la fonction (S2)

- `int pause(void)`
- *Inclusion* : `unistd.h`

Explications

- Endort le processus jusqu'à ce qu'un signal soit reçu :
  ↪ Le signal doit interrompre le processus ou provoquer l'appel à un gestionnaire
- Rend la main une fois le gestionnaire terminé

Valeurs retournées et erreurs générées

- Toujours -1
- Seule valeur possible pour l'erreur : `EINTR`

Fonction `alarm`

En-tête de la fonction (S2)

- `unsigned int alarm(unsigned int nb_sec)`
- *Inclusion* : `unistd.h`

Explications

- Programme une temporisation pour envoyer un signal "`SIGALRM`" dans `nb_sec` secondes

Paramètre(s)

- `nb_sec` : nombre de secondes

Valeurs retournées et erreurs générées

- 0 ou le nombre de secondes qu'il reste de la programmation précédente

Délais variables pour le multitâche !

Exemple : alarm et pause

```
void gestionnaire(int signum) {
    printf("J'ai␣reçu␣une␣alarme\n");
}

int main() {
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = gestionnaire;
    sigaction(SIGALRM, &action, NULL);

    alarm(3);

    printf("J'attends␣l'alarme\n");
    pause();
    printf("OK,␣j'ai␣fini\n");

    return EXIT_FAILURE;
}
```

Bloquer et débloquer des signaux (1/2)

En-tête de la fonction (S2)

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
- *Inclusion* : `signal.h`

Explications

- Permet de récupérer et/ou de modifier le masque des signaux bloqués
- Si `set` n'est pas nul, modifie le masque
- Si `oldset` n'est pas nul, récupère le masque précédent (ou en cours si `set` est nul)
- L'opération dépend du paramètre `how`

Bloquer et débloquer des signaux (2/2)

Paramètre(s)

- `how` peut prendre les valeurs suivantes :
  - `SIG_BLOCK` : les signaux de l'ensemble "`set`" sont ajoutés à l'ensemble des signaux déjà bloqués
  - `SIG_UNBLOCK` : les signaux de l'ensemble "`set`" sont supprimés de l'ensemble des signaux bloqués
  - `SIG_SETMASK` : les signaux bloqués sont exactement ceux de l'ensemble "`set`"
- `set` : un ensemble de signaux
- `oldset` : s'il n'est pas nul, la valeur précédente du masque des signaux est placée dans "`oldset`"

Valeurs retournées et erreur générée

- Retourne 0 si l'appel réussi, -1 sinon
- Erreur EINVAL si `how` est invalide

Exemple d'utilisation

```
int main() {
  sigset_t sigs_new, sigs_old;

  sigfillset(&sigs_new);
  sigdelset(&sigs_new, SIGINT); sigdelset(&sigs_new, SIGQUIT);
  sigprocmask(SIG_BLOCK, &sigs_new, &sigs_old);

  // Ici tous les signaux sont bloqués sauf SIGINT et SIGQUIT

  sigprocmask(SIG_SETMASK, &sigs_old, NULL);
  sigemptyset(&sigs_new);
  sigaddset(&sigs_new, SIGINT); sigaddset(&sigs_new, SIGQUIT);
  sigprocmask(SIG_BLOCK, &sigs_new, &sigs_old);

  // Ici seuls les signaux SIGINT et SIGQUIT sont bloqués

  sigprocmask(SIG_SETMASK, &sigs_old, NULL);

  // Suite du code

  return EXIT_SUCCESS;
}
```

Fonction `sigpending`

En-tête de la fonction (S2)

- `int sigpending(sigset_t *set)`
- *Inclusion* : `signal.h`

Explications

- Récupère la liste des signaux déclenchés pendant leur blocage

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Erreurs possibles :
  - `EFAULT` : `set` pointe en dehors de l'espace d'adressage accessible
  - `EINTR` : interrompu par un signal

Exemple d'utilisation

```
sigset_t sigs_new, sigs_blocked;
struct sigaction action;
unsigned int temps = 10;

sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = gestionnaire;
sigaction(SIGUSR1, &action, NULL);
sigaction(SIGUSR2, &action, NULL);

sigfillset(&sigs_new);
sigdelset(&sigs_new, SIGUSR1);
sigprocmask(SIG_BLOCK, &sigs_new, NULL);

while(temps != 0) temps = sleep(temps);

sigpending(&sigs_blocked);
if(sigismember(&sigs_blocked, SIGUSR1))
    printf("J'ai␣reçu␣SIGUSR1␣pendant␣ma␣pause.\n");
if(sigismember(&sigs_blocked, SIGUSR2))
    printf("J'ai␣reçu␣SIGUSR2␣pendant␣ma␣pause.\n");

// Seul le signal SIGUSR2 pourra être détecté
```

Fonction `sigsuspend`

En-tête de la fonction (S2)

- `int sigsuspend(const sigset_t *set)`
- *Inclusion* : `signal.h`

Explications

- Modifie le masque des signaux + mise en attente du processus :
  ↪ Réalisation atomique
- Processus réveillé si un signal non bloqué est reçu
- Masque antérieur replacé

Valeurs retournées et erreur générée

- Retourne 0 ou -1 en cas d'échec
- Erreurs possibles : comme pour *sigpending*

Exemple d'utilisation : le «serveur»

```
int main() {
    sigset_t sigs_new, sigs_old, sigs_wait;
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = gestionnaire;
    sigaction(SIGUSR1, &action, NULL);

    sigfillset(&sigs_new);
    sigprocmask(SIG_BLOCK, &sigs_new, &sigs_old);

    sigfillset(&sigs_wait);
    sigdelset(&sigs_wait, SIGUSR1);
    printf("Prêt à recevoir SIGUSR1, PID=%d\n", getpid());
    if(sigsuspend(&sigs_wait) == -1) {
        if(errno != EINTR) {
            perror("Erreur lors de l'attente du signal ");
            exit(EXIT_FAILURE);
        }
    }
    return EXIT_SUCCESS;
}
```

Exemple d'utilisation : le «client»

```
int main(int argc, char *argv[]) {
    pid_t pid;

    if(argc != 2) {
        fprintf(stderr, "Usage␣:␣%s␣PID␣où␣PID␣est␣le␣PID␣du␣programme
            ␣'serveur'\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    pid = atoi(argv[1]);

    if(kill(pid, SIGUSR1) == -1) {
        perror("Erreur␣lors␣de␣l'envoi␣du␣signal␣");
        exit(EXIT_FAILURE);
    }
    printf("Signal␣SIGUSR1␣envoyé.\n");

    return EXIT_SUCCESS;
}
```

Les signaux dits "*temps-réel*"

- Signaux additionnels normalisés POSIX.1b
- Pas de nom associé :
  ↪ Utilisation de numéros
  ↪ Constantes "`SIGRTMIN`" et "`SIGRTMAX`"
- Différence principale :
  ↪ Possibilité de mémoriser les occurrences successives
  ↪ Possible d'associer des (petites) données
- File d'attente associée à chaque signal (limitée à 1024)
- Délivrance des signaux suivant la priorité de ceux-ci :
  ↪ Petit numéro = plus prioritaire

Fonction `sigqueue` (1/2)

En-tête de la fonction (S2)

- `int sigqueue(pid_t pid, int sig, const union sigval valeur)`
- *Inclusion* : `signal.h`

Description

- Envoie un signal avec des informations supplémentaires

Paramètre(s)

- `pid` : *pid* du processus ou d'un groupe
- `num_sig` : numéro du signal à envoyer
- `valeur` : valeur envoyée qui peut être un entier ou un pointeur générique

Fonction `sigqueue` (2/2)

```
union sigval {
    int   sival_int;
    void *sival_ptr;
}
```

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EAGAIN` : limite du nombre de signaux atteinte
  - `EINVAL` : numéro de signal invalide
  - `EPERM` : pas la permission d'envoyer un signal au(x) processus visé(s)
  - `ESRCH` : processus ou groupe inexistant

Les gestionnaires

- Différents de ceux utilisés jusque ici :
  ↪ Nécessitent de traiter les données jointes
- Signature :

```
void gestionnaire(int num_sig, siginfo_t *info, void
                                        *rien)
```

  ↪ `num_sig` : le numéro du signal
  ↪ `info` : informations diverses
  ↪ `rien` : non normalisé POSIX donc non utilisé (`NULL`)

```
/* Structure siginfo_t (quelques champs seulement) */
siginfo_t {
    int      si_signo; /* Numéro de signal */
    int      si_code;  /* Code du signal   */
    sigval_t si_value; /* Valeur du signal */
    pid_t    si_pid;   /* PID de l'émetteur */
}
```

Positionner un gestionnaire

- Utilisation de la primitive `sigaction`
- Gestionnaire spécifié dans le champ `sa_sigaction`
  ↪ Et pas `sa_handler`
- Champ `sa_flag` doit contenir `SA_SIGINFO`

Exemple d'utilisation (1/2)

```
void gestionnaire(int num_sig, siginfo_t *info, void *rien) {
  printf("Valeur␣recue␣:␣%d\n", info->si_value.sival_int);
}

void fils() {
  struct sigaction action;

  action.sa_sigaction = gestionnaire;
  sigemptyset(&action.sa_mask);
  action.sa_flags = SA_SIGINFO;
  sigaction(SIGRTMIN + 1, &action, NULL);

  pause();

  exit(EXIT_SUCCESS);
}
```

Exemple d'utilisation (2/2)

```
int main() {
  pid_t pid;
  union sigval valeur;

  if((pid = fork()) == 0)
    fils();

  sleep(2);

  valeur.sival_int = 1234;
  sigqueue(pid, SIGRTMIN + 1, valeur);

  wait(NULL);

  return EXIT_SUCCESS;
}
```

Réduire les commutations de contexte

- À la réception d'un signal : plusieurs commutations de contexte
  ↪ Une pour gérer le(s) signal/signaux (mode noyau)
  ↪ Une pour exécuter le gestionnaire (mode utilisateur)
- Possible de les éviter :
  ↪ `sigwaitinfo` et `sigtimedwait`
- Dans les deux cas : processus mis en attente de la réception d'un signal
- L'exécution reprend ensuite normalement :
  ↪ Possible d'exécuter alors un gestionnaire manuellement
  ↪ Un simple appel de procédure !

Fonctions `sigwaitinfo` et `sigtimedwait` (1/2)

En-tête des fonctions (S2)

- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)`
- `int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`
- *Inclusion* : `signal.h`

Paramètre(s)

- `set` : ensemble des signaux attendus
- `info` : informations reçues
- `timeout` : limite maximale à attendre

```
struct timespec {
    long tv_sec;  /* secondes */
    long tv_nsec; /* nanosecondes */
}
```

Fonctions `sigwaitinfo` et `sigtimedwait` (2/2)

Valeurs retournées et erreurs générées

- Retourne le numéro du signal reçu ou -1 en cas d'erreur
- Erreurs générées :
  - `EAGAIN` : aucun signal reçu avant l'expiration du compte-à-rebours
  - `EINVAL` : valeur du compte-à-rebours invalide
  - `EINTR` : interruption par un gestionnaire de signal

Exemple d'utilisation

```
void fils() {
    sigset_t ensemble;
    siginfo_t info;

    sigfillset(&ensemble);
    sigprocmask(SIG_BLOCK, &ensemble, NULL);

    sigemptyset(&ensemble);
    sigaddset(&ensemble, SIGRTMIN + 1);
    printf("Le␣fils␣:␣j'attend␣un␣signal\n");
    sigwaitinfo(&ensemble, &info);
    printf("Le␣fils␣:␣j'ai␣reçu␣le␣signal␣avec␣la␣valeur␣%d\n", info.
        si_value.sival_int);

    exit(EXIT_SUCCESS);
}
```

Le gestionnaire n'est plus nécessaire.