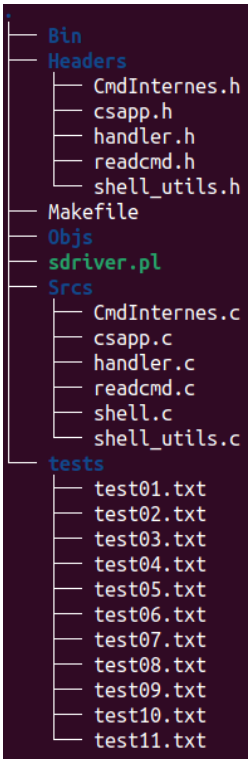
David Marchionini
Tom Clément

Systèmes et Réseaux informatiques
Rapport - Mini Shell

L'objectif de ce projet est de recréer une version simplifiée du shell Unix, un interpréteur de commandes. Le mini-shell est lancé et est contenu dans le terminal et doit pouvoir utiliser des appels système afin d'exécuter les commandes tapées en ligne de commande. Il doit pouvoir interpréter des commandes simples avec ou sans redirection d'entrées / sorties, et nous permettre d'exécuter des commandes en arrière-plan ainsi que des séquences de commandes. Afin d'avoir un programme stable, nous implémentons aussi une gestion d'erreurs en cas de commandes invalides et une gestion de processus zombis afin de protéger la mémoire et de conserver une certaine vitesse d'exécution.

Organisation du code du projet :



Nous avons organisé notre code de telle manière à séparer les headers (dans le répertoire Headers), les fichiers sources (répertoire Srcs), la création des fichiers objets (répertoire Objs) et du fichier exécutable (répertoire Bin) ainsi que les fichiers de tests (répertoire tests).
Dans chaque fichier de test, il y a une mini-description ainsi que les commandes qu'il exécute.

Dans le répertoire racine, il n'y a donc que le fichier Makefile et le script perl pour tester.

Pour compiler, lancer simplement la commande make puis ./Bin/shell, si vous voulez afficher chaque commande que vous donnerez à notre shell, alors lancer la commande make DEBUG=1 puis ./Bin/shell.

Pour tester avec nos fichiers tests, il suffit de lancer la commande :
./sdriver.pl -t tests/<file.txt> -s ./Bin/shell

1. Compréhension de la structure cmdline et des résultats de readcmd.

2. Commande pour terminer le shell

Ces deux commandes (quit et exit), sont implémentées en tant que commandes internes au shell (pas besoin de créer un processus fils pour exécuter ces commandes).

```
if(!strcmp(cmd[0], "quit") || !strcmp(cmd[0], "exit")){
      printf("%s\n", cmd[0]);
      exit(0);
}
```

3. Interprétation de commande simple

```
if (execvp(l->seq[0][0], l->seq[0]) < 0) {
                        perror("execpv ");
                        exit(2);
                  }
```

3Bis. Commandes internes

Nous avons créé un module CmdInternes permettant de gérer les commandes internes à notre shell (seulement 5 pour l'instant, pwd, cd, echo, quit et exit).

4. Commande simple avec redirection d'entrée ou de sortie

```
if (l->in){
                        fdIn = Open(l->in, O_RDONLY, 0);
                        if (fdIn==-1){perror("Open ");exit(2);}
                        Close(0);
                        if(dup(fdIn) < 0){perror("dup ");}
                        Close(fdIn);
                  }
                  if (l->out){
                        fdOut = Open(l->out, O_CREAT | O_WRONLY,
0644);
                        if (fdOut==-1){perror("Open ");exit(2);}
                        Close(1);
                        if(dup(fdOut) < 0){perror("dup ");}
                        Close(fdOut);
                  }
```

5. Gestion des erreurs

```
	/* If input stream closed, normal termination */
	if (!l) {
		printf("exit\n");
		exit(0);
	}

	if (l->err) {/* Syntax error, read another command */
		printf("error: %s\n", l->err);
		continue;
	}

	/* Si on clique juste sur entrée sans rien d'autres par
exemple */
	if(l->seq[0]==NULL){continue;}
```

Ainsi que les vérifications systématiques après un appel à une fonction système (utilisation de perror).

6. Séquence de commandes composée de deux commandes reliées par un tube
ET
7. Séquence de commandes composée de plusieurs commandes et de plusieurs tubes

```
	while(i<n_commandes){

	pidc= Fork();
	if(pidc==-1){perror("Fork ");exit(2);}

	if(pidc==0){ //child
	if(i!=n_commandes-1){ //all but last cmd
			Dup2(fds[i][1],1);

		}
		if(i!=0){//all but 1st cmd
			Dup2(fds[i-1][0],0);

		}
	if(execvp(l->seq[i][0], l->seq[i]) < 0){perror("execpv
");exit(2);}
	}
	close(fds[i][1]);

	if(i!=0){close(fds[i-1][0]);}
	while(waitpid(0,0,0) < 0);

	i++;
	}
```

8. Exécution de commandes en arrière-plan

Modification de la structure cmdline :

```
struct cmdline {
  char *err;
  char *in;
  char *out;
  char ***seq;
  int bg; /* If not null : execution in background. */
};
```

Et modification de la fonction readcmd(), initialisation de l->bg :

```
s->bg=0;
```

Ajout du case '&' :

```
case '&':
   /* Pour commande en background */
   if (cmd_len == 0) {
       s->err = "misplaced background";
       goto error;
   }

   s->bg=1;
   break;
```

9. Gestion des zombis

Création d'un module handler :

```
void HandlerChild(int sig){
   int status;
   waitpid(-1, &status, WNOHANG|WUNTRACED);
}
```

Et l'association du handler au signal SIGCHLD :

```
signal(SIGCHLD, &HandlerChild);
```

Pour conclure, nous n’avons pas eu le temps d’aller plus loin. Cependant, nous avons testé nos différentes parties de codes et n’avons pas remarqué d’erreurs précises.

Le projet était très intéressant car il nous a permis de comprendre toute la complexité d’un interpréteur de commande et de mettre en application ce qu’on a pu voir lors des différents cours/TD/TP de Systèmes Réseaux et même d’autre matières comme l’option MOCA, qui nous a permis d’organiser correctement notre code dans des fichiers et d’adapter en conséquence le Makefile.