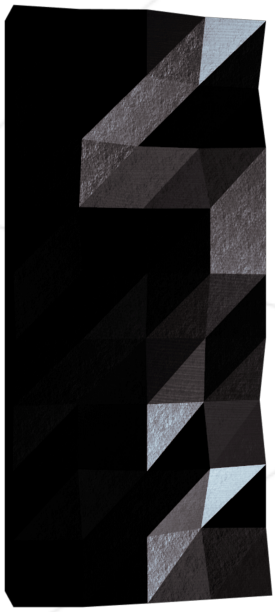

Projet Environnement UNIX I

42sh

Résumé: Le moment est enfin arrivé. Place au célèbre 42sh !

Table des matières

Chapitre I

Préambule

I.1 How to Get into Orbit

Getting into orbit over Kerbin is rather simple, but it will require some knowledge and preparation. Space begins at 70,000m above the planet Kerbin. Stay above that for an entire flight around the planet and you're in orbit. The process to get into orbit follows a simple progression :

- Launch straight up to 10km
- Change your Pitch to 45° east and keep the engines on until your projected Apoapsis is above 70km
- Change your pitch to horizontal just before you reach the Apoapsis
- Burn at Apoapsis until your Periapsis is above 70km

Specifications :

- Length : 15–20 minutes
- Difficulty : Harder than a suborbital flight, easier than an orbital intercept.
- Skills needed : Seat of the pants
- For version : Every version (tested on 0.23)

I.2 Steps

I.2.1 Step 0 - Rocket Design

A liquid fueled rocket with at least two stages preferably. Anything less will either only get you suborbital or an unwieldy expensive super large fuel tank thats only good for being an orbiting billboard. The cheapest orbiter you can build with the current stock game (0.22) is :

- Unmanned : Probodobodyne OKTO2 or Manned : Command Pod Mk1 with Mk16 Parachute and TR-18A Stack Decoupler
- FL-T400 Fuel Tank
- LV-909 Liquid Fuel Engine
- TR-18A Stack Decoupler
- FL-T800 Fuel Tank
- LV-T30 Liquid Fuel Engine

For fun, use the LVT-30 engine for your first stage as it doesn't have thrust vectoring and thus will challenge you to actually fly the craft into orbit using your WASD keys keeping your navball on target. Your control module by default provides enough SAS control by itself to prevent you from going out of control for this mission. Make sure your staging sequence is the way you want it, else you can add your flight to the countless list of catastrophic mission failures.

I.2.2 Step 1 - Launch Prep

Prepare your launch.

- Set your map view with the m key to see your rocket at the launch site from space, tilted so you are looking north. You want to be able to see your apoapsis marker during your gravity turn so you can gauge when to cut your engines and coast up to it prior to burning your orbiting maneuver.
- Set your thrust to maximum by holding Shift.
- Toggle on SAS by hitting t.

I.2.3 Step 2 - The Launch

Say your countdown if you wish, and hit spacebar to launch into. . .well, space. . .without the bar (you'll make a space station with a bar at this end of the galaxy later).

I.2.4 Step 3 - Get up to 10,000 meters

Keep your rocket pointed straight up (use your navball to keep your dot on the top dot on the blue part of the navball) until you hit around 9,900 meters. Your first engine should cut out before this, just jettison it with the spacebar and burn your final engine.

Throttle your second engine down to 2/3rds power since the atmosphere is weaker here and won't slow you down as much, and you will save precious fuel.

I.2.5 Step 4 - Gravity Turn 45 degrees East until 70km Apoapsis

Now for the fun part. Assuming you are pointing straight up, hit the d key to turn your ship using your navball until your dot is able to slide left and right on the 90 degree east line. Then tip on that line down toward the ground until you hit the 45 degree east mark. Hit your t key for SAS to help you keep it there, but don't rely on it. You will need to make course corrections until you get past 70km and enter space. At this point you should be going into space at a 45 degree angle from the ground, and in an easterly direction. Doing so will gain you speed and not waste the fuel you will need to get into orbit.

I.2.6 Step 5 - Get Your Apoapsis above 70km

While climbing up to 70km or 70,000 meters, it's now time to switch to your map view and control your ship from there entirely. Hopefully your navball is toggled on, else click on the collapsed tab at the bottom of the map view. You will need to keep burning your fuel until you see your apoapsis marker reach 70km. You can see how high it is by hovering your mouse over it. Once it hits 70km (I usually shoot for 75km to buy me some head room) your craft will be able to coast up to it without any further fuel. Feel free to cut your engines with the x key and save some fuel for your orbital burn.

I.2.7 Step 6 - Orient for On-Orbit Burn

As you approach apoapsis (preferably before 30 seconds prior) orient your ship to the 0 degree latitudinal mark, heading east. Again, you should be in your map view when you start your burn. Hit tab to center your view on Kerbin and zoom out so you can see your orbit as it forms.

I.2.8 Step 7 - Burn into Orbit

Once you are 10–30 seconds away from your apoapsis, begin your on-orbit burn using the Shift key to throttle up. Full throttle is best at lower altitude apoapsises since you don't want to burn past apoapsis lest you waste precious return fuel if that's your intent. If you aren't concerned, go full throttle baby at any point near apoapsis and claim a stake with the stars. You only need to burn your fuel long enough until you see a periapsis marker appear on the other side of the orbit from the apoapsis, and you see a full orbit circle, then cut your engine with the x key. Congrats, you made it into orbit. It's usually a good idea to keep burning your fuel until your new periapsis marker is above 70km. If it's below, then your orbit will cause your craft to aerobrake, eventually returning to Kerbin. If your periapsis is ever above 70km, congrats, you will orbit Kerbin forever. If you have a manned flight, and fuel is low, then only burn until your periapsis is below 70km to ensure a safe return to Kerbin.

I.3 Finishing word

After orbiting for a while, depending on your fuel (with this design you won't have much if any at all) you can orient for a deorbit burn by burning backwards in the direction of your travel, once you are at your apoapsis point for maximum efficiency lest you be stuck in space with a manned crew forever.

Chapitre II

Introduction

Dans toute scolarité, il existe des moments qui marquent les étudiants pour toujours. Le `42sh` est l'un de ces moments. Réaliser ce projet, c'est marquer une étape importante à `42`.

Il s'agit ici d'écrire un shell `UNIX` le plus stable et le plus complet possible. Vous connaissez déja de nombreux shells et chacun possède ses propres caractéristiques, du humble `sh` présent sur toutes les distributions `UNIX` du monde, au très complet et très complexe `zsh` que vous êtes si nombreux à utiliser sans savoir pourquoi. Bien sur, il existe de nombreux autres shells, comme `bash`, `mksh`, `tcsh`, `dash`, `fish`, `xonsh`, etc. Etant donné que `42sh` sera votre premier vrai shell, il est fréquent parmis les étudiants de vouloir choisir un shell de référence pour essayer d'en reproduire le comportement. C'est une bonne idée, mais à condition de choisir votre shell de référence en connaissance de cause. En effet, ceux d'entre vous qui choisiront `zsh` comme shell de référence s'engageront dans une quête longue et difficile, bien que très riche en enseignements. Par exemple, ils pourront y apprendre l'humilité et la notion de quantité de travail colossale.

La meilleure façon d'aborder la question du shell de référence consiste tout simplement à en essayer plusieurs et à se faire une idée de leurs différences, souvent subtiles, parfois tordues. Toutefois ne perdez pas de vue que si `sh` est souvent considéré comme le shell le plus "simple", le recoder complètement de manière stable est une réussite bien plus intéressante qu'un shell qui promet monts et merveilles mais qui se révèle incapable de faire plus que quelques pipes et redirections.

Shell de référence: le cas bash

La version de bash sur OSX est ancienne (dernier patch 2014-11-07). Je vous recommande d'installer la dernière version disponible via brew (4.4 2016-09-15)

Le maître mot içi est "stabilité". Un `42sh` humble mais indestructible vaudra toujours plus qu'un `42sh` proposant toutes les options imaginables mais qui plante dans un cas imprévu, puisque ce dernier vaudra 0. Assurez-vous donc de rendre un `42sh` stable, je n'insisterai jamais assez sur ce point.

Le projet est composé de 2 parties décrites ci-dessous et est légèrement différent des autres sujets auxquels vous êtes habitués.

- Une partie **obligatoire**, à réaliser impérativement. Ce sont les pré-requis minimum d'un shell.
- Une partie **modulaire**, qui ne sera prise en considération que si la partie obligatoire fonctionne dans son intégralité.

La partie obligatoire, seule, ne permet pas de valider le projet. Il vous faudra choisir et implémenter plusieurs points de la partie modulaire pour le valider. Ceci pour deux raisons : la première est que la partie obligatoire représente un shell extrêmement basique ; la seconde, c'est pour vous obliger à selectionner des fonctionnalités de la partie modulaire et à vous fixer vos propres objectifs dans la réalisation de ce projet.

Un dernier point, l'utilisabilité de votre `42sh` sera largement prise en compte. Il serait judicieux de vous assurer qu'un utilisateur d'un shell ordinaire soit capable d'utiliser votre `42sh` de manière intuitive ...

Chapitre III

Consignes

- Ce projet ne sera corrigé que par des humains. Vous êtes donc libres d'organiser et nommer vos fichiers comme vous le désirez, en respectant néanmoins les contraintes listées ici.
- Votre exécutable doit s'appeler `42sh`
- Vous devez rendre un Makefile avec toutes les règles usuelles.
- Si vous souhaitez utiliser votre `libft`, vous devez en rendre les sources dans un dossier nommé `libft` à la racine de votre dépot, avec un Makefile pour la compiler. Aucune version déja compilée de votre biliothèque ne sera accepté en soutenance. Le `Makefile` de votre `42sh` doit bien entendu compiler et linker avec votre `libft`.
- Vous devez gérer les erreurs de façon pertinente. En aucun cas votre programme ne doit quitter de façon inattendue (Segmentation fault, etc ...).
- Votre terminal ne doit jamais afficher n'importe quoi, gérez intelligemment sa configuration.
- Vous avez le droit d'utiliser les fonctions suivantes :
  - Toute la section 2 des mans
  - Les fonctions autorisées du 21sh
  - `getpwnam(3)`, `tcgetpgrp(3)` et `tcsetpgrp(3)`
  - Toutes les fonctions de la bibliothèque `termcaps`.
- Vous pouvez poser vos questions sur le forum, sur slack, ...
- Bon courage à tous !

Chapitre IV

Partie obligatoire

- Les pré-requis du **minishell**.
  - Affichage d'un prompt.
  - Exécution de commandes avec leurs paramètres et gestion du `PATH`.
  - Gestion des erreurs, sans utiliser `errno`, et de la valeur de retour des commandes
  - Gestion correcte des espaces et des tabulations.
- Les pré-requis du **21sh**.
  - Édition complète de la ligne de commande
  - Les opérateurs de redirection et d'aggrégation :
    - `>`
    - `>>`
    - `<`
    - `<<`
    - `>&`
    - `<&`
  - Les pipes `|`
  - Les séparateurs `;`
- Les built-ins suivants :
  - `cd`
  - `echo`
  - `exit`
  - `type`
- Les opérateurs logiques `&&` et `||`

Précédence des opérateurs

Les opérateurs, qu'ils soient de contrôle ; && || ou de redirections < > |, ont une précédence. Faites-y attention !
Par exemple, ces 2 commandes ne produisent pas du tout le même résultat:

- `ls doesnotexist . 2>&1 >/dev/null`
- `ls doesnotexist . >/dev/null 2>&1`

Pensez à bien regarder la **grammaire** de votre shell.

- La gestion des variables internes au shell (Ne vous occupez pas des variables en lecture-seule).
  - La création de variable interne selon la syntaxe : `name=value`.
  - L'exportation des variables internes vers l'environnement, via le built-in `export`.
  - La possibilité de lister les variables internes du shell via le built-in `set` (pas d'option requises).
  - Le retrait des variables internes et d'environnement, via le built-in `unset` (pas d'option requises).
  - La création de variable d'environnement pour une commande unique, exemple : `HOME=/tmp cd`.
  - L'expansion simple des paramètres selon la syntaxe `${}` (pas de format supplémentaire requis).
  - La gestion des parametres speciaux, comme `?` pour le code de sortie de la commande precedente.
- Gestion du job control, avec les built-ins `jobs`, `fg`, `bg` et l'opérateur `&`
- Une gestion correcte de tous les signaux.
- Chaque built-in doit avoir au minimum les options dictées par le standard POSIX, sauf cas explicite comme `set` ou `unset`.

Explication pour env, setenv, unsetenv

env est un binaire et non un built-in dans les shells. Il vous a été demandé de l'implémenter pour que vous compreniez son comportement. Vous n'êtes pas obligé de le fournir cette fois-ci.

Les built-ins setenv et unsetenv sont exclusifs aux shells de la famille **Csh**. Le comportement de certains shell ne devrait pas être copié: Unix FAQ: Csh Considered Harmful

Chapitre V

Partie modulaire

Les fonctionnalités demandées dans la partie obligatoire représentent le strict minimum de ce qu'on attend d'un shell. Il va maintenant vous falloir choisir et implémenter des features un peu plus avancées. Un minimum de **6 features** de la liste ci-dessous est requis pour valider le projet.

Gardez toutefois à l'esprit que la stabilité sera beaucoup plus importante que la quantité. N'incluez pas une option qui pose un problème au reste du programme par exemple. Et n'oubliez pas que cette partie ne sera évaluée que si la partie obligatoire est complète et indestructible.

Les features modulaires :

- Les inhibiteurs `"` (double quote), `'` (simple quote) et `\` (backslash)
- Le pattern matching (globing) : `*`, `?`, `[]`, `!` et les intervals de caractères avec `-`
- L'expansion du tilde et les formats supplémentaires des paramètres :
  - `~`
  - `${parameter:-word}`
  - `${parameter:=word}`
  - `${parameter:?word}`
  - `${parameter:+word}`
  - `${#parameter}`
  - `${parameter%}`
  - `${parameter%%}`
  - `${parameter#}`
  - `${parameter##}`
- Les commandes groupées et sous-shell : `()`, `{};`
- La substitution de commande : `$()`
- L'expansion arithmétique : `$(())`
  Seulement ces opérateurs :
  - Incrémentation, Décrémentation `++ --`
  - Addition, Soustraction `+ -`
  - Multiplication, Division, Modulo `* / %`

  - Comparaison `<= >= < >`
  - Égalité, Différence `== !=`
  - ET/OU logique `&& ||`

  Voir la page Arithmetic Precision and Operations pour plus d'explication sur le fonctionnement des opérateurs.
- La substitution de processus : `<()`, `>()`
- La gestion complète de l'historique :
  - Les expansions :
    - `!!`
    - `!word`
    - `!number`
    - `!-number`
  - La sauvegarde dans un fichier pour être utilisé sur plusieurs sessions
  - Le built-in `fc` (toutes les options POSIX)
  - Recherche incrémentale dans l'historique avec `CTRL-R`
- Complétion dynamique contextuelle des commandes, des built-ins, des fichiers, des variables internes et d'environnement. Qu'entend-on par contextuelle ? Prenons la commande "`ls  /`" et que votre curseur est sur le `/`, alors une complétion contextuelle ne proposera que le contenu du dossier racine et non les commandes ou built-ins. Idem pour ce cas-ci : "`echo ${S`", la complétion ne devra proposer que des noms de variables qui commencent par `S`, qu'elles soient internes ou d'environnements.
- Les deux mode d'édition de la ligne de commande Vi et Readline. La possibilitée de choisir entre l'un ou l'autre mode se fera par l'intermédiaire de l'option `-o` du built-in `set`.

  Voici la liste des raccourcis que vous devez implémenter :
  - **Vi** : `#`, `  `, `v`, `j`, `k`, `l`, `h`, `w`, `W`, `e`, `E`, `b`, `B`, `^`, `$`,`0`, `|`, `f`, `F`, `t`, `T`, `;`, `,`, `a`, `A`, `i`, `I`, `r`, `R`, `c`, `C`, `S`, `x`, `X`, `d`, `D`, `y`, `Y`, `p`, `P`, `u`, `U`.
  - **Readline** : `C-b`, `C-f`, `A-f`, `A-b`, `C-a`, `C-e`, `C-x-x`, `Backspace`, `C-d`, `C-u`, `C-k`, `A-d`, `C-w`, `C-y`, `C-p`, `C-n`, `C-_`, `C-t`, `A-t`.

  L'explication des raccourcis de Vi peut être trouvé dans l'implémentation de sh et plus particulièrement la section `EXTENDED DESCRIPTION`. Pour Readline, regardez cette page.
- La gestion des **alias** via les built-ins `alias` et `unalias`
- Une table de hachage ainsi que le built-in `hash` pour intéragir avec elle.
- Le built-in `test` avec les opérateurs suivants : `-b`, `-c`, `-d`, `-e`, `-f`, `-g`, `-L`, `-p`, `-r`, `-S`, `-s`, `-u`, `-w`, `-x`, `-z`, `=`, `!=`, `-eq`, `-ne`, `-ge`, `-lt`, `-le`, `!`. Ainsi que la possibilité d'un simple opérande, sans opérateur.

En cas de doute sur le comportement d'une fonctionnalité, référez-vous au standard POSIX. Il est tout à fait acceptable de choisir d'implémenter différemment une fonctionnalité, si ça vous semble cohérent pour votre shell, mais il n'est pas acceptable de faire moins que les specs POSIX par "flemme".

Chapitre VI

Partie Bonus

VI.1 Liste des bonus possibles

- Le Shell Script (`while`, `for`, `if`, `case`, `function`, etc.)
- Autocomplétion pour les paramètres des commandes/built-ins
- Shell conforme à la norme POSIX

VI.2 Pré-requis pour la prise en compte des bonus

Une nouveauté est au menu concernant les bonus du 42sh. Ceux-ci ne seront pris en compte qu'à condition que votre code soit **clair** et **propre**.

Qu'est ce qu'on entend par là ?

- Pas de ternaire toutes les 3 lignes
- Des noms de fonctions explicites (pas de `ft_parse1`, `ft_parse2`, etc ...)
- Cela s'applique également aux noms de variables
- Utiliser judicieusement le **qualifier const**
- Avoir un historique **git** et des messages de commit explicites
- Avoir des tests automatisés

En somme, respecter les parties conseillées de la **Norme**

VI.3 Un peu d'histoire

Une petite pause père Castor?

Chapitre VII

Rendu et peer-évaluation

Rendez votre travail sur votre dépôt `GiT` comme d'habitude. Seul le travail présent sur votre dépôt sera évalué.

Bon courage á tous et n'oubliez pas votre fichier auteur !