243 exit et terminaison de processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Terminaison volontaire des processus

Terminaison normale

- Le processus a terminé son travail
- L'utilisateur a fait `fichier/quitter`
- Etc.

Terminaison suite à une erreur

- Arguments erronés pour un programme en ligne de commande
- Format de fichier non reconnu
- Etc.

Dans les deux cas

- Appel système `exit(3)`
- Retour de la fonction `main`

$\rightarrow$ Indique une valeur de retour

- Convention: $0 \rightarrow$ OK, $\neq 0 \rightarrow$ voir le man

Terminaison involontaire des processus

Terminé par un autre processus

- Via l'appel système `kill(2)`, s'il a les droits
- On y reviendra...

Erreur fatale (généralement : bogue du programme)

- Les **fautes** du CPU sont la cause principale

$\rightarrow$ Division par 0, erreur de segmentation, etc.

Terminé par le système

- Ressource manquante (mémoire)
- Arrêt du système (*shutdown*)

Gestion des signaux Unix

Dans la plupart de ces cas un processus peut être notifié pour gérer son interruption involontaire

Fin des processus

Le SE doit

- Fermer les fichiers ouverts
- Informer le parent (signal SIGCHLD)
- Faire adopter les enfants par init (ou autre)
- Marquer les zones mémoires comme libres
- Mettre à jour ses structures de données internes (statistiques et nettoyage)
- Ne pas réutiliser le PID trop tôt

Questions

- Pourquoi le SE s'occupe-t-il de faire tout ça ?
  Ne peut-il pas laisser ça au programme ?
- Pourquoi on demande aux apprentis programmeurs de libérer quand même les ressources ?

Terminer un processus

Fonction exit

- `exit(3)` « `void exit(int valeur_de_sortie)` »

Généralement, la valeur de sortie vaut

- `EXIT_SUCCESS` (0) si tout s'est bien déroulé
- `EXIT_FAILURE` (1) en cas d'erreur

Questions

- Quels sont les cas d'erreur d'exit ?
- Pourquoi exit ne retourne pas de valeur ?

Fausses sorties

- `exit(3)` est une fonction de bibliothèque (C ISO/IEC)
- Effectue des actions programmées
- Puis termine le processus

Actions programmées

- Flush les entrée-sortie de `stdio(h)`
- Supprime les fichiers créés par `tmpfile(3)`

→ `atexit(3)` ajoute une action programmée

C'est fait côté bibliothèque

- Conservées par un `fork(2)`
- Perdues par un `execve(2)`

→ Non appelé si terminaison par un signal ou une **vraie** sortie

Question

- Que se passe-t-il si une fonction enregistrée par `atexit` appelle `exit` ?

Vraies sorties

Le vrai appel système (POSIX)

- `_exit(2)` termine le processus immédiatement
- Sans faire les actions programmées

Multi-threads (POSIX)

- `pthread_exit(3)` termine le thread courant
- Mais effectue les actions programmées, si c'était le dernier thread

Le vraiment vrai appel système (Linux)

- Sous GNU Linux, `_exit(2)` appelle `exit_group(2)`
- `exit_group(2)` termine toutes les tâches (threads) du processus
- Vrai appel système Linux « `exit` », ne termine que la tâche courante
  Pas d'enveloppe dans la glibc.

Exemple: exit.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

void bye(void) { printf(", le monde!\n"); }
int main(int argc, char **argv) {
  atexit(bye);
  int i = atoi(argv[1]); // pas d'argument => segfault
  printf("Bonjour %d!\n", i);
  printf("Au revoir");
  switch(i) {
    case 0: return i;
    case 1: exit(i);
    case 2: _exit(i);
    case 3: syscall(SYS_exit_group, i);
    case 4: syscall(SYS_exit, i);
  }
}
```

Attendre la terminaison

Appel système wait

- `wait(2)` « `int wait(int *status)` »
- `waitpid(2)` « `int waitpid(int pid, int *status, int option)` »
- Permet à un parent d'attendre la fin de l'exécution d'un enfant

Arguments

- pid : l'enfant à attendre
- status : raison de terminaison (code de retour ou n° de signal)
- option : diverses options (voir man)

Exemple de wait

```
int main(int argc, char **argv) {
  pid_t pfils = fork();
  if (pfils == -1) { perror("Echec du fork"); return 1; }
  if(pfils == 0) {
    execvp(argv[1], argv+1);
    perror(argv[1]); return 1;
  }

  printf("J'attend %d...\n", pfils);
  int status;
  int w = wait(&status);
  if (w == -1) { perror("waitpid"); exit(1); }
  if (WIFEXITED(status)) {
    printf("Status=%d\n", WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    psignal(WTERMSIG(status), argv[1]);
  }
  return 0;
}
```

Processus zombi

Le SE conserve les informations d'un processus

- Raison de la terminaison
- Code de retour / numéro du signal
- Ressources consommées (voir `wait3(2)` et `wait4(2)`, non-POSIX)

$\rightarrow$ À l'intention du parent

État zombi

- Durant ce temps, le processus enfant est dans un état zombi (repéré par un Z et un *defunct* lors d'un ps)
- Coût d'un zombi : une entrée dans la table des processus
- Quand le parent s'informe (`wait(2)`), ces informations sont nettoyées

Un zombi ne consomme pas d'autre ressource

init

- Si un processus se termine, ses enfants sont hérités par `init` (tous les enfants, zombis ou non)
- `init` effectue les `wait(2)` nécessaires à leur nettoyage.

subreaper

- Sous Linux par des *subreaper* autre que `init` peuvent être définis
- Voir appel système `prctl(2)`