Université Lille1
Sciences et Technologies

Licence d'informatique
Module de Programmation des systèmes

Gestionnaire de travaux

Philippe MARQUET et Anne-Françoise LE MEUR

Ce document est le support de travaux dirigés et travaux pratiques en vue de l'implantation des fonctionnalités de gestion des travaux d'un shell.

1 Gestionnaire de travaux

Un gestionnaire de travaux reprend les fonctionnalités du shell traitant de la manipulation de travaux (*jobs*) permettant de lancer l'exécution d'un programme, en avant-plan ou en arrière plan, d'interrompre ou suspendre un processus, de relancer son exécution, etc.

Un gestionnaire de travaux est donc chargé du traitement des signaux envoyés par les saisies des caractères tels `<intr>` (`control-C`) pour les répercuter aux processus s'exécutant en avant-plan. Un gestionnaire de travaux est également en charge de la réalisation des commandes de manipulation de travaux telles `bg` qui relance en arrière-plan un processus suspendu.

L'implantation d'un gestionnaire de travaux repose principalement sur le maintien d'une structure mémorisant l'ensemble des travaux, sur la redirection aux travaux de signaux générés par le système, et sur la génération de signaux pour la réalisation des commandes de manipulation de travaux.

Travaux en avant- et arrière-plan

Un gestionnaire élémentaire est capable de manipuler un travail en avant-plan et un ensemble de travaux en arrière-plan. De plus, d'autres travaux peuvent être suspendus ; leur exécution se poursuivra quand le gestionnaire de travaux les signalera à l'aide du signal `SIGCONT`.

Exemple de session d'un gestionnaire de travaux

La session suivante est extraite d'une session shell (`tcsh`) :

```
% xclock -update 1
^Z
[1]  + Suspended                    xclock -update 1
% jobs
[1]  + Suspended                    xclock -update 1
% xeyes &
[2] 503
% jobs
[1]  + Suspended                    xclock -update 1
[2]  - Running                      xeyes
% stop %2
[2]  + Suspended (signal)           xeyes
% bg %1
[1]    xclock -update 1 &
% jobs
[1]    Running                      xclock -update 1
[2]  + Suspended (signal)           xeyes
% fg %2
```

```
xeyes
^C
% jobs
[1]  + Running                       xclock -update 1
% kill %1
% echo $SHELL
/bin/tcsh
[1]    Terminated                    xclock -update 1
% jobs
%
```

Exercice 1 (Expérimentation)
Déterminez, a priori, le comportement des commandes saisies dans cette session.
Vérifiez votre réponse en saisissant les commandes dans un terminal. □

Utilisation des signaux

La saisie des caractères `<intr>` (`control-C`), `<susp>` (`control-Z`) ainsi que les commandes `fg`, `bg`, `stop`, et `kill` sont implantées par des générations ou redirections de signaux telles que présentées à la figure.

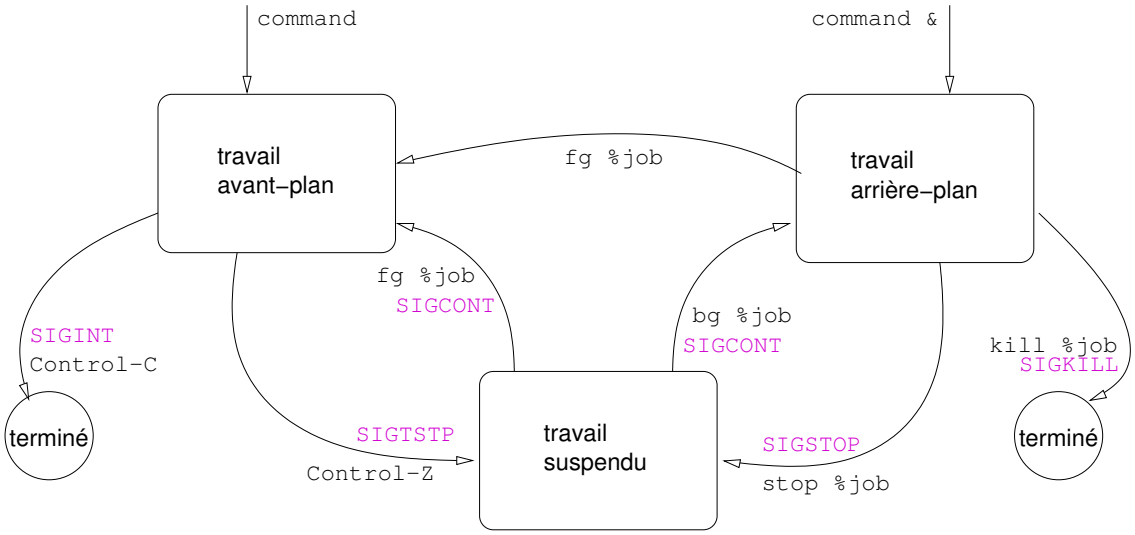

FIGURE 1 – Liens entre les commandes de gestion de travaux et les signaux

2 Implantation d'un gestionnaire de travaux

Exercice 2 (Structures de données pour les jobs et fonctions associées)
La gestion des différents jobs nécessite de maintenir des informations sur chaque job. Un job est identifié par son pid ou bien son job id, ces numéros lui sont attribués à son lancement. Ainsi, par exemple, le job `xeyes` de la session shell montrée précédemment a pour pid 503 et pour job id 2. De plus, a tout moment, on a besoin de savoir si le job est exécuté en arrière-plan (BG), en avant-plan (FG) ou s'il est stoppé (ST).

Proposez des structures de données pour représenter toutes ces informations. De plus, proposez un ensemble de fonctions permettant de manipuler ces structures. □

Organisation du code du gestionnaire de travaux

Les fichiers sources d'un squelette de l'implantation d'un gestionnaire de travaux sont disponibles à partir de la page du cours, `http://www.lifl.fr/~marquet/cnl/pds/`. Vous y trouverez :
- un `Makefile` ;
- `mshell.c` : contient entre autre le `main()` ;
- `cmd.c`, `cmd.h` : contient les commandes `fg`, `bg`, `stop`... ;
- `jobs.c`, `jobs.h` : contient la librairie gérant les structures de données associées aux jobs ;
- `sighandlers.c`, `sighandlers.h` : contient les traitants de signaux ;
- `common.c`, `common.h` : contient des données et fonctions communs aux différents fichiers.

A priori, vous n'avez à modifier que les fichiers `cmd.c` et `sighandlers.c`. Les squelettes des fonctions à implémenter sont déjà donnés ; aucune autre fonction n'est nécessaire.

Le mini-shell peut fonctionner en mode *verbose* (`mshell -v`) ce qui permet d'avoir des informations sur les handlers, fonctions sollicités, etc. Une variable globale `verbose` est prévue à cet effet. Pensez à l'utiliser ! Le fichier `jobs.c` contient un exemple de son utilisation, à vous de voir selon les situations quelles informations afficher.

Traitants de signal

Il est nécessaire de rediriger les signaux `SIGINT`, `SIGTSTP` et `SIGCHLD` envoyés au mini-shell par le système. Un signal `SIGINT` est envoyé à chaque fois que l'utilisateur fait un `control-C` au clavier. Un signal `SIGTSTP` est envoyé lors d'un `control-Z`. Enfin le signal `SIGCHLD` est envoyé à chaque fois que l'exécution d'un programme lancé dans le mini-shell a terminé ou a été bloquée.

Exercice 3
Donnez l'implémentation du traitant de signal pour `SIGINT` :

```
void sigint_handler(int sig);
```

Ce traitant redirige le signal vers le processus qui est en avant-plan afin de provoquer sa terminaison. □

Exercice 4
Donnez l'implémentation du traitant de signal pour `SIGTSTP` :

```
void sigstp_handler(int sig);
```

Ce traitant redirige le signal vers le processus qui est en avant-plan afin de le suspendre. □

Exercice 5
Donnez l'implémentation du traitant de signal pour `SIGCHLD` :

```
void sigchld_handler(int sig);
```

Ce traitant recherche tous les jobs terminés ou stoppés. Il détermine si le job a été stoppé, terminé par un signal ou a terminé normalement son exécution. Selon les cas, les structures de données sont mises à jour de façon appropriée et un message est affiché sur la sortie standard pour renseigner sur l'état du job. □

Exercice 6
Il est nécessaire d'installer ces traitants de signal pour personnaliser le traitement des signaux `SIGINT`, `SIGTSTP` et `SIGCHLD`. Donnez l'implémentation de la fonction `signal_wrapper()`

```
typedef void handler_t(int);
int signal_wrapper(int signum, handler_t *handler);
```

qui permet d'associer un traitant à un signal donné. □

Commandes du mini-shell

Le mini-shell offre un certain nombre de commandes :
- `exit` permet de quitter le mini-shell
- `stop` permet d'arrêter un job
- `kill` permet de provoquer la terminaison d'un job
- `bg` envoie un job en arrière-plan
- `fg` met un job en avant-plan
- `jobs` donne des informations sur les jobs courants

Exercice 7
La commande `fg` est invoquée soit par `fg` *`pid`* ou `fg` *`jid`*

Le prototype de la fonction implémentant la commande *fg* est le suivant :

```
void do_fg(char **argv);
```

`argv` contient la chaîne de caractères correspondant à la commande tapée dans le mini-shell, comme par exemple `"fg %1"`. Une fonction `treat_argv()` est mise à votre disposition pour chercher un job dans la liste des travaux en fonction d'un pid ou d'un jid. Son prototype est :

```
struct job_t * treat_argv(char **argv);
```

La valeur `NULL` est retournée si aucun job n'a été trouvé.

Lorsqu'un job est exécuté en avant-plan, l'utilisateur n'a pas la main. Pour pouvoir implémenter ce comportement, écrivez une fonction

```
void waitfg(pid_t pid)
```

qui bloque tant le job `pid` n'est pas mis en arrière-plan.

La fonction `do_fg()` envoie le signal approprié au job qui doit être passé en avant-plan, met à jour les informations associées au job et bloque tant que le job n'est pas passé en arrière-plan. Donnez son implémentation.

Les commandes `bg`, `stop` et `kill` sont invoquées sur le même modèle que `fg` et sont respectivement implémentées par :

```
void do_bg(char **argv);
void do_stop(char **argv);
void do_kill(char **argv);
```

Donnez leurs implémentations. □

Exercice 8
La commande `exit` permet de sortir du mini-shell. Le prototype de la fonction qui implémente `exit` est le suivant :

```
void do_exit();
```

Inspirez-vous du comportement du shell que vous utilisez habituellement (cette remarque est d'ailleurs aussi valable pour toutes les autres commandes). Par exemple, que se passe-t-il lorsque vous faites un exit dans un shell alors qu'un job de ce shell est arrêté ? □

Exercice 9
La commande `jobs` affiche l'état de tous les jobs lancés dans le mini-shell, voyez l'exemple donné en début de document.

La fonction qui implémente cette commande a pour prototype :

```
void do_jobs();
```

Implémentez cette fonction. □

3 Travaux communicants par tubes

Il s'agit d'ajouter la possibilité de traiter dans notre gestionnaire d'enchaînements de travaux dont les entrées/sorties standard sont redirigées vers/depuis des tubes anonymes.

On pourra ainsi traiter des commandes telles

```
% ls -l | wc -l
34
```

Exercice 10 (Connexion de deux commandes par un tube anonyme)
Il s'agit d'implanter une commande

```
pipe command1 to command2
```

qui exécute les commandes `command1` et `command2` en redirigeant la sortie standard de La première commande vers un tube anonyme depuis lequel sera redirigée l'entrée standard de la seconde commande. □

Exercice 11 (Des tubes dans le gestionnaire de travaux)
Question 11.1 (Un tube, deux commandes) Modifiez le gestionnaire de travaux pour prendre en compte la possibilité de lancer des commandes selon les deux syntaxes suivantes

```
% commande1 | commande2
```

et

```
% commande1 | commande2 &
```

une telle commande ne crée qu'un unique travail, composé de deux processus. □

Question 11.2 (Des tubes et des commandes) Étendez votre implantation de l'exercice précédent pour autoriser l'enchaînement de multiples commandes.

Soit à réaliser

```
% commande1 | commande2 | ... | commanden
```

Il s'agit d'itérer `n-1` fois le traitement suivant :

- créer un tube qui va relier la commande `i` à la commande `i+1`
- créer un fils qui
  - duplique son descripteur d'accès en écriture sur le tube vers sa sortie standard,
  - exécute la commande `commandi` et termine ;
- le père redirige son descripteur en lecture sur le tube vers son entrée standard et continue.

puis de créer un dernier fils pour exécuter la commande `commandn`. □

Fin du sujet