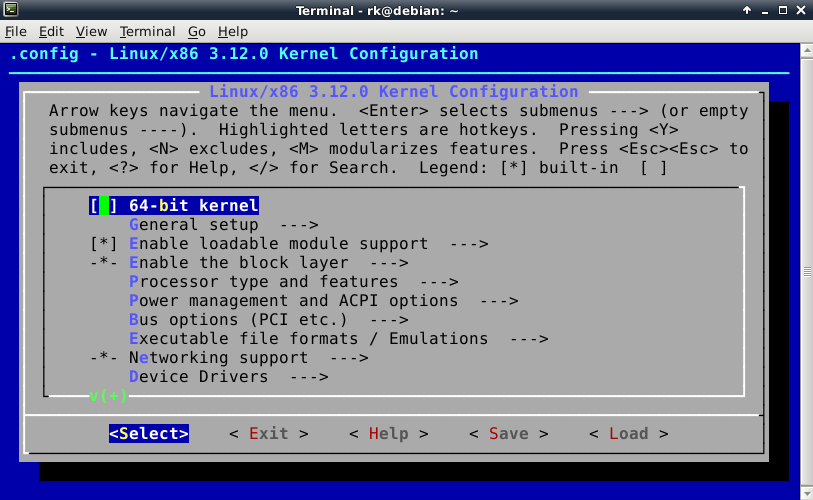
Secure Embedded System (SeS)

U-boot Valgrind Compile kernel



Auteur : Spinelli Isaia

Prof : [Schuler Jean-Roland](https://mse.hes-so.ch/consultation/master/horairesMSE.php?annee=2020&id=81&type=1)

Date : 14.09.2020

Salle : A4 – Lausanne

Classe : SeS

Table des matières

[Introduction - 2 -](#_Toc51949953)

[Références - 2 -](#_Toc51949954)

[Valgrind - 3 -](#_Toc51949955)

[Introduction - 3 -](#_Toc51949956)

[Question 1 - 3 -](#_Toc51949957)

[Limitations - 8 -](#_Toc51949958)

[Question 2 - 8 -](#_Toc51949959)

[Présentation du programme fourni - 9 -](#_Toc51949960)

[Correction du programme - 9 -](#_Toc51949961)

[Conclusion - 16 -](#_Toc51949962)

[U-boot - 17 -](#_Toc51949963)

[Compile Kernel - 18 -](#_Toc51949964)

[Conclusion - 19 -](#_Toc51949965)

[Difficultés rencontrées - 19 -](#_Toc51949966)

[Compétences acquises - 19 -](#_Toc51949967)

[Résultats obtenus - 19 -](#_Toc51949968)

# Introduction

Ce rapport est composé de trois laboratoires distincts :

1. Valgrind
2. U-boot
3. Compile Kernel

# Références

Valgrind :

<http://pub.phyks.me/sdz/sdz/debuguer-facilement-avec-valgrind.html>

<https://en.wikipedia.org/wiki/Valgrind>

<https://www.valgrind.org/>

U-boot :

Compile Kernel :

# Valgrind

Valgrind est un outil de [programmation](https://fr.wikipedia.org/wiki/Programmation_informatique) [libre](https://fr.wikipedia.org/wiki/Logiciel_libre). Il permet de détecter automatiquement de nombreux bugs de gestion de la mémoire, de threading et de profiler des programmes en détail.  De nombreux modules tiers ont été écrits pour répondre à certaines demandes. Voici quelques exemples :

* *Massif :* Profileur du tas et de la pile. L'interface graphique massif-visualizer permet d'afficher les mesures issues de Massif.
* *Cachegrind :* Profileur de la mémoire cache et de prédiction de branchement. L'outil graphique KCacheGrind Permet de visualiser les mesures issues de Cachegrind.
* *Callgrind :*  Analyseur de graphe d'appel de fonctions.  KCacheGrind permet aussi d'exploiter les mesures de cet outil.
* *Memcheck :* Détecteur d’erreur mémoire.

## Introduction

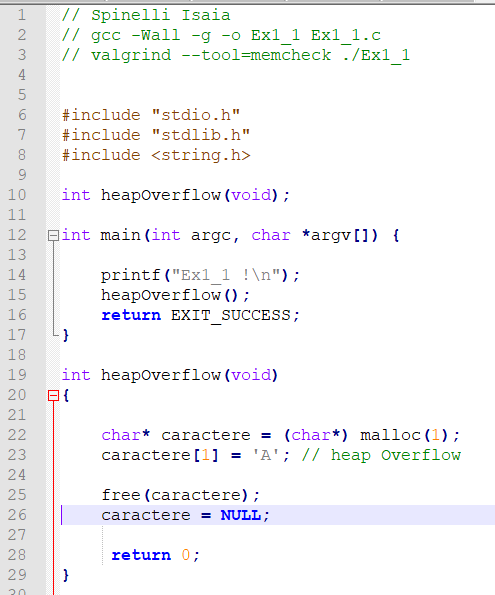
Durant ce chapitre l’outil Valgrind est utilisé afin de rechercher des erreurs et corriger des programmes. Le but est de se familiariser avec l’outil et le maîtriser.

## Question 1

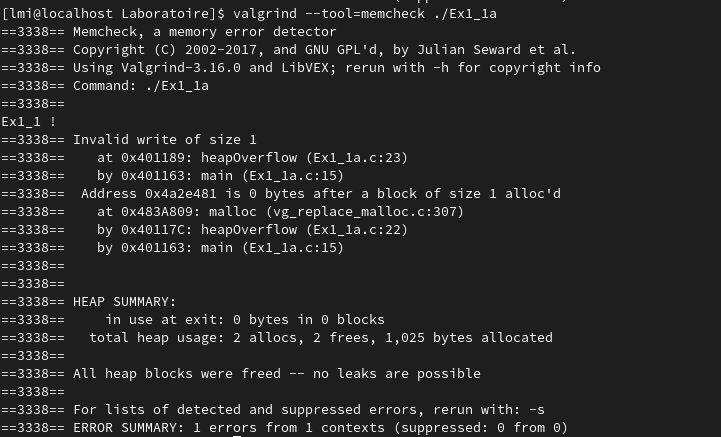
Le premier exercice consiste à créer trois petits programmes en C contenant des erreurs afin d’utiliser l’outil Valgrind pour trouver ces erreurs.

### Programme 1

Le 1er programme comporte une erreur de type « heap overflow » :



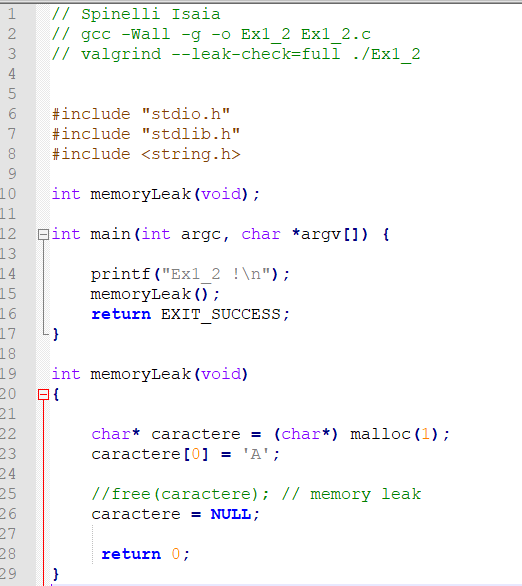
Ce programme très simple contient une erreur de type heap overflow à la ligne 23. Afin de trouver cette erreur, Valgrind a été utilisé avec l’outil « memcheck » :



La sortie indique correctement qu’il y a une erreur. Une écriture invalide (overflow) de la taille d’un byte a été repéré à la ligne 23.

### Programme 2

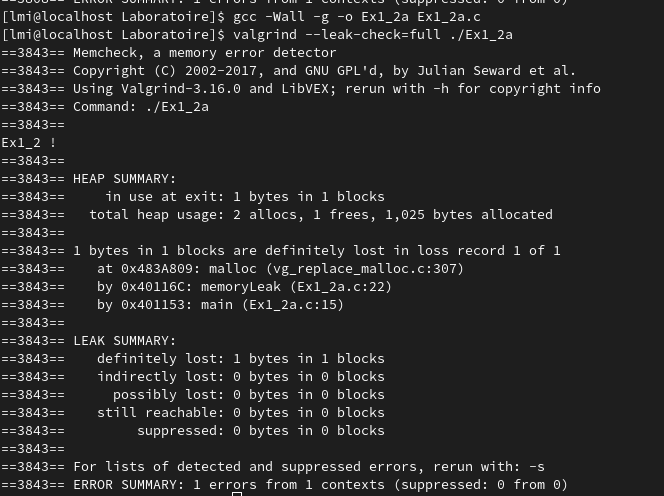
Le 2eme programme comporte une erreur de type « memory leak » :



Ce programme est très ressemblant au premier, deux différence ont été apportées :

1. L’erreur de type heap overflow a été corrigée à la ligne 23.
2. Une erreur de type memory leak a été introduite en commentant la ligne 25.

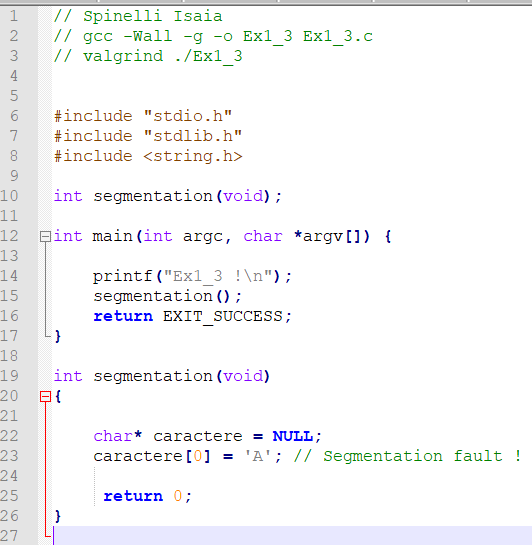
Afin de permettre à Valgrind de détecter ce type d’erreur, l’option « --leak-check=full » de memcheck a été ajoutée.



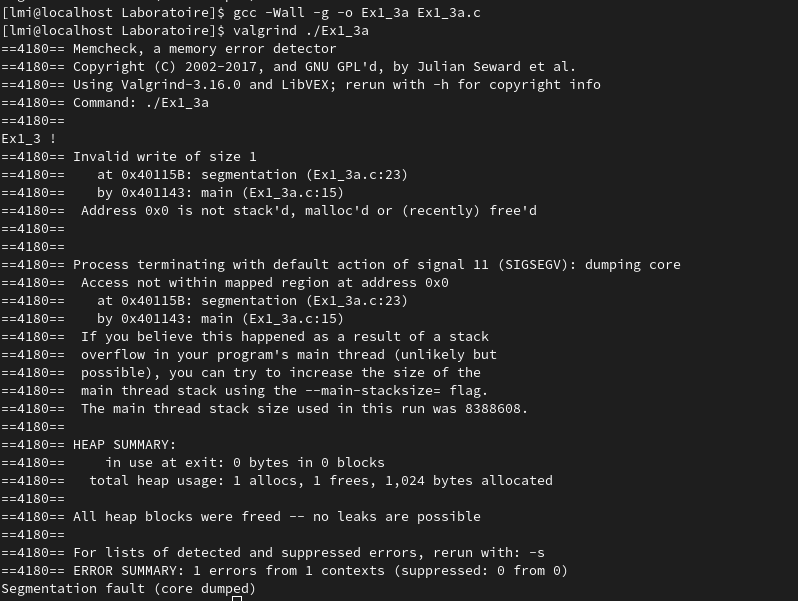
Encore une fois Valgrind a pu détecter l’erreur et indique qu’un byte a été perdu. De plus, il affiche la ligne lors de l’allocation du byte en question (ligne 22).

### Programme 3

Le 3eme programme comporte une erreur de type « Segmentation » :



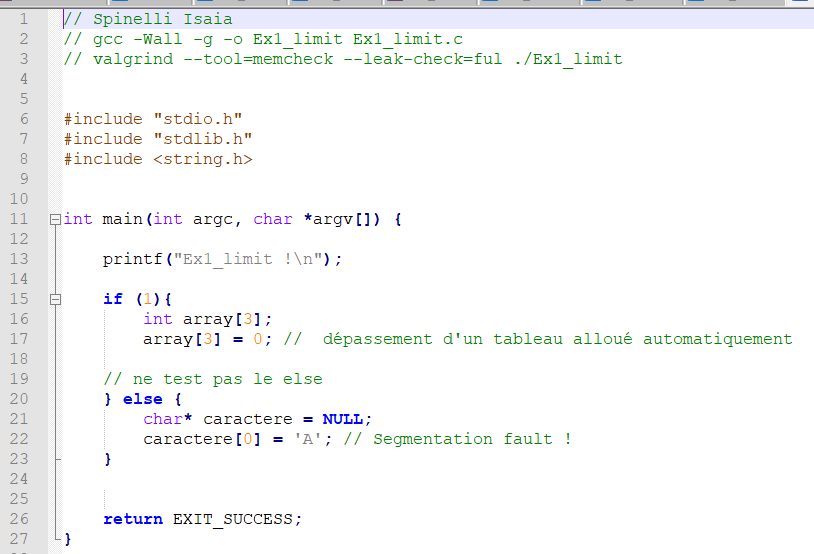
Ce programme contient une erreur de segmentation à la ligne 23 en écrivant dans une adresse NULL. On peut voir la sortie du Valgrind ci-dessous :



Sans problème Valgrind a pu détecter l’erreur de segmentation tout en indiquant la ligne fautive.

## Limitations

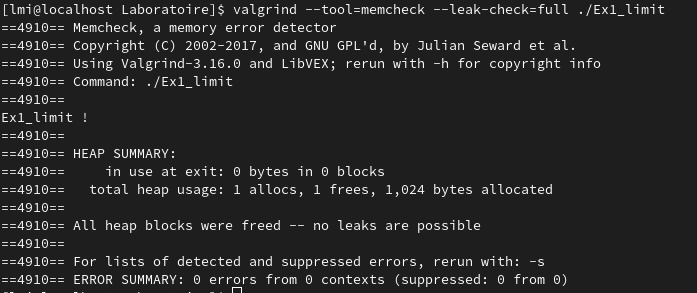
Un quatrième programme a été réalisé afin de tester quelques limite de Valgrind :



Ce dernier programme contient deux erreurs :

1. À la ligne 17, une erreur de dépassement d’un tableau alloué automatiquement.
2. À la ligne 22, la même erreur du programme précédent dans une condition toujours fausse.

Ci-dessous, la sortie indiquée par Valgrind :



Cette fois, l’outil ne détecte aucune erreur. On peut donc en conclure que Valgrind est un outil très puissant mais ne permet pas de détecter un dépassement sur un tableau alloué automatique. De plus, il ne vérifie pas les lignes de code dans des conditions toujours fausses.

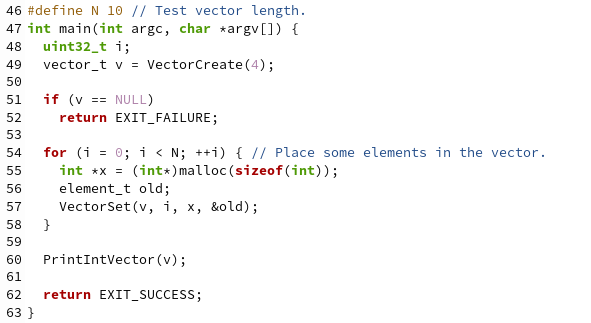
## Question 2

Pour cette partie, un programme nous a été fourni. Celui-ci est spécialement mauvais est contient donc plusieurs erreurs. Le but est de contrôler et corriger ce programme en utilisant Valgrind.

### Présentation du programme fourni

Ce programme implémente en C une structure nommée « vector\_t » contenant l’adresse d’un tableau et une taille. Plusieurs fonctions permettant la manipulation de ce type de structure ont été implémentées.

Ci-dessous, le programme « main » qui crée un vecteur de 4 éléments, le remplit de N éléments, puis, l’affiche.

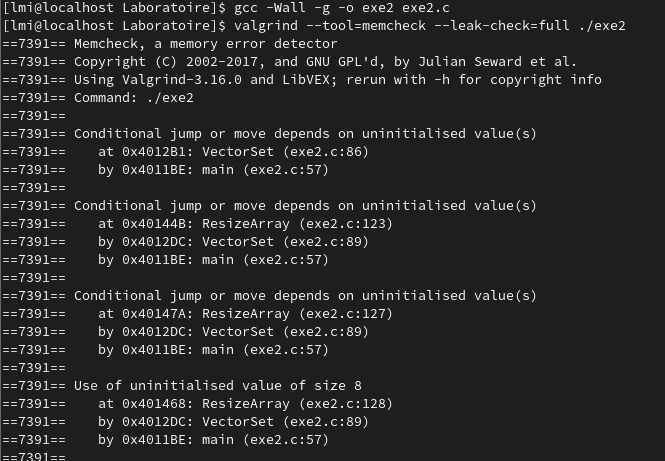


### Correction du programme

Le système de correction du programme est basé sur une boucle répétitive constituée de quatre étapes :

1. Compilation du programme
2. Exécution de Valgrind sur le fichier exécutable
3. Recherche de l’erreur
4. Correction de l’erreur

Voici pour commencer la compilation et l’exécution de Valgrind sur la base du programme fourni :



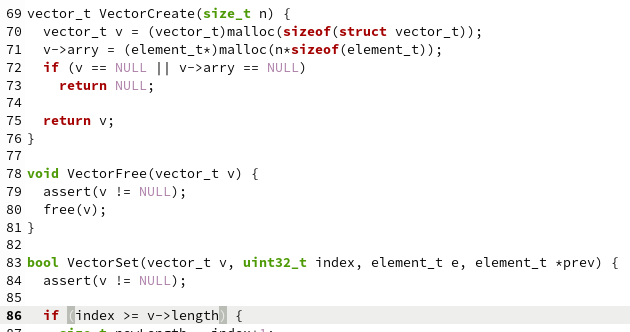
En haut de l’image on peut voir les commandes permettant la compilation ainsi que l’exécution de l’outil Valgrind.

Ensuite, une grande quantité d’erreurs toujours basées sur une valeur non initialisée. La première est détectée à la ligne 89.

Au bas du terminal, la quantité d’erreurs totale est indiquée (58) :



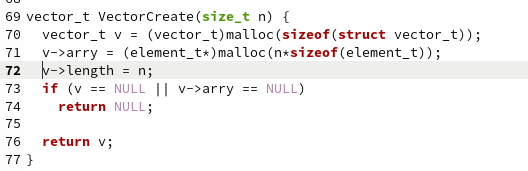
Afin de rechercher et comprendre l’erreur, voici quelques lignes du code au-dessus de la ligne 89 comme indiqué par Valgrind :



On peut voir à la ligne 86 une comparaison entre un paramètre de la fonction « VectorSet » (index) et un attribut du vecteur v (length). On peut donc conclure que l’attribut « length » du vecteur n’est sûrement pas initialisé.

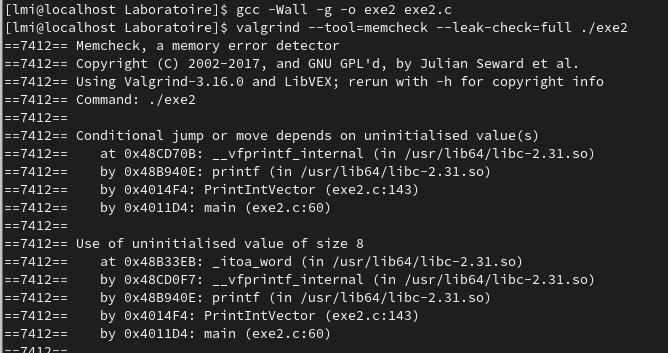
La fonction permettant de créer un vecteur « VectorCreate » est à la ligne 69. On peut confirmer le fait que la longueur n’est pas initialisée.

Voici la modification apportée :



*Remarque : Pour faire correctement les choses, cette correction ne suffit pas. En effet, les n éléments insérés lors de la création du tableau doivent être initialisés. Dans notre cas, cela n’est pas impératif car juste après la création du vecteur, on remplit le vecteur avec des nouveaux éléments.*

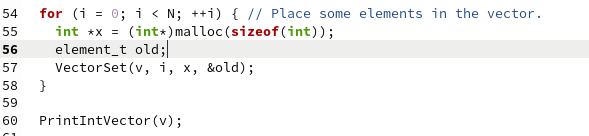
Après une nouvelle compilation et exécution de l’outil Valgrind, on peut voir que les erreurs détectées sont différentes. De plus, le nombre total d’erreur est passé de 58 à 53.

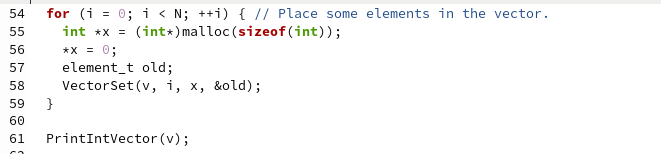




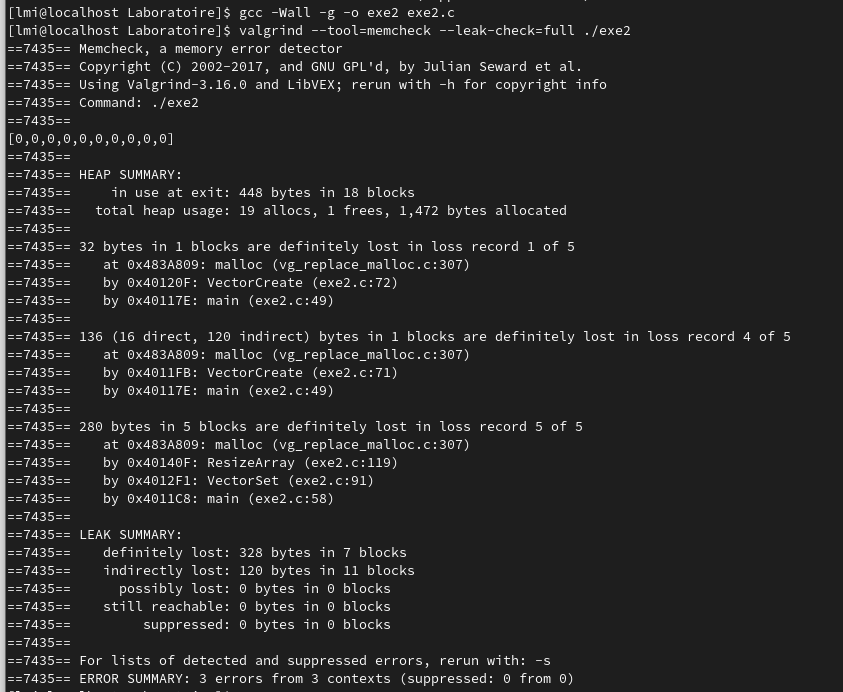
La cause des erreurs est une fois de plus des valeurs non-initialisées. Cependant, les erreurs proviennent de la fonction « PrintIntVector ».

En recherchant un peu, on peut s’apercevoir que la boucle responsable de remplir le vecteur contient une erreur. En effet, la variable x insérée dans le vecteur est correctement allouée mais pas initialisée.



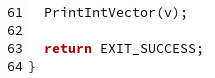
Voici donc la correction apportée : 

Après une nouvelle compilation et exécution de l’outil Valgrind, on peut voir que les erreurs détectées sont différentes. De plus, le nombre total d’erreur est passé de 53 à 3.

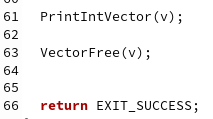


Cette fois, la cause des erreurs est une fuite de mémoire. On peut voir qu’énormément de bytes sont perdus, principalement dû à la fonction « VectorCreate ».

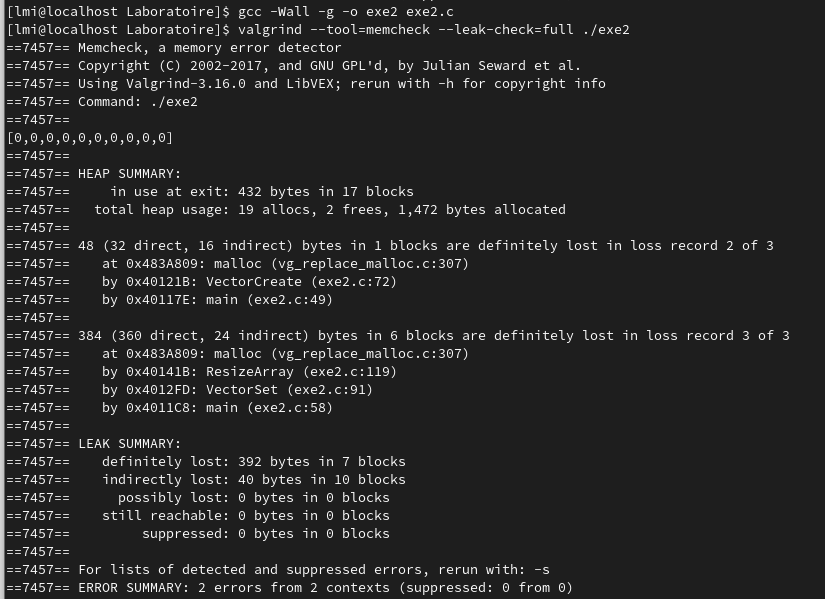
On peut voir que la fonction « VectorFree » permettant de libérer toute la mémoire de vecteur n’est jamais appelée :



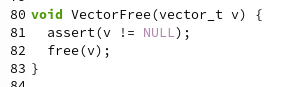
Voici l’ajout de l’appel à la fonction :



Après une nouvelle compilation et exécution de l’outil Valgrind, on peut voir que les erreurs détectées sont ressemblantes mais que la quantité de byte perdu indirectement a diminuée alors que la quantité de byte perdu définitivement a augmentée. Cependant, le nombre total d’erreur est passé de 3 à 2.

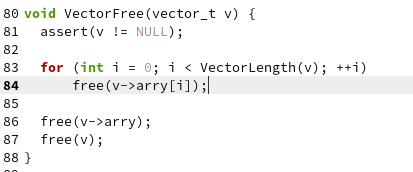


On peut vite en déduire que la fonction « VectorFree » n’a pas été implémenté par « Linus Torvalds » :

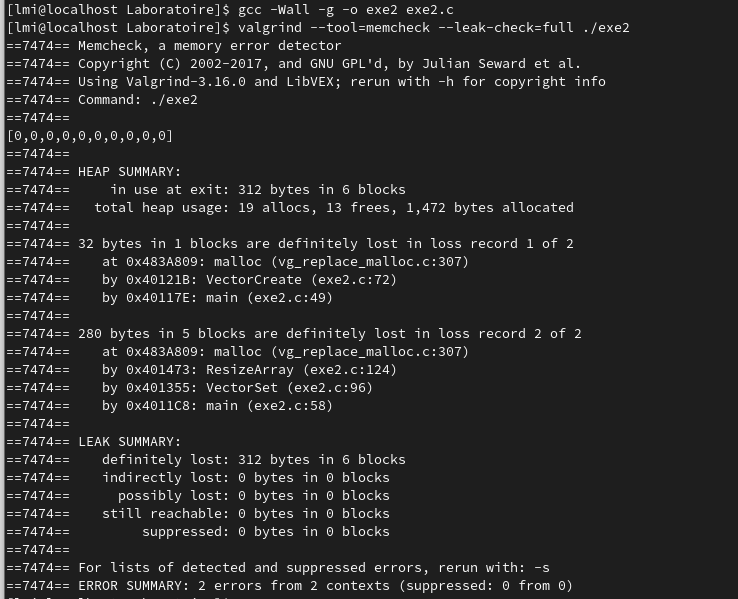


Voici les modifications apportées :

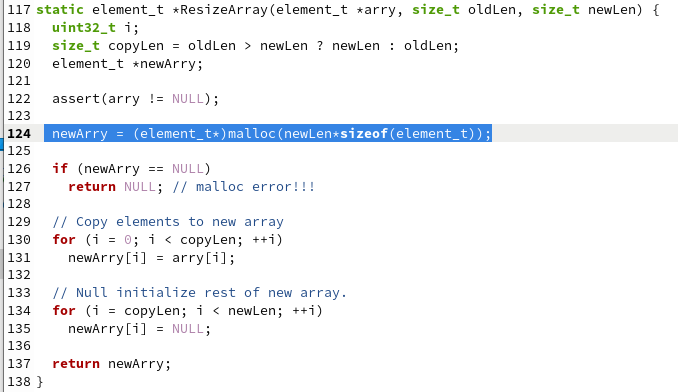
* Ligne 83-84 : Libérer la mémoire de tous les éléments du tableau
* Ligne 86 : : Libérer la mémoire du tableau



Après une nouvelle compilation et exécution de l’outil Valgrind, on peut voir que les erreurs détectées sont encore causées par des fuites de mémoire. Le nombre de byte perdu à encore diminué mais il reste des erreurs.



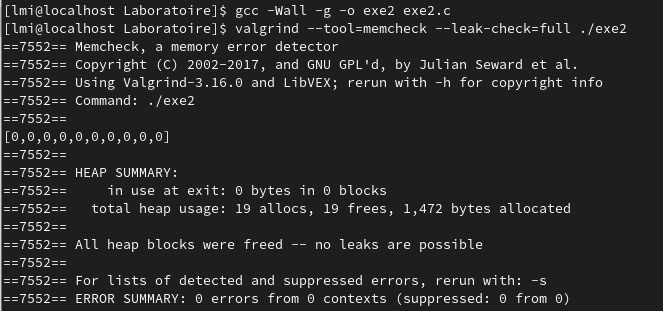
On peut voir que la fonction « ResizeArray » est aussi une source d’erreur. On peut donc y jeter un coup œil :



On voit que cette fonction alloue de la mémoire mais ne la libère jamais. De ce fait, l’ajout d’un simple appel à la fonction free devrait suffire :



Finalement, après une dernière compilation et exécution de l’outil Valgrind, une sortie sans erreur peut être observée :



## Conclusion

Ce laboratoire m’a permis de me familiariser avec l’outil Valgrind. En effet, cet outil est très appréciable car le langage C est plutôt permissif et il est vite arrivé de faire des erreurs. Je suis convaincu qu’il me sera fort utile pour ma vie professionnel car il est simple d’utilisation et très efficace.

# U-boot

U-boot de l’anglais « Universal-Boot » est un [logiciel libre](https://fr.wikipedia.org/wiki/Logiciel_libre), utilisé comme [chargeur d'amorçage](https://fr.wikipedia.org/wiki/Chargeur_d%27amor%C3%A7age), surtout sur les [systèmes embarqués](https://fr.wikipedia.org/wiki/Syst%C3%A8me_embarqu%C3%A9). U-Boot peut être divisé en deux étapes : la plate-forme chargerait un petit SPL (Secondary Program Loader), qui est une version allégée de U-Boot, et le SPL ferait la configuration matérielle initiale et le chargement de la version plus grande et complète de U-Boot.

## Introduction

L’objectif de ce laboratoire « U-boot » est de se familiariser avec ce logiciel et surtout d’apprendre à le manipuler afin de pouvoir effectuer des modifications quelconques de configuration. Par exemple, durant ce chapitre, nous allons voir trois différentes manipulations :

1. Changement de la configuration de u-boot (changement du prompt par défaut de u-boot).
2. Changement de la partition BOOT (vfat -> ext4).
3. Changement de l'initialisation du réseau.

## U-boot configuration

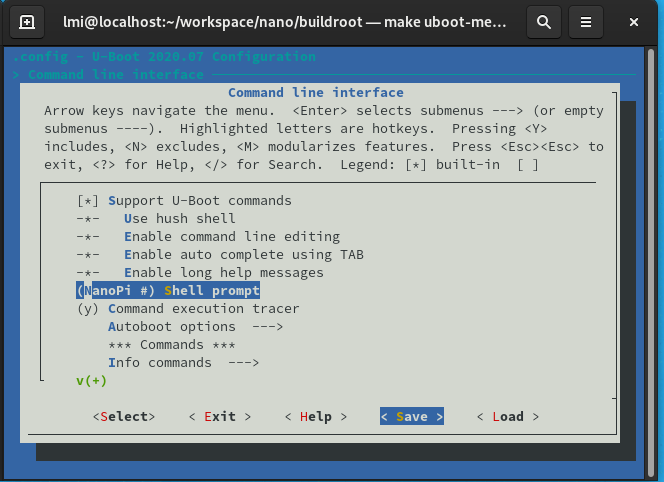
Dans ce chapitre nous allons voir comment changer le prompt par défaut de u-boot de « => » à « NanoPi # ».

### Manipulations

Pour commencer, il faut changer la configuration de u-boot :

*« make uboot-menuconfig »*

La configuration en question est dans le sous-menu « *Command line interface -> Shell prompt* » comme on peut le voir ci-dessous :

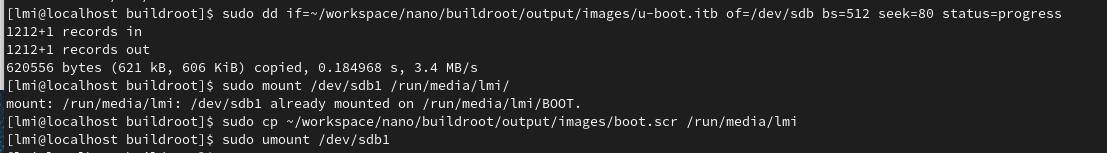


Il ne faut pas oublier de sauvegarder.

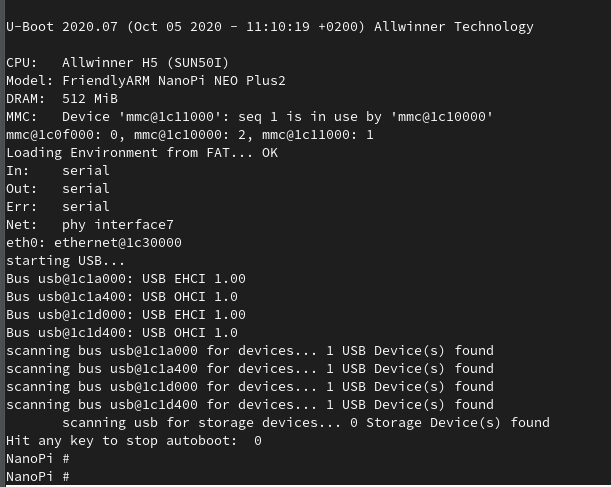
Ensuite, il est nécessaire de reconstruire u-boot grâce à la commande suivante :

*« make uboot-rebuild »*

*« make »*

Une fois la configuration modifiée, il faut copier les nouveaux fichiers (u-boot.itb et boot.scr) sur la carte SD :

Finalement, il est possible de visualiser la modification en redémarrant la cible avec la carte SD. Il faut appuyer sur une touche avant le démarrage automatique du kernel. Voici le nouveau prompt U-boot :



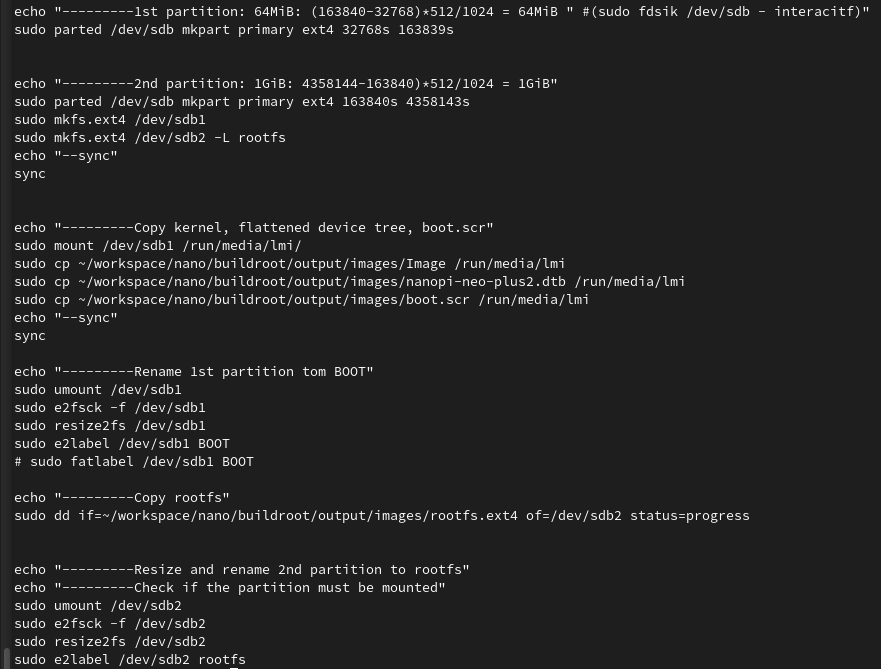
## BOOT partition ext4

Par défaut, la partition BOOT est en format vfat. Dans cette section, nous allons changer ce format en ext4. Ce dernier est en général plus apprécié pour plusieurs raisons :

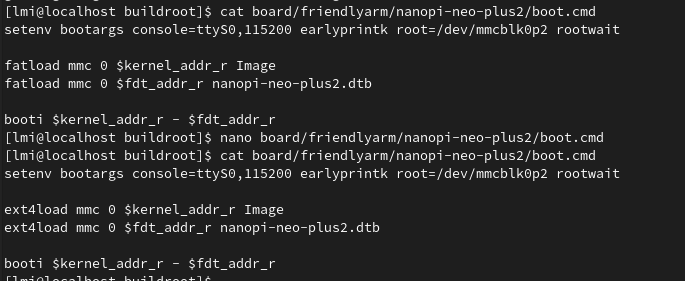
1. Ext4 est un système de fichiers journalisé.
2. La lecture et écriture est plus rapide qu’en vfat.
3. La sécurité est plus prise en charge sur ext4.

### Manipulations

Pour commencer, il faut modifier notre script qui génère la carte SD afin de modifier le format vfat en ext4. Voici les deux principales lignes à modifier :

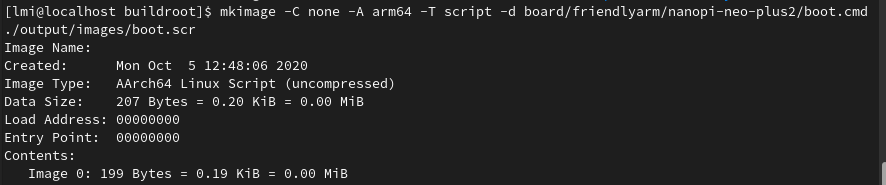


Ensuite, il est nécessaire de modifier les commandes exécutées par u-boot via le script « scr » afin qu’il charge l’image linux et le FDT (flattened device tree) depuis un foramt ext4 et non plus vfat. Pour ce faire, il faut modifier le fichier « boot.cmd » :



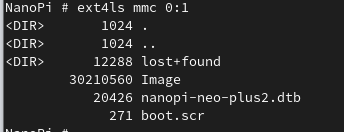
Sur l’image précédente, il est possible de voir où se trouve le fichier de commande de boot (boot.cmd). De plus, on peut voir le contenu du fichier avant et après la modification.

Une fois ce fichier modifié, il faut générer l’image de ce fichier pour U-boot via la commande « *mkimage* » :



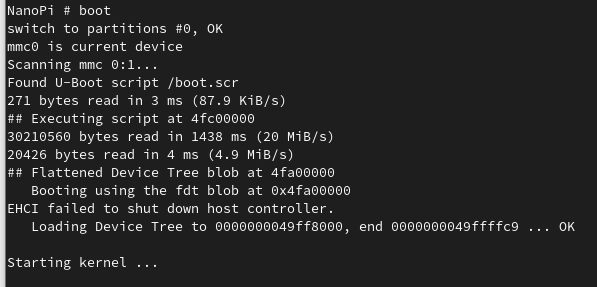
Il faut maintenant générer la carte SD avec le nouveau script.

Il est possible de confirmer la modification du format de la partition via U-boot en utilisant la commande « *ext4ls* » comme ceci :



*Remarque : « mmc 0 » correspond à la carte SD. « :1 » correspond à la partition 1 de la carte SD.*

Finalement, en tapant la commande « boot », le noyau linux se lance :

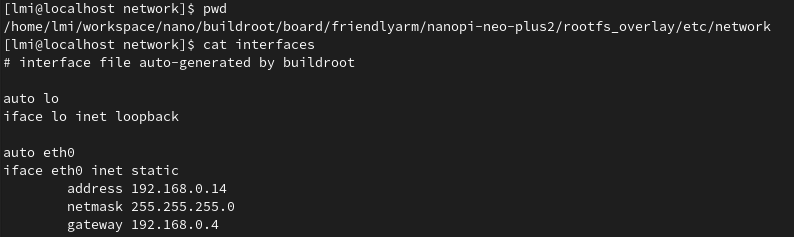


## Change network initialization

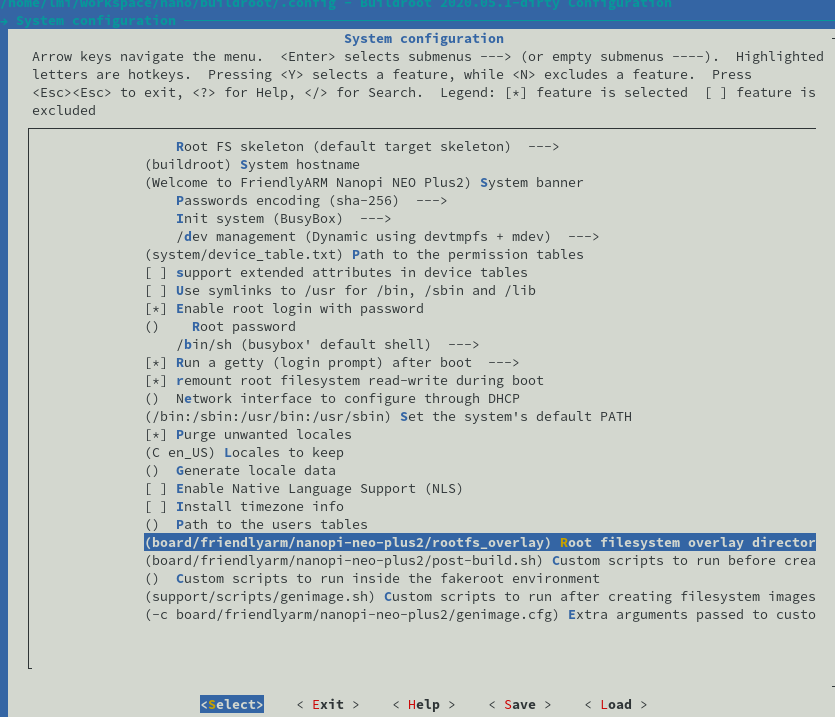
Linux initialise le réseau avec le script « /etc/init.d/S40network », qui lit le fichier de configuration « /etc/network/interfaces ». Cependant, par défaut, ce fichier de configuration ne configure par le réseau « eth0 » que nous utilisons.

Afin d’y remédier, il faut créer correctement le fichier de configuration et utiliser la fonctionnalité d’overlay afin que ce fichier soit toujours dans le rootfs de la cible.

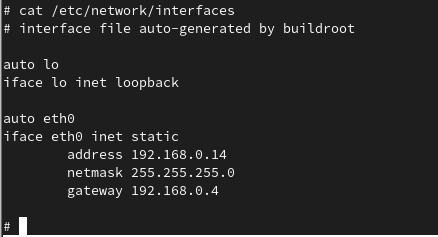
Pour commencer, il faut créer ce fichier dans le répertoire d’overlay associé à la board utilisée :



Ensuite, il faut s’assurer que Buildroot prend en compte la fonctionnalité d’overlay et qu’il possède le bon chemin jusqu’au rootfs d’overlay. Pour ce faire, il faut modifier la configuration de Buildroot si cela n’est pas fait via la commande « make menuconfig » :



Après avoir recompiler Buildroot et regénérer la carte SD, il est possible de voir le fichier de configuration précédemment crée :



## Compile Kernel

# Conclusion

## Difficultés rencontrées

## Compétences acquises

## Résultats obtenus

Date : 02.11.20

Nom de l’étudiant : Spinelli Isaia