
Étude et développement d'un module supportant le système de fichiers QNX6

Malik CHOUGAR - Lise MATET - Jean DE BONFILS LAVERNELLE - Mathis GREAU



RAPPORT FINAL DU PROJET DE MASTER 1

Faculté des Sciences et Techniques de l'Université de Limoges

Projet encadré par M.Damien SAUVERON et M. Maxime SAUVAYRE

Table des matières

| | | |
|----------|---|-----------|
| 1 | Introduction | 5 |
| 2 | Présentation de QNX | 7 |
| 2.1 | Histoire | 7 |
| 2.2 | Les différentes versions de QNX : | 9 |
| 2.2.1 | Les différentes versions de QNX RTOS : | 9 |
| 2.2.2 | Les différentes versions de QNX/NEUTRINO : | 10 |
| 2.3 | Architecture de QNX NEUTRINO | 10 |
| 2.4 | Le système de fichiers QNX6 | 13 |
| 2.5 | QNX Neutrino et les systèmes de fichiers | 14 |
| 2.6 | La gestion de processus sous le système d'exploitation QNX | 15 |
| 2.6.1 | La gestion de processus | 16 |
| 2.6.2 | La fonction posix_spawn() | 16 |
| 2.6.3 | La fonction Spawn() | 17 |
| 2.6.4 | La fonction fork() | 18 |
| 2.6.5 | La fonction exec*() | 20 |
| 2.7 | La gestion de la mémoire | 22 |
| 2.8 | Différents types de fichiers UNIX | 24 |
| 2.8.1 | Les fichiers de type « Regular » | 24 |
| 2.8.2 | Les fichiers de type « Directory » | 25 |
| 2.8.3 | Les fichiers de type « Symbolic Link » | 25 |
| 2.8.4 | Les fichiers de type « FIFO Special »(named pipe) | 25 |
| 2.8.5 | Les fichiers de type « Socket » | 26 |
| 2.8.6 | Les fichiers de type « Device File » | 27 |
| 3 | Autopsy et les ingest modules | 28 |
| 3.1 | Autopsy | 28 |
| 3.1.1 | Présentation | 28 |
| 3.1.2 | fonctionnalités | 29 |
| 3.2 | Les différents types de modules Autopsy | 30 |
| 3.3 | Développement d'ingest modules | 30 |
| 4 | Étude du système de fichiers par reverse Engineering des drivers de QNX6 | 33 |
| 4.1 | Présentation du reverse engineering | 33 |
| 4.2 | Légalité du processus de reverse engineering | 35 |
| 4.3 | Présentation de GHIDRA | 36 |
| 4.4 | Identification des Shared objects et Dynamic Linking | 37 |
| 4.4.1 | Dynamic Linker | 38 |
| 4.5 | Reverse-Engineering sur « mkqnx6fs » | 39 |
| 4.6 | Analyse du code de « mkqnx6fs » et "fs-qnx6.so" | 41 |
| 4.7 | Résultats obtenus du reverse engineering | 45 |

| | |
|--|------------|
| 5 Étude du système de fichiers en analysant l'hexadécimal de ce dernier | 49 |
| 6 Partitionnement | 54 |
| 6.1 Définition d'une partition | 54 |
| 6.2 Les différents types de partitionnement | 54 |
| 6.3 Le partitionnement sous les différents systèmes d'exploitation | 54 |
| 6.4 La table de partitionnement Master Boot Record (MBR) | 55 |
| 6.5 Partitionnement de QNX | 57 |
| 6.5.1 Quelques exemples de partitions présentes dans le système d'exploitation QNX : | 58 |
| 7 Architecture du système de fichiers de QNX6 | 59 |
| 7.1 Définition d'un Block Device ou Block Special File | 59 |
| 7.1.1 Interprétation d'un fichier normal comme étant un fichier de type Block Device | 60 |
| 7.2 Blocs | 61 |
| 7.3 Super Block | 61 |
| 7.4 Inodes | 65 |
| 7.5 Récupération des longs noms de fichiers | 70 |
| 7.6 Récupération et organisation des données | 73 |
| 7.7 Récupération des données effacées | 77 |
| 7.7.1 Lecture du BackUp Super block | 77 |
| 7.7.2 Autre méthode pour récupérer l'ID d'un Inode | 80 |
| 7.7.3 Récupération des données sans Inode ID | 81 |
| 8 Développement de l'ingest Module | 82 |
| 8.1 Définition des classes | 82 |
| 8.2 Développement du parser pour le système de fichiers QNX6 | 83 |
| 8.2.1 Fonction : parseSuperBlock | 83 |
| 8.2.2 Fonction : getInodesFromRootNode | 84 |
| 8.2.3 Fonction : getDirTree | 86 |
| 8.2.4 Récupération des données | 88 |
| 8.2.5 Fonction : getDirsAndFiles | 91 |
| 8.2.6 Fonction : getDataFromPTR | 95 |
| 8.2.7 Fonction récupérant les noms longs | 96 |
| 8.2.8 Fonction : getDeletedContent | 98 |
| 8.3 Interaction avec Autopsy | 99 |
| 9 Guide d'utilisation | 107 |
| 9.1 Installation | 107 |
| 9.2 Utilisation | 107 |
| 9.3 Visualisation de l'arborescence générée | 111 |
| 9.4 Résultats obtenus | 114 |

| | |
|---|------------|
| 10 Gestion de Projet | 115 |
| 10.1 Planning réel et répartition des tâches | 115 |
| 10.2 Réunions | 118 |
| 10.3 Organisation | 118 |
| 10.4 Imprévus | 118 |
| 10.5 Grille de participation | 119 |
| 11 Conclusion | 119 |
| 12 Annexe | 120 |
| 13 Bibliographie | 121 |
| 13.1 Code utile au développement de l'ingest module | 121 |
| 13.2 Présentation de QNX | 121 |
| 13.3 Reverse engineering | 121 |
| 13.4 Architecture du système de fichiers de QNX6 | 122 |
| 13.5 Autopsy | 123 |
| 13.6 Reverse Engineering | 123 |
| 13.7 Autre | 123 |

1 Introduction

Nous souhaitons tout d'abord remercier notre encadrant M. Damien SAUVERON qui nous a conseillés et orientés pour mener à bien notre projet. Nous souhaitons également remercier M. Maxime SAUVAYRE qui nous a également conseillés et proposés ce projet.

Le rapport est long mais nous avons voulu y inclure TOUT le travail réalisé cette année lors de la réalisation de ce projet. De plus, nous avons privilégié la lisibilité à travers de nombreuses captures d'écrans. Toutefois, les parties les plus importantes du projet sont les chapitres 7 et 8.

Nous allons à travers ce rapport vous détailler l'ensemble du travail que nous avons réalisé pour notre projet de fin d'année qui porte sur l'étude et le développement d'un module supportant le système de fichiers QNX6.

Il faut savoir que notre projet a tout d'abord été un projet de recherche. Le système d'exploitation QNX étant très peu documenté sur Internet, il nous a fallu beaucoup de temps pour comprendre les spécifications et les caractéristiques du système d'exploitation QNX et aussi pour comprendre ce qui nous été réellement demandé. Nous nous sommes donc mis à travailler sérieusement sur le projet depuis le mois d'octobre 2020 en commençant par faire toutes les recherches nécessaires pour la concrétisation de notre projet. A partir du mois de janvier 2021, nous avions décidé d'organiser des réunions hebdomadaires entre l'ensemble des membres du groupe afin de nous départager le travail à faire pour la semaine suivante et revoir l'état d'avancement de notre projet. Nous avions également organisé plusieurs réunions avec M. Damien SAUVERON afin de lui poser plusieurs questions au sujet de notre projet. Malheureusement, certains membres du groupe ont cessé de collaborer complètement ou en partie le projet pour des raisons personnelles.

Nous allons donc vous présenter dans ce rapport l'ensemble des points qui ont été traités pour mener à bien notre projet. Nous détaillerons tout d'abord les spécifications du système d'exploitation QNX : son histoire, ses différentes versions, son architecture, la gestion des processus sous QNX, etc. Nous détaillerons par la suite le logiciel de forensic Autopsy et nous expliquerons ce que sont les Ingest Modules. On expliquera le fonctionnement du système d'exploitation QNX et nous détaillerons l'étude du système de fichiers par reverse engineering des drives de QNX6. On réalisera par la suite une étude du système de fichiers en analysant l'hexadécimal et on détaillera également l'architecture de QNX6. On passera ensuite à la partie la plus importante du projet qui consistera à expliquer le développement de l'Ingest Module et à détailler les différents résultats obtenus. **Le code complet de l'ingest module est disponible en cliquant sur le lien :** <https://github.com/jdbonfils/QNX6-Files-System-Reader-Ingest-Module>

Une courte vidéo de présentation de l'ingest module développé est également disponible en cliquant sur le lien :

https://www.youtube.com/watch?v=H9FppPDLrpY&ab_channel=JeandeBonfils

2 Présentation de QNX

2.1 Histoire

UNIX est un système d'exploitation multitâche et multi-utilisateur créé par l'informatien amércain Ken Thompson en 1969. Plusieurs systèmes d'exploitation dérivés d'UNIX ont depuis vu le jour dont QNX.

QNX (Quantum Unix) est un système d'exploitation temps réel privatif (closed-source) de type UNIX. Il a été créé au début des années 80 par la société QNX Software Systems (anciennement Quantum Software Systems Limited), fondée par Dan Dodge et Gordon Bell, deux anciens étudiants canadiens de l'Université de Waterloo (Ontario, Canada). QNX est dérivé du système d'exploitation *UNIX*². Il a porté l'appellation QUNIX avant d'être renommé QNX pour des soucis de droits d'auteurs.

Il est constitué d'un micronoyau temps réel et d'un très grand nombre de programmes système de type UNIX. Au début des années 80. L'une des premières utilisations de ce système était dans les processeurs de type INTEL 8088. Puissant de ses presque 44 000 programmes système, QNX a d'abord été conçu pour les systèmes embarqués temps réel (Real Time Opearting System, RTOS) et a été utilisé dans la réalisation d'applications dans le domaine de l'industrie. Avec l'apparition des systèmes portables à la fin des années 80, les concepteurs de QNX ont décidé de revoir la structure du noyau et de l'adapter aux besoins actuels du marché, une nouvelle version nommée QNX4 voit le jour en s'inspirant du modèle POSIX (Portable Operating System Interface).



FIGURE 1 – Logo QNX

Vers la fin des années 90, avec l'apparition des architectures Symmetric Multiprocessing (SMP) où un seul et unique système d'exploitation contrôle de manière équitable plusieurs processeurs identiques qui sont connectés entre eux à une même et unique mémoire partagée, la société QNX Software Systems décide d'adapter son système d'exploitation à cette nouvelle architecture qui commence à équiper de plus en plus de machines : une nouvelle version nommée Neutrino voit le jour en 2001.

Neutrino connaîtra une ascension fulgurante à partir de 2004 notamment dans le monde de l'industrie automobile et des technologies liées aux systèmes d'infodivertissement avec l'intégration des systèmes embarqués dans les voitures. Neutrino équipera la plateforme QNX CAR Application Platform qui sera utilisée pour la gestion de la navigation (Satellite navigation), de la télécommunication (Wireless vehicle safety communications), le Vehicle Tracking ainsi que d'autres technologies liées au monde automobile. En 2004, On dénombra près de 200 modèles de voitures qui sont équipées du système d'exploitation Neutrino dans le monde en passant par Audi, Toyota, Porsche, BMW, Ford, Jaguar et Lincoln.

Conscients du futur prometteur des systèmes embarqués dans le monde de l'automobile, l'industrie, l'aéronautique etc, une dizaine de groupes internationaux souhaitent

racheter la société QNX Software Systems. Elle est finalement rachetée par le groupe international américain Harman International Industries, Incorporated qui est considéré comme étant l'un des principaux équipementier automobile des constructeurs allemands notamment en matière de systèmes d'infodivertissement. En 2010, la société canadienne BlackBerry Limited qui est connue sous le nom de Research In Motion (RIM) rachète la société. En Septembre 2010, BlackBerry Limited annonce la création d'une tablette équipée d'un système d'exploitation basé sur un noyau QNX. Il sera à la base de plusieurs systèmes d'exploitation du constructeur BlackBerry notamment BlackBerry OS, BlackBerry Tablet OS et BlackBerry 10. Le système d'exploitation QNX a aussi été utilisé par la société américaine Cisco Systems afin d'équiper les routeurs CRS, Cisco 12000 et Cisco ASR9000. Il est aussi à la base du système d'exploitation CISCO IOS-XR.



FIGURE 2 – Tablette BlackBerry avec un système d'exploitation nommé BlackBerry Tablet OS basé sur QNX.



FIGURE 3 – L'importance de QNX dans le secteur automobile

En Janvier 2013, QNX fait un grand pas dans le domaine de l'automobile en lançant une version 2.0 de la plate-forme QNX CAR Application Platform. Cette version sera considérée comme étant un standard par les firmes spécialisées dans le domaine de l'automobile. On dénombrera en 2016, près de 60 millions de véhicules équipés par ce système et aujourd'hui près de 60% des véhicules dans le monde le sont aussi.

Actuellement, QNX est massivement utilisé dans les systèmes embarqués liés à la sécurité. Nous citons comme exemple, les applications liées à la sécurité ferroviaire, les applications de gestion de construction urbaine, les systèmes de gestion de l'orientation des appareils ANTI-TANK, les applications liées aux domaines des mines et forages (Caterpillar), les applications militaires, les applications liées au domaine médical et au nucléaire.

2.2 Les différentes versions de QNX :

Depuis son développement en 1980, le système d'exploitation QNX a subi de nombreuses modifications et mises à jour afin de répondre aux besoins actuels du marché notamment aux besoins liés à la sécurité.

2.2.1 Les différentes versions de QNX RTOS :

Ci-dessous les différentes versions de QNX RTOS (REAL TIME OPERATING SYSTEM) ainsi que leur date d'apparition :

| QNX RTOS | |
|----------|---------|
| Date | Version |
| 1980 | |
| 1983 | BETA |
| 1984 | 1.0 |
| 1987 | 2.0 |
| 1989 | 2.21 |
| 1990 | 4.0 |
| 1994 | 4.1 |
| 1995 | 4.2 |
| 1995 | 4.22 |
| 1995 | 4.24 |
| 1997 | 4.25 |

2.2.2 Les différentes versions de QNX/NEUTRINO :

Ci-dessous les différentes versions de QNX/NEURTINO ainsi que leur date d'apparition :

| QNX RTOS | |
|------------|---|
| Date | Version |
| 1996 | 1.0 |
| 1998 | 2.0 |
| 1999 | 2.10 |
| 18/01/2001 | 6 |
| 2001 | 6.1.0 |
| 28/09/2001 | 6.1.0 (PATCH A) |
| 04/0/2002 | 6.2 |
| 18/10/2002 | 6.2(PATCH A) |
| 18/02/2003 | 6.2.1 |
| 03/06/2004 | 6.3 6.3.0 (SP1) 6.3.0 (SP2) 6.3.0 SP3 / OS 6.3.2 6.3.2 |
| 30/10/2008 | 6.4.0 |
| 05/2009 | 6.4.1 |
| 06/2010 | 6.5.0 |
| 11/07/2012 | 6.5 SP1 |
| 28/02/2014 | 6.6 |
| 04/01/2017 | 7.0 |
| 23/06/2020 | 7.1 |

2.3 Architecture de QNX NEUTRINO

Un système d'exploitation est un ensemble de programmes systèmes qui permettent de gérer les ressources d'une machine et les demandes faites par les logiciels de cette même machine. Le principal but est d'éviter les interférences entre les différents logiciels qui souhaitent accéder aux ressources. La partie principale des systèmes d'exploitation qui est considérée comme la partie la plus difficile à programmer et à concevoir est appelée micro-noyau (kernel). L'un des rôles principaux des micro-noyaux est de permettre à différents composants d'une même machine de communiquer entre eux. Nous allons présenter l'architecture du système d'exploitation QNX NEUTRINO.

QNX NEUTRINO est composé d'un micro-noyau qui constitue un ensemble de programmes système. Il est compatible avec plusieurs bibliothèques dynamiques, plusieurs architectures réseaux dont IPV6, IPV4, IPSec, FTP, HTTP, SSH, etc. Il est également compatible avec une large gamme de microprocesseurs : x86, ARM, XScale, PowerPC, MIPS

et SH-4. Bien. Le but de QNX est de fournir l'API POSIX (Portable Operating System Interface) à différents systèmes informatiques de technologies et de structures différentes. Malgré le fait que les systèmes d'exploitation POSIX soient créés sur une base UNIX, QNX NEUTRINO n'est pas basé sur UNIX. L'avantage de QNX Neutrino par rapport à un système d'exploitation de type POSIX est qu'il permet de mieux gérer les systèmes embarqués. L'un des principaux points forts de QNX NEUTRINO est qu'il est évolutif et flexible, les développeurs peuvent facilement rendre une version de QNX compatible avec les besoins actuels du marché en omettant ou en rajoutant des systèmes de fichiers, des interfaces graphiques et des réseaux et cela en très peu de temps. Le fait qu'il soit dérivé d'UNIX lui permet d'hériter des modèles de sécurité d'UNIX. Il est aussi réputé comme étant l'un des meilleurs OS (Operating System) pour les systèmes temps réel embarqués par sa capacité à gérer efficacement les threads, le changement de contexte, le multitâche etc. Contrairement aux systèmes d'exploitation monolithique où les services utilisateurs et les services du noyau sont implémentés sous le même espace d'adressage, le fait que QNX NEUTRINO soit composé d'un micro-noyau lui permet de mieux protéger sa mémoire. De plus, QNX bénéficie de quelques avantages comparé aux autres systèmes d'exploitations comme le fait que les drivers et le système de fichiers s'exécutent en dehors du Kernel évitant de causer des problèmes si l'un d'eux crashe. Ci-dessous, une figure qui illustre bien la différence entre les systèmes d'exploitation monolithiques et micro-noyaux.

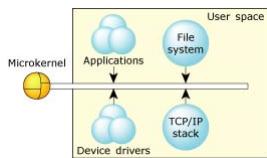


FIGURE 4 – Protection de la mémoire dans l'architecture micro-noyau



FIGURE 5 – Mémoire non protégée dans l'architecture monolitique

QNX NEUTRINO est certifié Common Criteria ISO/IEC 15408 Evaluation Assurance Level (EAL) 4+ qui est une certification de référence en matière de sécurité des produits et systèmes informatiques. QNX NEUTRINO est reconnu comme étant parmi l'un des OS les plus fiables. Le module appelé *procnto* est constitué du micro-noyau QNX ainsi que du gestionnaire de processus. A partir de la version 6.3.0, *procnto* permet aussi de gérer les Named Semaphores qui apparaissent dans l'arborescence /dev/sem. QNX est aussi compatible avec des centaines de fonctions POSIX .

QNX NEUTRINO est aussi composé d'un bus central (Software Bus) qui reçoit des données en sortie et envoie des données en entrée à un ensemble de services minimaux qui coopèrent entre eux pour effectuer une tâche donnée. On représente l'architecture de QNX NEUTRINO dans la figure ci-dessous :

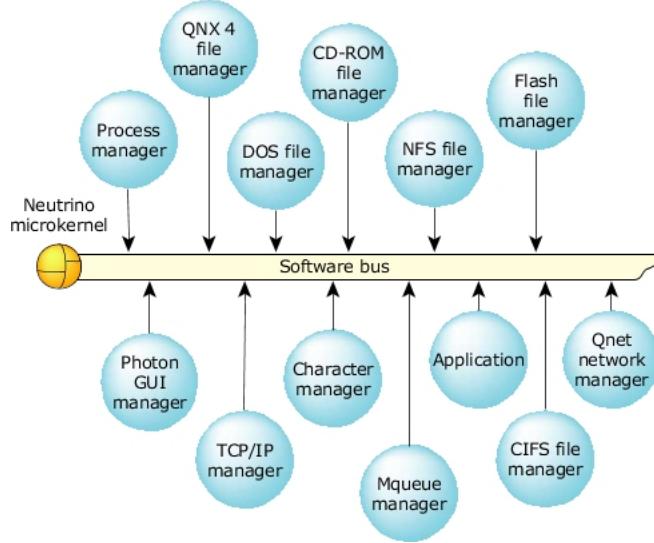


FIGURE 6 – L'architecture de QNX.

2.4 Le système de fichiers QNX6

QNX6 ressemble beaucoup aux systèmes d'exploitations dérivés d'UNIX. Il implémente des blocks, des inodes (index node) et des répertoires. De plus, il est possible de créer des systèmes de fichiers petit et grand boutiste (little-endian, big-endian) ce qui rend QNX6 particulièrement polyvalent surtout qu'il est utilisé dans une grande variété de systèmes embarqués qui peuvent disposer de boutismes différents.

Chaque fichier est divisé en blocs de taille fixe de 512, 1024, 2048 et 4096 octets. Ce choix est fait à la création du système de fichiers.

Chaque système de fichiers QNX6 dispose de deux Super Blocks qui contiennent toutes les informations globales du système de fichiers (ex : nombre total de blocs) et disposent d'un numéro de série sur 64 bits. Ce numéro de série sert à identifier le Super Block actif : on copie tous les blocs modifiés vers le Super Block inactif puis on reconstruit le système de fichiers. Une fois cette étape effectuée, le numéro de série du nouveau Super Block est augmenté de 1 (et il devient actif).

De plus chaque Super Block contient des Root Nodes pour chaque partie du système de fichiers (inode, Bitmap et Longfilename). Chacun de ces Root Nodes contient : la taille totale des données stockées et le niveau d'adressage. Si le niveau d'adressage est égal à 0, alors jusqu'à 16 blocs directs peuvent être adressés par chaque noeud (node). Si le niveau est à 1, alors on ajoute un niveau d'adressage indirect où chaque bloc indirect peut contenir (taille du bloc)/4 octets de pointeurs vers des blocs de données. Pour le niveau 2, on ajoute un autre niveau d'adressage indirect et on aura au maximum de $16 \cdot 256 \cdot 256 = 1048576$ blocs qui peuvent être adressés.

Les pointeurs inutilisés sont toujours mis à 0. Les données sont toujours contenues sur les niveaux les plus bas.

On trouve le premier Super Block à l'adresse 0x2000 (0x2000 étant la taille du bloc de démarrage). Le second se trouve après le premier : sa position peut être calculée grâce à l'information du nombre total de blocs contenus dans le premier Super Block. On trouve également à l'adresse 0x1000, la taille réservée pour chaque Super Block quelle que soit la taille des blocs.

2.5 QNX Neutrino et les systèmes de fichiers

QNX Neutrino fournit une variété de systèmes de fichiers et permet notamment de :

- Démarrer et arrêter dynamiquement des systèmes de fichiers
 - Faire fonctionner plusieurs systèmes de fichiers en même temps
 - Faire profiter les applications d'un chemin d'accès unifié quelque soit le système de fichiers utilisé

QNX Neutrino peut notamment utiliser les systèmes de fichiers suivants : Power-Safe filesystem (utilise QNX6), QNX 4 filesystem, DOS filesystem, Linux Ext2 filesystem. Au lancement, QNX Neutrino lance le système de fichiers voulu (par défaut Power-Safe) et les autres sont lancés comme des gestionnaires autonomes.

Comme il est commun de lancer plusieurs systèmes de fichiers sur QNX Neutrino, les développeurs ont créé des drivers et des bibliothèques pour éviter la répétition de code. Cela permet d'ajouter un nouveau système de fichiers avec un coût pour l'OS relativement bas.

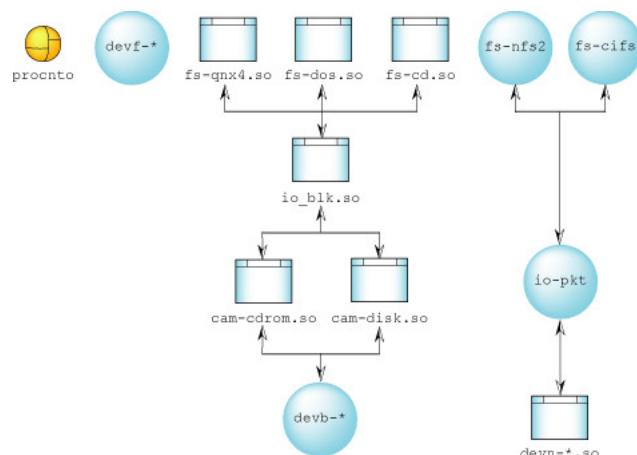


FIGURE 7 – Organisation des systèmes de fichier

On remarque sur le schéma que les systèmes de fichiers sont au-dessus de *io-blk* : Block i/O module. Le module *io-blk* est un gestionnaire de ressources qui envoie à chaque appareil physique un *block-special file*. Par exemple, dans le cas d'un disque dur */dev/hd0*, ces fichiers peuvent être accédés en utilisant les primitives POSIX usuelles (*open()*, *read()*...). Cette compatibilité POSIX ne fait pas tout car tous les systèmes de fichiers ne sont pas capables de fournir tous les services POSIX.

2.6 La gestion de processus sous le système d'exploitation QNX

Le système d'exploitation QNX permet comme n'importe quel autre système d'exploitation de gérer les processus d'une façon spécifique. De manière générale, un processus est composé du code, des données, des descripteurs de fichiers, d'une pile et des tables de signaux. Sous QNX, le gestionnaire de processus « Process Manager » permet de créer plusieurs processus de type POSIX. Chaque processus POSIX est composé de plusieurs threads POSIX (appelées aussi pthreads). Pour rappel, la norme POSIX a été définie en 1988 par l'IEEE (Institute Of Electrical and Electronic Engineers) et son appellation a été proposée par Richard Stallman. Elle a émergé d'un projet de standardisation des interfaces de programmation des logiciels compatibles avec les systèmes d'exploitation de type UNIX. La famille de normes POSIX est désignée par la norme IEEE 1003. Les threads POSIX sont quant à eux formellement désignés par la norme IEEE Std 1003.1c-1995 (POSIX. 1c, Threads Extensions) et ils représentent des sous-standards de la norme POSIX et décrivent une interface de programmation permettant de gérer les threads. Cette interface est disponible sur la plupart des systèmes d'exploitation modernes dérivés des systèmes UNIX comme Linux (développé par Linus Torvalds), Solaris (développé par Sun Microsystems), certaines versions modernes de BSD (créées par l'Université de Californie Berkeley) et certaines versions modernes de Mac OS X (développées par Apple).

Sous le système d'exploitation QNX Neutrino RTOS, il existe un module nommé *procnto* qui regroupe le noyau du système d'exploitation *microkernel* ainsi que le gestionnaire de processus *Process Manager*. La présence de ce module est primordiale pour assurer le bon fonctionnement du système d'exploitation. Ce module permet notamment de gérer les actions suivantes :

- La gestion de la création et destruction des processus.
- La gestion des attributs des processus désignés par l'uid (*User ID*) et le gid (*Group ID*).
- La gestion de la mémoire : librairies partagées, protection de la mémoire .
- La gestion des chemins d'accès.

Les processus de type *User* peuvent accéder directement aux fonctions du noyau du système d'exploitation *microkernel* en faisant directement appel aux fonctions du gestionnaire de processus et aux fonctions du noyau *kernel* en envoyant des messages au module *procnto*. Pour envoyer un message, un processus de type *User* doit utiliser la primitive *MsgSend**(). Il est important de souligner le fait que les threads qui s'exécutent dans le module *procnto* fonctionnent de la même manière que les threads qui appartiennent aux autres modules : ils évoquent le *microkernel* de la même manière. L'ensemble des threads qui composent un système partagent la même interface du noyau *kernel* : le fait que le code du gestionnaire de processus et le *microkernel* partagent le même espace d'adressage n'implique pas l'utilisation d'une interface spécifique.

2.6.1 La gestion de processus

Comme nous l'avons vu précédemment, la gestion de processus consiste à créer, détruire et gérer les attributs des processus. Les gestion des attributs des processus désigne la gestion des identifiants des processus *Process IDs*, la gestion des identifiants de groupe *Group IDs* et la gestion des identifiants des utilisateurs *User IDs*.

La fonction première du module *procnto* est de créer dynamiquement des processus qui vont directement dépendre de la responsabilité du module *procnto* pour la gestion de la mémoire et la gestion des chemins d'accès.

Nous allons ci-dessous, détailler un ensemble de fonctions nécessaires pour la gestion de processus et expliquer brièvement leur fonctionnement. Les fonctions qui seront abordées sont : *posix_spawn()*, *spawn()*, *fork()* et *exec*()*.

2.6.2 La fonction *posix_spawn()*

Cette fonction permet de créer un processus fils *child process* en spécifiant un fichier *executable* à charger. L'appel de cette fonction intervient généralement après un *fork()* suivi d'un *exec*()*. L'intérêt premier d'utiliser la fonction *fork()* suivi par la fonction *exec*()* est la flexibilité dans le changement de l'environnement par défaut, hérité par les processus fils. Cette approche est plus efficace car il n'est pas nécessaire de dupliquer l'espace mémoire : il faut juste détruire et remplacer le contenu de l'espace mémoire quand la fonction *exec*()* est appelée.

Il existe également une variante de la fonction « *posix_spawn()* » nommée « *posix_spawnp()* » qui recherche l'exécutable de manière indépendante en utilisant le chemin de l'appelant. Cette variante ne nécessite pas de préciser le chemin d'accès absolu du programme à « *spawn* ». On préférera l'utilisation de la fonction « *posix_spawn()* » au lieu de la fonction « *posix_spawnp()* ». Ci-dessous, la signature de la fonction « *posix_spawn()* » :

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
               const posix_spawn_file_actions_t *restrict file_actions,
               const posix_spawnattr_t *restrict attrp,
               char *const argv[restrict],
               char *const envp[restrict]);
```

FIGURE 8 – La signature de la fonction *posix_spawn()*

La fonction « posix_spawn() » nous donne accès aux différentes classes d'environnement suivantes :

- Les descripteurs de fichiers *file descriptors*
- Les processus utilisateurs *Process User* et *group ID*
- Les *signal masks*, *ignored signals* et les attributs du *scheduler*

Ci-dessous, la signature de la fonction « posix_spawnp() » :

```
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
                  const posix_spawn_file_actions_t *restrict file_actions,
                  const posix_spawnattr_t *restrict attrp,
                  char *const argv[restrict],
                  char *const envp[restrict]);
```

FIGURE 9 – La signature de la fonction posix_spawnp()

2.6.3 La fonction **Spawn()**

La fonction « spawn() » est similaire à la fonction « posix_spawn() » vue précédemment. La fonction « spawn() » possède plusieurs variantes qu'on détaillera par la suite. Elle permet de gérer les classes d'environnement suivantes :

- Les descripteurs de fichiers *file descriptors*
- Process Group ID
- Signal mask
- Les signaux ignorés *ignored signals*
- Le noyau *node* de création du processus
- Les règles *policies* de *scheduling*
- Les paramètres de *scheduling*
- La taille maximum de la pile *maximum stack size*
- Le *runmask* pour les systèmes SMP.

SMP est une architecture qui consiste à utiliser un nombre important de processeurs au sein d'un même ordinateur.

La fonction « spawn ()» présente plusieurs variantes que l'on va détailler ci-dessous :

- spawn() : avec le chemin d'accès explicite.
- spawnp() : fonction qui cherche le chemin d'accès et fait appel à la fonction « spawn() » pour le premier fichier « executable » qui correspond.
- spawnl() : elle fait appel à la fonction « spawn() » en prenant en compte les arguments passés par ligne de commande.
- spawnle() : elle fait appel à la fonction « spawnl() » avec comme paramètres les variables d'environnement passées de manière explicite.
- spawnlvp() : elle fait appel à « spawnp() » qui suit la commande de recherche de chemin.

-
- spawnlpe() : elle fait appel à « spawnlp() » qui prend en paramètres les variables d'environnement de manière explicite.
 - spawnv() : les paramètres sont passés par ligne de commande et ils sont pointés par un tableau de pointeurs.
 - spawnve() : spawnv() avec des variables d'environnement passées de manière explicite.
 - spawnvp() : elle fait appel à la fonction spawn() qui suit la commande de recherche de chemin.
 - spawnvpe() : elle fait appel à la fonction « spawnvp() » en prenant comme paramètres les variables d'environnement passées de manière explicite.

Après l'appel de la fonction spawn(), le processus fils hérite de plusieurs attributs du processus père tels que le Process Group ID, l'attribut *Session Membership*, les attributs *Real User ID* et *Real Group ID*, etc.

2.6.4 La fonction fork()

La fonction fork() est appelée à l'intérieur d'un processus courant (processus père) dans deux cas :

- Pour créer une nouvelle instance du processus en cours d'exécution
- Pour créer un processus qui exécute un nouveau programme

La fonction permet de créer un processus fils qui possède les mêmes attributs et le même code que le processus père (processus courant). On dit que le processus fils hérite des caractéristiques du processus père. Cependant, certaines données et caractéristiques sont propres au processus courant et le processus fils ne les hérite pas.

Parmi ces données et caractéristiques dont le fils ne peut pas hériter, on cite :

- Les minuteries (timers)
- Les verrous des fichiers (file locks)
- L'identifiant du processus courant (Process ID). Pour rappel, chaque processus possède un identifiant unique qui le distingue des autres processus
- Les signaux d'attente et les alarmes

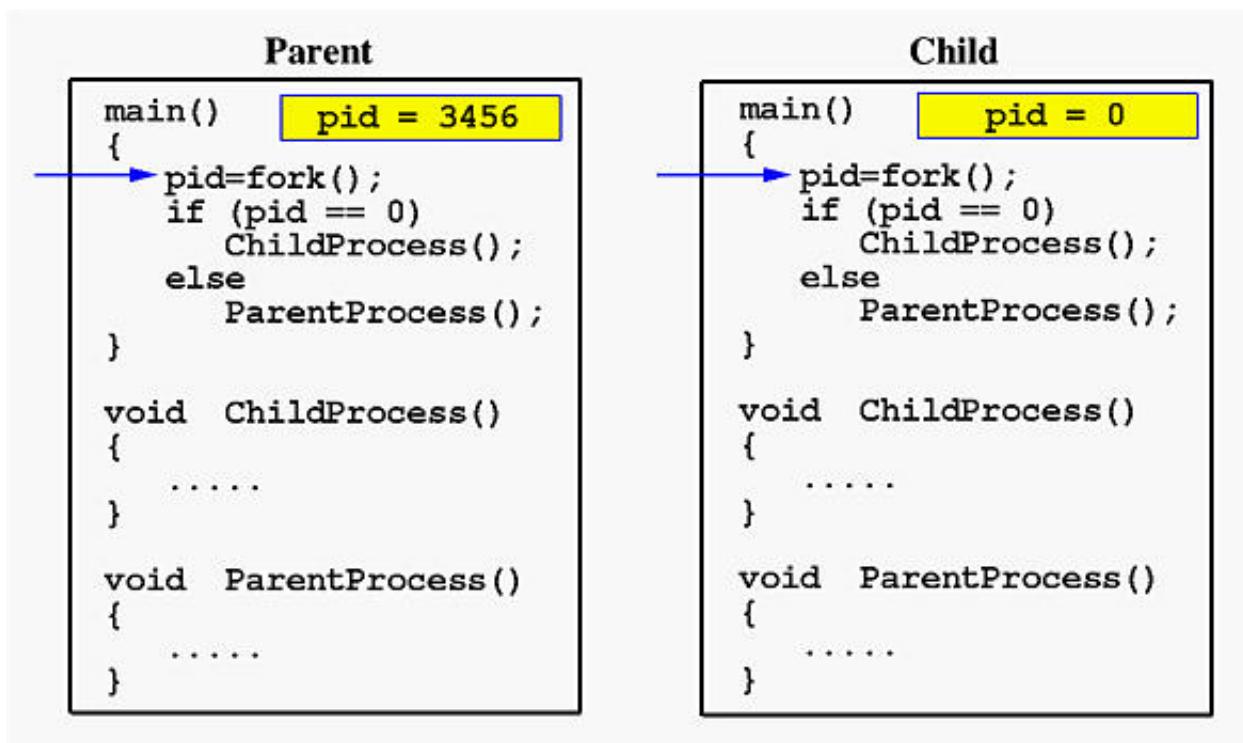


FIGURE 10 – Schéma explicatif de la fonction fork()

Ci-dessous, la signature de la fonction fork() :

```
#include <unistd.h>

pid_t fork(void);
```

FIGURE 11 – Signature de la fonction fork()

2.6.5 La fonction exec*()

La fonction `exec*()` est généralement appelée après l'appel de la fonction `fork()` afin de charger le processus fils. Pour qu'un fichier soit chargé en utilisant la commande `exec*()`, il doit être au format ELF (Executable and Linking Format). Cette dernière remarque s'applique aussi pour les appels des fonctions `spawn()` et `posix_spawn()`.

Lorsqu'elle est appelée, la fonction `exec*()` permet de remplacer le processus courant par un nouveau processus qui est extrait à partir d'un fichier de type *executable*. L'appel de la fonction `exec*()` est généralement suivi par un appel à la fonction `posix_spawn()`.

La fonction `exec*()` possède plusieurs variantes que nous allons détailler :

- `execl()` : fonction qui exécute la fonction `exec()` mais les paramètres sont passés par ligne de commande

- `execle()` : fonction qui appelle la fonction `execl()`, les variables d'environnement sont passées de manière explicite

- `execlp()` : fonction qui exécute la fonction `execl()` qui suit le chemin de recherche de commande (command search path)

- `execlepe()` : fonction qui exécute la fonction `execlp()` en prenant comme paramètres les variables d'environnement qui sont passées de manière explicite

- `execv()` : fonction qui exécute la fonction `execl()` avec un tableau de pointeurs qui pointe sur la ligne de commande.

- `execve()` : fonction qui appelle la fonction `execv()` avec comme paramètres, les variables d'environnement passées de manière explicite

- `execvp()` : fonction qui exécute la fonction `execv()` qui suit le chemin de recherche de commande (command search path)

- `execvpe()` : fonction qui exécute la fonction `execvp()` avec comme paramètres, les variables d'environnement passées de manière explicite

Ci-dessous, l'ensemble des signatures des fonctions vues ci-dessus :

```
#include <unistd.h>

extern char **environ;

int execl(const char * pathname, const char * arg, ...
          /*, (char *) NULL */);
int execlp(const char * file, const char * arg, ...
           /*, (char *) NULL */);
int execle(const char * pathname, const char * arg, ...
           /*, (char *) NULL, char *const envp[] */);
int execv(const char * pathname, char *const argv[]);
int execvp(const char * file, char *const argv[]);
int execvpe(const char * file, char *const argv[], char *const envp[]);
```

Feature Test Macro Requirements for glibc (see
[feature_test_macros\(7\)](#)):

```
execvpe():
_GNU_SOURCE
```

FIGURE 12 – Signature des fonctions exec*()

2.7 La gestion de la mémoire

Dans un ordinateur, en plus de la mémoire physique, il existe un second type de mémoire appelée mémoire virtuelle. La mémoire virtuelle permet au contenu de la mémoire de venir de n'importe où, que ce soit de la RAM, d'un système de fichiers, etc. L'un des principaux avantages de cette mémoire est qu'elle assure la sécurité des données qu'elle stocke en donnant une vue de la mémoire à n'importe quel moment. L'un des rôles principaux du gestionnaire de mémoire est de gérer la mémoire virtuelle. Ci-dessous, une illustration qui résume les différences entre la mémoire virtuelle et la mémoire physique :

| PHYSICAL MEMORY VERSUS VIRTUAL MEMORY | |
|---|---|
| PHYSICAL MEMORY | VIRTUAL MEMORY |
| Actual RAM and a form of computer data storage that stores currently executing programs | A memory management technique that creates an illusion to users of a larger physical memory |
| An actual memory | A physical memory |
| Faster | Slower |
| Uses the swapping technique | Uses paging |
| Limited to the size of the RAM chip | Limited by the size of the hard disk |
| Can directly access the CPU | Cannot directly access the CPU |

FIGURE 13 – Les différences entre la mémoire physique et la mémoire virtuelle

Dans toutes les machines et périphériques équipés du système d'exploitation QNX, il existe une unité de gestion de mémoire appelée *Memory Management Unit* abrégée MMU. Le rôle principal de cette unité de gestion de la mémoire est de traduire les adresses virtuelles en adresses physiques. Le MMU se lance au démarrage de la machine avant même le démarrage du kernel. Le travail du MMU consiste dans un premier à diviser la mémoire physique en blocs pages de 4-KB. Quand le gestionnaire de processus reçoit une requête, il alloue de l'espace dans la mémoire physique désignée par la System RAM ou *sysram*. Une partie de la mémoire physique contient aussi une image du système de fichiers. Le gestionnaire de mémoire ou *Memory Manager* crée un nombre important de pages virtuels pour servir toute la mémoire. Cependant, en général il n'y a pas de liens entre les pages mémoire et les mémoires virtuelles mais dans le cas où il y aurait un lien entre ces deux éléments, on dira que la page virtuel est soutenue par la page physique.

Ci-dessous, un schéma qui montre le partage de la mémoire physique par le MMU :

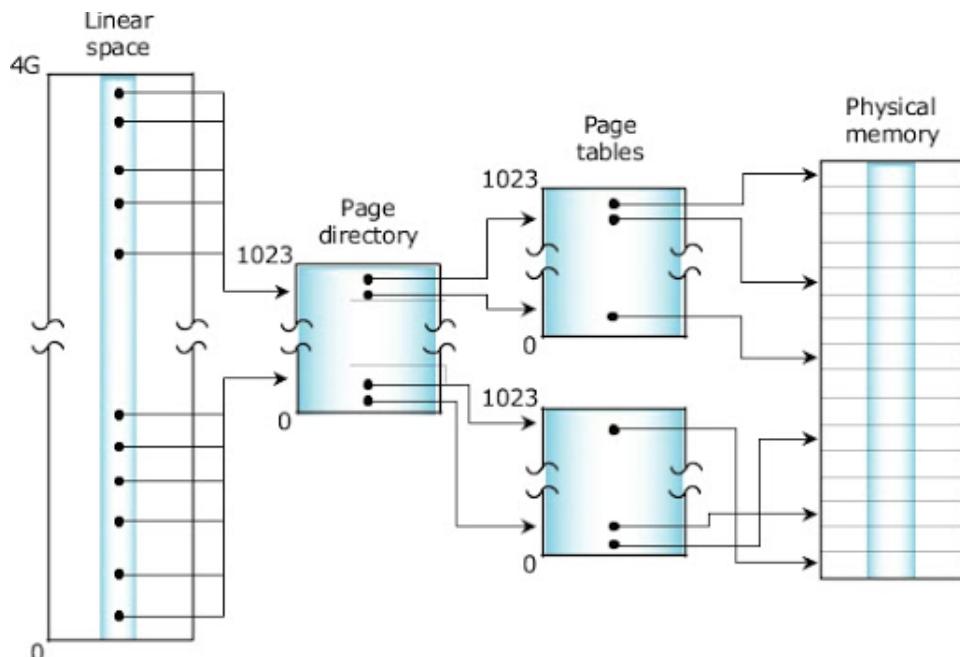


FIGURE 14 – Le partage de la mémoire physique par le Memory Management Unit

2.8 Différents types de fichiers UNIX

Un système d'exploitation de type UNIX dans sa version POSIX(*Portable Operating System*) est constitué de 7 types de fichiers qu'on détaillera par la suite :

- Regular
- Directory
- Symbolic link
- FIFO Special
- Socket
- Device File(Character Special)

Sous UNIX, à l'exception des périphériques réseaux, tous les éléments (fichiers et périphériques) sont gérés comme étant des fichiers et appartiennent à un système de fichiers particulier. Sachant qu'il existe plusieurs systèmes d'exploitation dérivés d'UNIX, comme QNX par exemple, chacun de ces systèmes d'exploitation aura une implémentation différente par rapport à UNIX en ajoutant éventuellement d'autres types de fichiers, il aura donc une architecture qui varie par rapport au système d'exploitation UNIX. Nous avons par exemple le système d'exploitation *Solaris* qui a été développé par Sun Microsystems. L'OS *Solaris* est dérivé du système d'exploitation UNIX, il intègre donc naturellement les différents types de fichiers présents sous UNIX en plus de son propre type de fichiers *Door* qui permet à deux processus différents de communiquer. Nous allons ci-dessous, détailler et expliquer les différents types de fichiers UNIX.

2.8.1 Les fichiers de type « Regular »

Ces fichiers sont des fichiers qui n'ont pas d'architecture précise sous Unix, leur architecture et implémentation dépend des logiciels qui les utilisent. Sous Unix, pour visualiser la liste des fichiers de type *regular*, il faut executer la commande suivante :

```
malik@malik-VirtualBox:~/Bureau/Réseau$ ls -l
total 36
-rwxrwxr-x 1 malik malik 353 mars 13 20:53 attaquant.py
-rwxrwxr-x 1 malik malik 574 mars 19 09:17 clientPortKnocking.py
```

FIGURE 15 – Exécution de la commande « ls –l »

2.8.2 Les fichiers de type « Directory »

Le type de fichiers « Directory » représente le type de fichiers le plus utilisé sous UNIX. Pour obtenir la liste des fichiers de type *Directory*, il faut utiliser la commande avec l'option « d » :

```
malik@malik-VirtualBox:~/Bureau/Réseau$ ls -dl  
drwxrwxr-x 4 malik malik 4096 mars 19 11:14 .
```

FIGURE 16 – Exécution de la commande « ls –dl »

d : signifie que c'est un fichier de type *Directory*

« rwx » : pour le premier groupe de trois symboles : son propriétaire peut lire, écrire(modifier) et exécuter.

« rwx » : pour le deuxième groupe de trois symboles : le groupe peut lire, écrire et exécuter.

« r-x » : pour le troisième groupe de trois symboles : le reste du monde peut uniquement lire, exécuter le fichier sans pouvoir le modifier.

2.8.3 Les fichiers de type « Symbolic Link »

Un fichier de type *Symbolic Link* est un fichier qui fait référence à un autre fichier, il contient généralement une représentation textuel du chemin qui permet de mener au fichier référencé.

En exécutant la commande *ls –l*, les fichiers de type *Symbolic Link* sont caractérisés par la lettre « l » avant le premier groupe qui détermine les droits d'accès du propriétaire du fichier.

2.8.4 Les fichiers de type « FIFO Special »(named pipe)

Si deux processus appartiennent à un seul et unique processus parent et sont lancés par le même utilisateur, ils pourront communiquer à travers un fichier de type *FIFO Special* (communication inter-processus). La sortie du premier programme (output) constituera l'entrée du second (input). Ce mécanisme de communications entre différents programmes via des fichiers de type *FIFO Special* constitue l'un des points forts des systèmes d'exploitation UNIX.

Les fichiers de type *FIFO Special* peuvent être créés n'importe où dans le système de fichiers. Pour créer un fichier de type *FIFO Special*, il faut utiliser la commande *mkfifo* comme dans l'exemple ci-dessous :

```
malik@malik-VirtualBox: ~/Bureau/Réseau  
malik@malik-VirtualBox:~/Bureau/Réseau$ mkfifo mypipe
```

FIGURE 17 – Création d'un fichier de type *FIFO Special* avec la commande *mkfifo*

Les fichiers de type *FIFO Special* sont reconnaissables en exécutant la commande « ls -l ». Le 1er groupe permettant de définir les droits d'accès du propriétaire sera alors précédé par la lettre « p ». Il est important de souligner qu'ils permettent d'échanger les informations que dans un seul sens (unidirectionnelle).

```
prw-rw-r-- 1 malik malik 0 mars 19 14:48 mypipe
```

FIGURE 18 – Analyse d'un fichier de type « FIFO Special »

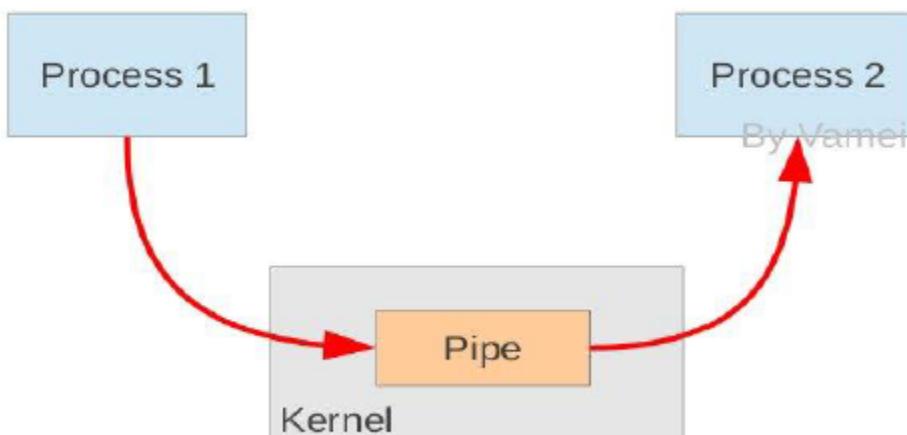


FIGURE 19 – Schéma montrant la communication unidirectionnelle via un fichier de type *FIFO Special* ou *Named Pipe*

2.8.5 Les fichiers de type « Socket »

Un fichier de type *Socket* permet à deux processus de communiquer et d'échanger des informations (communication inter-processus). Il permet en utilisant les appels systèmes « sendmsg() » et « recvmsg() » de faire transiter les fichiers de description *file descriptors* des deux processus. Contrairement aux fichiers de type *FIFO Special*, les fichiers de type *Socket* permettent de faire transiter les données dans les deux sens : *communication bidirectionnelle*.

Quand on exécute la commande « ls -l », les fichiers de type *Socket* sont caractérisés par la lettre « s » avant le premier groupe de 3 symboles qui définit les droits d'accès du propriétaire.

2.8.6 Les fichiers de type « Device File »

Les fichiers de type *Device File* décrivent les droits d'accès d'un périphérique particulier et les opérations permises sur ce périphérique. Il existe deux types de fichiers de type *Device* : les fichiers de type *Character* qui fournissent un accès à un périphérique d'entrée/sortie comme par exemple le *Terminal File*, le *Null File* ou le *File Descriptor File*.

Il existe aussi un second type de *Device File* : les fichiers de type *Block Device*. Les fichiers de ce type sont autorisés à accéder aux Entrées/Sorties des périphériques de bloc qui font des Entrées/Sorties tamponnées, c'est-à-dire que les données sont collectées dans une mémoire tampon et quand la quantité de données collectées est équivalente à la taille d'un bloc, le transfert se fait.

Pour les fichiers de type *Character Device*, quand on exécute la commande « `ls -l` », les fichiers obtenus sont caractérisés par la lettre « `c` » avant le premier groupe de trois symboles qui définit les droits d'accès du propriétaire.

Pour les fichiers de type *Block Device*, quand on exécute la commande « `ls -l` », les fichiers obtenus sont caractérisés par la lettre « `b` » avant le premier groupe de trois symboles qui définit les droits d'accès du propriétaire.

3 Autopsy et les ingest modules

3.1 Autopsy

3.1.1 Présentation



FIGURE 20 – Logo d’Autopsy

Autopsy est un logiciel gratuit et Open Source permettant de réaliser un travail d’investigation sur des sources de données telles que des images disques, des fichiers VM (Virtual Machine file), des disques locaux ou encore des répertoires.

Le logiciel est disponible sur Windows principalement même si une version moins complète existe sur Linux. Pour analyser une source de données, l’utilisateur va faire appel à des *Ingest Modules* qui vont réaliser certaines tâches pour en extraire des informations intéressantes. Certains de ces modules sont présents de base dans Autopsy comme par exemple :

1. *Extension Mismatch Detector* permettant de détecter les fichiers ayant une mauvaise extension.
2. *Recent Activity Module* extrait l’activité de l’utilisateur enregistrée par les navigateurs WEB, les programmes installés et le système d’exploitation.
3. *EXIF Parser module* extrait des informations contenues dans un fichier image telle que la localisation, la date, l’heure ou encore le modèle d’appareils photo.

En plus des ingest modules inclus de base dans Autopsy, l’utilisateur peut aussi importer des ingest modules développés par des personnes tierces (3rd-Party Modules). La liste de ces ingest modules est disponible en cliquant sur le lien : https://github.com/sleuthkit/autopsy_addon_modules/tree/master/IngestModules. Il est également possible de développer ses propres ingest modules en Python ou en Java grâce à l’API d’Autopsy mettant à disposition des classes et des méthodes permettant de réaliser des opérations sur les fichiers.

L'utilisation classique d'Autopsy se modélise par un workflow en quelques étapes :

1. La création d'un *Case* (l'équivalent d'un projet).
2. L'ajout d'une source de données sur laquelle on va réaliser les analyses.
3. L'analyse des résultats obtenus par les ingest modules choisis.
4. Éventuellement, la génération d'un rapport sur les résultats obtenus.

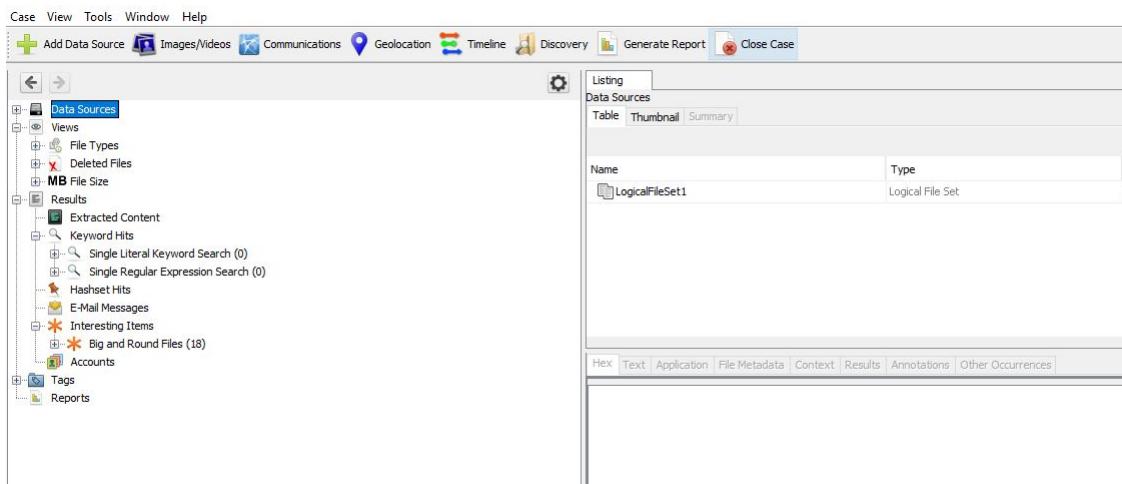


FIGURE 21 – Interface principale d'Autopsy

3.1.2 fonctionnalités

Autopsy a donc de multiples avantages comme sa facilité d'utilisation grâce à son interface graphique, mais surtout son extensibilité grâce à la possibilité d'ajouter des modules développés par la communauté. De plus, le logiciel est complètement gratuit et Open Source.

Voici la liste des principales fonctionnalités d'Autopsy :

- Collaboration avec d'autres examinateurs.
- Analyse de la chronologie : Affichage des événements du système dans une interface graphique pour aider à identifier l'activité.
- Recherche par mots-clés.
- Artefacts WEB : Extraction de l'activité WEB des navigateurs courants pour aider à identifier l'activité des utilisateurs.
- Analyse d'activités : Identification des documents récemment consultés et des périphériques USB.
- Analyse des raccourcis.
- Analyse des courriers électroniques.
- Extrait les informations de la localisation géographique et de la caméra (fichiers JPEG).
- Tri des types de fichiers : regroupage des fichiers par type pour trouver tous les documents et images.

-
- Lecture des médias : Permet de visualiser des vidéos et des images dans l'application et ne nécessite pas de visionneur externe.
 - Analyse robuste de système de fichiers : **Prise en charge des systèmes de fichiers courants, notamment NTFS, FAT12/FAT16/FAT32/ExFAT, HFS+, ISO (CD-ROM), Ext2/Ext3/Ext4, Yaffs2 et UFS de The Sleuth Kit.**
 - Filtrage du Hash Set
 - Extraction des chaînes de caractères Unicode
 - Détection du type de fichiers basée sur les signatures et la détection de non-concordance d'extensions.
 - Signalement des fichiers et des dossiers en fonction de leur nom et de leur chemin d'accès.
 - Prise en charge d'Android : Extraction des données à partir de SMS, journaux d'appels, contacts, Tango, Mots avec des amis, et plus encore.

Le logiciel Autopsy repose en grande partie sur l'outil forensic *The Sleuth Kit (TSK)* qui est un ensemble de librairies et de commandes permettant l'investigation de fichiers de façon non intrusive.

3.2 Les différents types de modules Autopsy

Plusieurs types de modules existent dans Autopsy. On cite notamment : les ingest modules abordés précédemment et les Report Modules. Ces derniers sont utilisés le plus souvent après l'extraction de données. Ils permettent de mettre en forme les données obtenues et génèrent un rapport sous différents formats tels que HTML ou Excel. D'autres modules existent comme les *Content Viewers* ou les *Result Viewers* mais ils ne seront pas utiles dans le cadre de la réalisation de notre projet.

3.3 Développement d'ingest modules

Les modules qui nous intéressent principalement pour ce projet sont les Ingest Modules. Ils vont nous permettre d'extraire les données à partir d'une source. Deux types d'Ingest Module existent :

La première catégorie de module appelée *File Ingest Modules* où chaque fichier (même ceux contenus dans des dossiers compressés) d'une source de données va être analysé par ce type de module. Ils vont permettre de réaliser différentes opérations sur chacun de ces fichiers indépendamment des autres.

La seconde catégorie de module appelée *Data Source-level Ingest Modules* permet l'extraction de fichiers à analyser depuis une source de données. Contrairement au *File Ingest Module*, le module s'occupe d'extraire les fichiers qu'il veut analyser. Dans le cadre du projet, ce sont donc de ce type d'ingest module dont nous aurons besoin.

Les modules sont créés en Jython qui est un langage de programmation permettant d'utiliser des classes Java en Python et l'héritage de classes Java par des classes Python.

Tout d'abord, on importe toutes les classes nécessaires pour développer l'ingest module.

```
32
33
34     import jarray
35     import inspect
36     from java.lang import System
37     from java.util.logging import Level
38     from org.sleuthkit.datamodel import SleuthKitCase
39     from org.sleuthkit.datamodel import AbstractFile
40     from org.sleuthkit.datamodel import ReadContentInputStream
41     from org.sleuthkit.datamodel import BlackboardArtifact
42     from org.sleuthkit.datamodel import BlackboardAttribute
43     from org.sleuthkit.datamodel import TskData
44     from org.sleuthkit.autopsy.ingest import IngestModule
45     from org.sleuthkit.autopsy.ingest.IngestModule import IngestModuleException
46     from org.sleuthkit.autopsy.ingest import DataSourceIngestModule
47     from org.sleuthkit.autopsy.ingest import FileIngestModule
48     from org.sleuthkit.autopsy.ingest import IngestModuleFactoryAdapter
49     from org.sleuthkit.autopsy.ingest import IngestMessage
50     from org.sleuthkit.autopsy.ingest import IngestServices
51     from org.sleuthkit.autopsy.ingest import ModuleDataManager
52     from org.sleuthkit.autopsy.coreutils import Logger
53     from org.sleuthkit.autopsy.casemodule import Case
54     from org.sleuthkit.autopsy.casemodule.services import Services
55     from org.sleuthkit.autopsy.casemodule.services import FileManager
56     from org.sleuthkit.autopsy.casemodule.services import Blackboard
```

FIGURE 22 – Imports à réaliser

La méthode classique pour créer un Ingest Module consiste à utiliser le design pattern Fabrique en définissant une classe Factory dans laquelle on définit un certain nombre d'éléments comme le nom de l'ingest module.

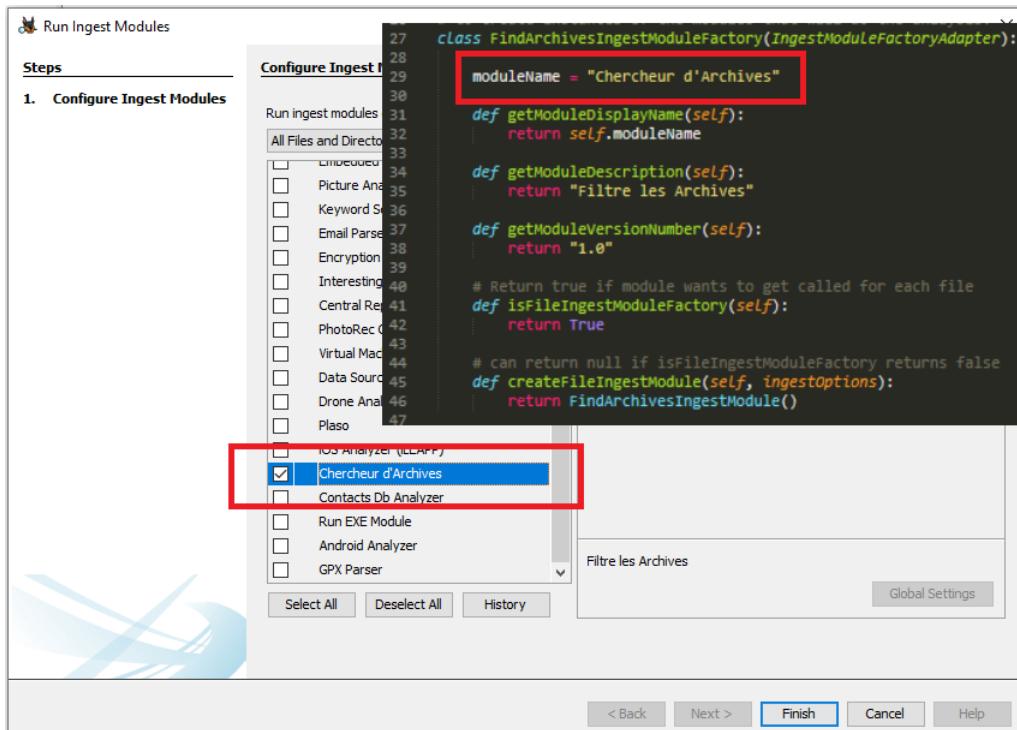


FIGURE 23 – Classe Facotry

Puis on crée la classe de l’Ingest Module. Dans cette classe, va être défini trois méthodes importantes :

- La méthode *startUp()* correspondant à l’initialisation, C’est aussi dans cette méthode que l’on réalise les opérations susceptibles d’échouer.
- La méthode *process()* est la méthode dans laquelle on va réaliser l’analyse de chaque fichier présent dans la source de données.
- La méthode *shutDown()* où l’on définit la libération des ressources.

Ci-dessous, un exemple simple où l’on cherche à récupérer tous les fichiers archives d’une source de données. Tout d’abord, on ignore tout ce qui n’est pas un fichier (répertoire...). Ensuite, si le fichier est une archive, on va l’ajouter à une sous-liste de la liste *interesting file*.

```
def process(self, file):  
    # Use blackboard class to index blackboard artifacts for keyword search  
    blackboard = Case.getCurrentCase().getServices().getBlackboard()  
  
    # On skip si c'est pas un fichier  
    if ((file.getType() == TskData.TSK_DB_FILES_TYPE_ENUM.UNALLOC_BLOCKS) or (file.getType() == TskData.TSK_DB_FILES_TYPE_ENUM.UNUSED_BLOCKS) or (file.isFile() == False)):  
        return IngestModule.ProcessResult.OK  
  
    #On recherche les archives  
    if (file.getNameExtension().contains('zip') or file.getNameExtension().contains('rar') or file.getNameExtension().contains('gzip')):  
  
        #Créer une sous-partie dans 'interesting file' où seront rassemblé les archives.  
        art = file.newArtifact(BlackboardArtifact.ARTIFACT_TYPE.TSK_INTERESTING_FILE_HIT)  
        att = BlackboardAttribute(BlackboardAttribute.ATTRIBUTE_TYPE.TSK_SET_NAME.getTypeID(), FindArchivesIngestModuleFactory.moduleName, "Fichiers Archives")  
        art.addAttribute(att)  
  
        try:  
            # index the artifact for keyword search  
            blackboard.indexArtifact(art)  
        except Blackboard.BlackboardException as e:  
            self.log(Level.SEVERE, "Error indexing artifact " + art.getDisplayName())  
  
        # Fire an event to notify the UI and others that there is a new artifact  
        IngestServices.getInstance().fireModuleDataEvent(ModuleDataEvent(FindArchivesIngestModuleFactory.moduleName, BlackboardArtifact.ARTIFACT_TYPE.TSK_INTERESTING_FILE_HIT))  
  
    return IngestModule.ProcessResult.OK
```

FIGURE 24 – Méthode process réalisant l’analyse de fichiers

4 Étude du système de fichiers par reverse Engineering des drivers de QNX6

4.1 Présentation du reverse engineering

L'ingénierie inverse, ou retro-ingénierie, ou reverse engineering ou reverse est un procédé qui permet de déterminer le processus de fonctionnement interne d'un système. Il possède plusieurs domaines d'application : industriel, informatique, électronique, etc. Dans notre cas, nous détaillerons par la suite l'application du reverse dans le domaine informatique et dans le cadre du système d'exploitation QNX. Le but premier de l'application de la méthode de reverse est d'analyser un composant ou système afin de le dupliquer, comprendre son fonctionnement et éventuellement l'améliorer.

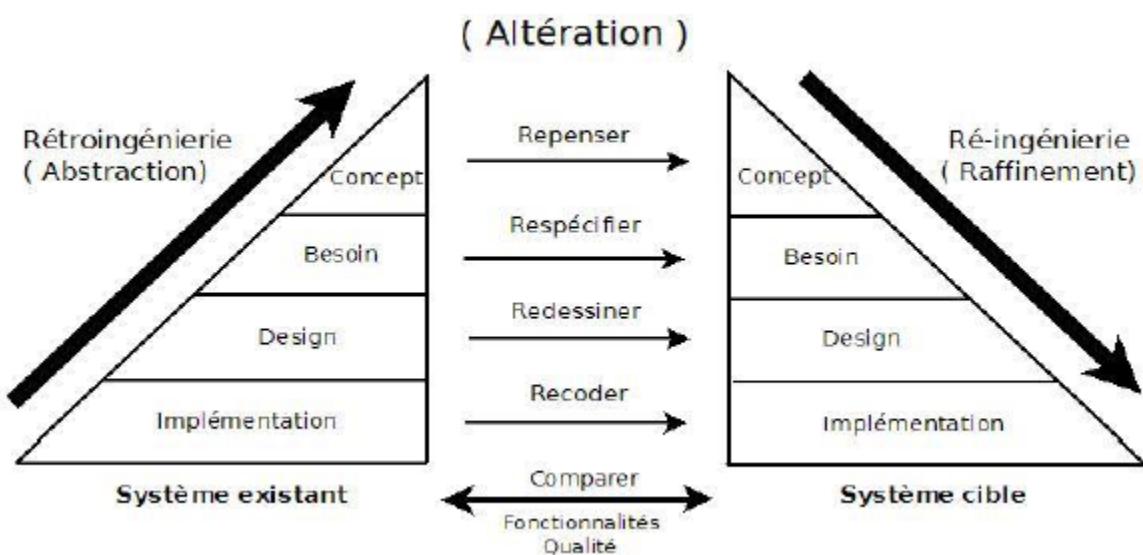


FIGURE 25 – Cycle de vie du processus de Reverse Engineering

Historiquement, le reverse a été évoqué pour la première fois quand les Romains ont copié le procédé de standardisation employé par l'empire Carthaginois afin d'imposer leur puissance sur l'ensemble des empires de la Méditerranée.

Dans le domaine de l'électronique, le reverse consiste à étudier les différents circuits qui constituent un composant électronique, récupérer le code assembleur du « firmware », qui est un programme intégré dans le composant et est indispensable à son bon fonctionnement. On parle de « désassemblage ».

Dans le domaine industriel, nous avons par exemple l'utilisation du procédé de reverse engineering pour reproduire des pièces détachées dont le fabricant n'assure plus le support.

Ci-dessus, un schéma qui illustre le processus de reverse-engineering :

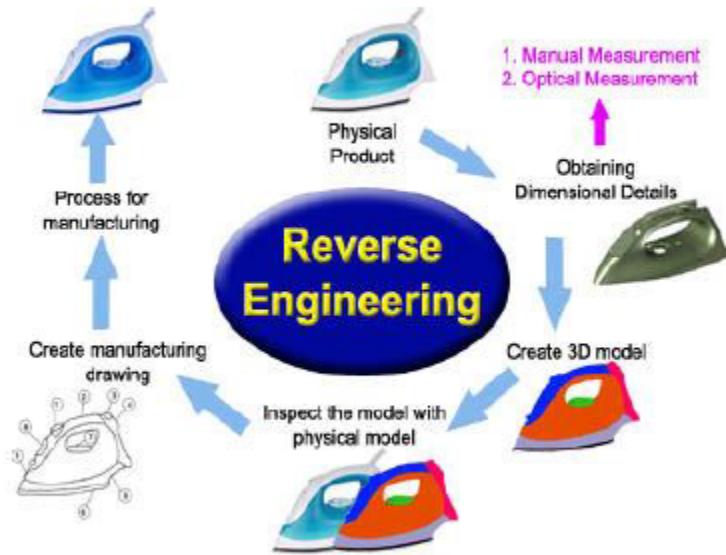


FIGURE 26 – Schéma du processus de Reverse Engineering

Dans le domaine informatique, on évoquera par la suite l'utilisation du logiciel « Ghidra », l'utilisation du reverse dans le domaine informatique et notamment son utilisation pour étudier le fonctionnement du système d'exploitation QNX. On utilise des désassemblateurs ou décompilateurs pour analyser ces logiciels. On évoque par exemple le projet Samba qui a été créé en 1992 et qui a donné lieu à la réimplémentation des protocoles SMB/CIFS, qui sont des protocoles permettant le partage de ressources sur des réseaux locaux sur des machines équipées du système d'exploitation Windows, sous GNU/Linux en utilisant le procédé de reverse engineering. Dans le monde des jeux vidéos, on parle de « cracking », qui consiste à créer une copie d'un jeu vidéo dont la valeur est parfois onéreuse. Le reverse engineering peut être aussi utilisé pour écrire les pilotes des périphériques comme les webcams et scanners. La technique de reverse engineering peut être aussi utilisé dans le domaine de la cryptographie qu'on détaillera par la suite.

Le processus de reverse engineering est aussi utilisé pour définir des logiciels de défense ou antivirus. Par exemple, l'étude de fichiers binaire malicieux « rootkit » et virus informatique permet d'anticiper les failles et de développer des antivirus permettant de réagir de manière efficace contre ces menaces. L'étude de fichiers binaire en utilisant le procédé de reverse engineering étant facile, on privilégiera ce type de langage de programmation appelé aussi langages de programmation semi-compilés.

```

00401578 EC FF OR AL,BP
00401579 8D75 EB LEA EST,CHORD PTR SS:[EBP+10]
0040157D 6F0E85 MOVZX EDX,BYTE PTR DS:[ESI]
00401583 009400 E4FFFF MOV BYTE PTR SS:[EBP+ECX-1C].AL
00401587 6AC0 CL AL
00401589 FECB DEC ECX
0040158B 48 EC DEC ESI
0040158D 4AC9 TEST CL,CL
00401591 ^75 ED JNC SHORT Max++_do.0040157D
00401593 6B75 FB MOV EST,CHORD PTR SS:[EBP-8]
00401595 6A 46 PUSH AD
00401598 59 POP ECX
00401599 F2105 NEG WORD PTR ES:[ED1],WORD PTR DS:[ESI]
0040159B 4C90 XOR EAX,EAX
0040159D 644C FF 00 MOV BYTE PTR SS:[EBP-13,0]
0040159F 55F6 XOR ESI,ESI
004015A1 7E 03 JMP SHORT Max++_do.00401526
004015A2 8A40 FD MOV CL,BYTE PTR SS:[EBP-9]
004015A5 FE45 FF INC BYTE PTR SS:[EBP+1]
004015A8 009400 EF MOVZX EDWORD PTR SS:[EBP+10]
004015A9 604005 E4FEFFFF LEA EDX,WORD PTR SS:[EBP+ECX-11C]
004015B0 6200 ADD CL,BYTE PTR DS:[EBX]
004015B8 8A10 MOU DL,BYTE PTR DS:[EBX]
004015B9 8840 FD MOV BYTE PTR SS:[EBP-31,CL]
004015BD 009400 E4FFFF MOVZX ECX,CL
004015C0 800000 E4FFFF MOV EDX,WORD PTR SS:[EBP+ECX-11C]
004015C2 85E5 FE LEA EC000000 PTR SS:[EBP+ECX-11C]
004015C4 8A11 MOU DL,BYTE PTR DS:[EBX],DL
004015C5 8810 MOU BYTE PTR DS:[EBX],DL
004015CE 8A55 FE MOU DL,BYTE PTR SS:[EBP-2]
004015D1 9911 MOV BYTE PTR DS:[ECX],DL
004015D3 6B6009 MOVZX ECX,BYTE PTR DS:[ECX]
004015D5 009400 MOU DL,BYTE PTR DS:[EBX]
004015D9 85C0 MOU ECX,BX
004015D9 S1E1 FF000000 PHD ECX,0FF
004015E1 009400 E4FEFFFF MOU AL,BYTE PTR SS:[EBP+ECX-11C]
004015E3 60149E LEA EDX,WORD PTR DS:[ESI+ED1]
004015E5 3602 XOR EDX,WORD PTR DS:[ED1],AL
004015E7 48 EC INC ECX
004015E8 4BF5 CMP ESI,EBX
004015F0 ^7C B0 JL SHORT Max++_do.004015A2
004015F2 BE FF000000 MOU ESI,BF
004015F7 8D87 FF000000 LEA EDX,WORD PTR DS:[EDI+FF]
004015FD 6B40 ED MOU ECX,WORD PTR SS:[EBP-14]
00401600 0F8E0001 MOUZX ECX,BYTE PTR SS:[ECX+EBX]
00401604 8B45 FB MOU EDX,WORD PTR SS:[EBP-8]
00401606 89C11 MOU EDX,WORD PTR DS:[ECX+ECX]
00401609 8909 MOU BYTE PTR DS:[EBX],CL
0040160C 8BCE MOU ECX,ESI
0040160F 4E DEC ESI
00401610 48 DEC ECX
00401613 85C9 TEST ECX,ECX
00401615 ^75 E9 JNC SHORT Max++_do.004015F0
00401617 6B40 ED MOU EDX,WORD PTR DS:[EBP-9],ED1
00401619 6B45 FB MOU EDX,WORD PTR SS:[EBP-8]
0040161A 25E5 ED SUB EDX,WORD PTR SS:[EBP-14],EBX
0040161D 0SF8 ADD EDX,EBX
0040161F 897D F0 MOU EDX,WORD PTR SS:[EBP-10],ED1
00401622 90B45 E4 CMP EDX,WORD PTR SS:[EBP-10]
00401625 ^7E 00 JE Max++_do.00401578
00401625 FF75 00 PUSH EDWORD PTR SS:[EBP+8]

```

Decrypt 1

Decrypt 2

Last Cycle

Next Block

FIGURE 27 – Reverse Engineering sur des rootkits

4.2 Légalité du processus de reverse engineering

La plupart des sociétés éditrices de logiciels interdisent d'utiliser le processus de reverse engineering sur leurs logiciels. Ils mettent généralement cela en évidence dans les CLUF (Contrat de Licence Utilisateur Final) ou EULA (End User License Agreement). Cependant, dans de nombreux pays, l'utilisation du reverse engineering est autorisé par la loi et les clauses imposées par les sociétés ne sont pas toutes reconnues par ces états car ces derniers redéfinissent eux-mêmes leurs propres lois.

En France, l'utilisation du reverse est autorisée par la loi mais avec des limites. Pour plus d'informations, il faut consulter les exceptions spécifiques définies dans l'article L 122-6-1 du code de la propriété intellectuelle.

4.3 Présentation de GHIDRA

GHIDRA est un logiciel de Reverse Engineering (SRE – Software Reverse Engineering). Il a été créé par la section Cybersécurité de la NSA et a été diffusé en version Open Source le 04 Avril 2019. Il permet d'étudier le fonctionnement interne de logiciels tels que les malwares et virus. C'est un logiciel qui est très utilisé dans le domaine de la cybersécurité notamment pour tester les vulnérabilités des systèmes et des réseaux informatiques. Ghidra est téléchargeable sur le site : ghidra-sre.org et est actuellement (Février 2021) disponible en version 9.2.2 pour l'ensemble des systèmes d'exploitation (macOS, Linux et Windows). Pour installer Ghidra, il faut tout d'abord télécharger le Zip File en cliquant sur le lien donné précédemment. Il faudra aussi télécharger Java Development Kit 11+ (JDK 11+) et Java Runtime 8+.

Java Development Kit 11 : oracle.com/fr/java/technologies/javase-jdk11-downloads.html

Java Runtime 8 : <https://www.java.com/fr/download/>

Il existe plusieurs logiciels permettant de faire du Reverse-Engineering tels que GHIDRA , IDA PRO, CFF Explorer, API Monitor, WinHex, Hiew, Fiddler, Relocation Section Editor, Scylla, Pied, etc.

Dans notre cas, nous utiliserons le logiciel « GHIDRA » présenté ci-dessus pour faire du Reverse-Engineering sur la commande « mkqnx6fs ».



FIGURE 28 – Logo de GHIDRA

4.4 Identification des Shared objects et Dynamic Linking

Pour comprendre le fonctionnement du système de fichiers QNX6 nous avons dû réaliser du reverse engineering sur les drivers de QNX6. Cela nous permettant de voir comment le système de fichiers est créé et organisé. Pour cela nous avons dû dans un premier temps identifier les fichiers intéressant à analyser par reverse engineering qui serait susceptible de nous en apprendre plus sur le système de fichiers. Après quelques recherches sur internet, nous avons identifié la commande *mkqnx6fs* permettant de formater un système de fichiers QNX 6. Le fonctionnement de cette commande est donc très intéressant pour notre projet. En effet, comprendre comment le système de fichiers est créé nous a énormément aidés à développer, par la suite, l'Ingest Module récupérant les données du système de fichiers.

Syntax:

```
mkqnx6fs [-Bq] [-b blocksize] [-e endian] [-g groups]
           [-i inodes] [-n blocks] [-O options] [-o options]
           [-r percent] [-T type] [-u uuid] host
```

FIGURE 29 – Syntaxe de la commande *mkqnx6fs*

Cependant cette commande utilise des shared object, ".so" ou encore appelé Dynamically Linked Shared Object Libraries. Ces fichiers sont des librairies contenant du code précompilé qui peut être réutilisé dans un programme. Elles fournissent des fonctions, des routines, des classes, des structures de données...etc. C'est en quelque sorte un homologue au .dll de Windows.

Les shared objects sont utilisés de deux façon :

- 1) Lié dynamiquement au moment de l'exécution. Les bibliothèques doivent être disponibles pendant la phase de compilation/liaison. Les Shared Objects ne sont pas inclus dans le composant exécutable mais sont liés à l'exécution.
- 2) Chargé/déchargé dynamiquement et lié pendant l'exécution en utilisant les fonctions du système de chargement de liaison dynamique ou dynamic linking.

4.4.1 Dynamic Linker

Le Dynamic Linker est la partie d'un système d'exploitation qui charge et lie les Shared Libraries nécessaires à un programme au moment de l'exécution. Le système d'exploitation et le format du programme déterminent le fonctionnement et l'implantation du Dynamic Linker. **Le lien est souvent considéré comme un processus réalisé lors de la compilation du programme, tandis qu'un Dynamic Linker est une partie spéciale d'un système d'exploitation qui charge des bibliothèques partagées externes dans un processus en cours d'exécution, puis lie ces bibliothèques partagées de manière dynamique au processus en cours d'exécution.** Cette approche est également appelée Dynamic Linking .

La commande mkqnx6fs utilise très probablement des Shared Objects contenant des fonctions susceptibles de nous intéresser pour le projet. D'après la documentation disponible sur internet, il semblerait que la commande mkqnx6fs utilise le Shared Object *fs-qnx6.so*. Cependant, lorsque qu'on utilise la commande "ld" qui permet de voir quelles Shared Libraries sont utilisées dans un programme, on ne retrouve aucune trace de *fs-qnx6.so*.

```
jean-Aspire-E5-575G:~/ftp$ ldd -v mkqnx6fs
 linux-gate.so.1 (0xf7f30000)
 libc.so.3 => not found
jean-Aspire-E5-575G:~/ftp$ ldd --version
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
```

FIGURE 30 – Vérification des shared libraries utilisées par mkqnx6fs

De plus quand on analyse la commande mkqnx6fs par reverse-engineering avec Ghidra, on ne retrouve aucune trace du fichier *fs-qnx6.so*. Après quelques recherches, nous avons trouvé que cela venait du fait que le lien se faisait automatiquement.

En effet, un programme peut ne pas savoir quelles fonctions il doit appeler avant de s'exécuter. Voici un exemple, provenant de https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.sys_arch/topic/dll.html et confirmant que *fs-qnx6.so* est bien utilisé par mkqnx6fs :

Considérons un pilote de disque "générique". Il démarre, sonde le matériel et détecte un disque dur. Le pilote charge alors dynamiquement le code io-blk pour gérer les blocs du disque, car il a trouvé un périphérique orienté bloc. Maintenant que le pilote a accès au disque au niveau des blocs, il trouve deux partitions présentes sur le disque : une partition DOS et une partition Power-Safe (QNX6). Plutôt que de forcer le pilote du disque à contenir des pilotes du système de fichiers pour tous les types de partitions qu'il peut rencontrer, nous avons fait simple : il n'a aucun pilote de système de fichiers ! Au moment de l'exécution, il détecte les deux partitions et sait alors qu'il doit charger le code du système de fichiers fs-dos.so et fs-qnx6.so pour gérer ces partitions.

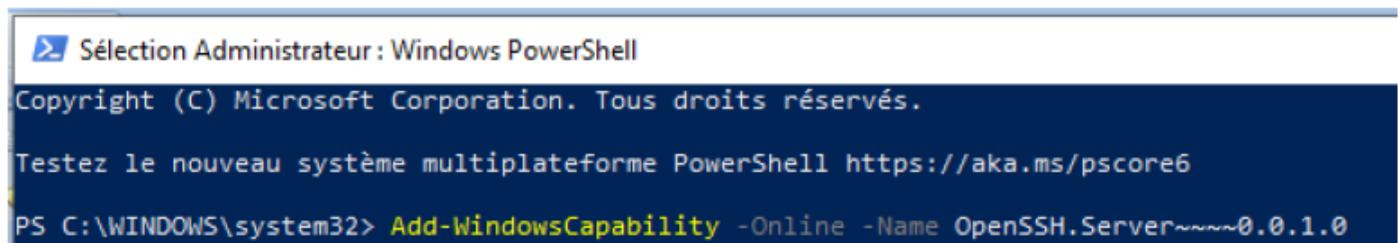
Maintenant que nous connaissons les fichiers intervenant dans la création du système de fichiers QNX6, nous pouvons les analyser par reverse engineering avec Ghidra.

4.5 Reverse-Engineering sur « mkqnx6fs »

Après avoir installé le framework GHIDRA, il faut dans un premier temps importer le fichier de la commande *mkqnx6fs* à partir de la machine virtuelle QNX. Il existe plusieurs manières d'importer un fichier ou un dossier à partir de la machine virtuelle QNX. Pour réaliser cela, nous avons utilisé le protocole SSH associé au numéro de port 22.

On lance en mode administrateur le logiciel Windows PowerShell sur le système d'exploitation Windows et on exécute les commandes suivantes :

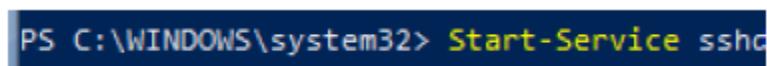
1 - Add-WindowsCapability -Online -Name OpenSSH.Server 0.0.1.0



```
Sélection Administrateur : Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.
Testez le nouveau système multiplateforme PowerShell https://aka.ms/pscore6
PS C:\WINDOWS\system32> Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

FIGURE 31 – Ajouter des Windows Capabilty

2- Start-Service sshd



```
PS C:\WINDOWS\system32> Start-Service sshd
```

FIGURE 32 – exécution de la commande Start-Service sshd

3- New-NetFirewallRule -Name sshd -DisplayName 'OpenSSH Server (sshd)' -Enabled True -Direction Inbound -Protocol TCP -Action Allow -LocalPort 22



```
PS C:\WINDOWS\system32> New-NetFirewallRule -Name sshd -DisplayName 'OpenSSH Server (sshd)' -Enabled True -Direction Inbound -Protocol TCP -Action Allow -Loca
```

FIGURE 33 – Gestion du firewall

4 -On se connecte ensuite en mode administrateur (root user) sur la machine virtuelle QNX et on lance la commande suivante :

la commande « *mkqnx6fs* » se trouvant dans le dossier sbin, on lance la commande scp :
scp /sbin/mkqnx6fs NomUtilisateur@AdresseIPLocale :EmplacementSurLaMachine-DeDestination



```
# scp /sbin/mkqnx6fs Malik@192.168.137.1:C:\
```

FIGURE 34 – Exécution de la commande scp

Il faudra ensuite introduire le mot de passe de la machine destinatrice. Enfin, la commande « mkqnx6fs » sera disponible à l'emplacement précisé comme dernier paramètre de la commande scp. Dans l'exemple ci-dessus, nous avons choisi le disque C.

Après avoir importé la commande « mkqnx6fs » sur le framework GHIDRA, on obtient des informations sur la commande « mkqnx6fs » que nous allons détailler ci-dessous :

| Import Results Summary | |
|------------------------|--|
| i | Project File Name: mkqnx6fs |
| | Last Modified: Sun Feb 28 12:01:49 WAT 2021 |
| | Readonly: false |
| | Program Name: mkqnx6fs |
| | Language ID: x86:IE:32:default (2.9) |
| | Compiler ID: GCC |
| | Processor: x86 |
| | Endian: Little |
| | Address Size: 32 |
| | Minimum Address: 08048000 |
| | Maximum Address: .note.gnu.build-id::0804815f |
| | # of Bytes: 70775 |
| | # of Memory Blocks: 31 |
| | # of Instructions: 180 |
| | # of Defined Data: 443 |
| | # of Functions: 242 |
| | # of Symbols: 307 |
| | # of Data Types: 40 |
| | # of Data Type Categories: 2 |
| | Created With Ghidra Version: 9.2.2 |
| | Date Created: Sun Feb 28 12:01:46 WAT 2021 |
| | ELF File Type: executable |
| | ELF Original Image Base: 0x8048000 |
| | ELF Prelinked: false |
| | ELF Required Library [0]: libc.so.3 |
| | Executable Format: Executable and Linking Format (ELF) |
| | Executable Location: /C:/mkqnx6fs |
| | Executable MD5: bcc2dd0cbc07e39c66aeeff88d5a3a56 |
| | Executable SHA256: 9a886beel6dffeb0b5ef0e5c0cc648ef6903bf7030f02cbf473d0dd3ab501bb84 |
| | FSRL: file:///C:/mkqnx6fs?MD5=bcc2dd0cbc07e39c66aeeff88d5a3a56 |
| | Relocatable: false |

FIGURE 35 – Les différentes informations recueillies

- Le nom du projet GHIDRA : mkqnx6fs.
- Les modifications sont autorisées sur le code obtenu après l'exécution du décompilateur (Readonly = false).
- Le nom du programme importé pour effectuer le Reverse-Engineering : mkqnx6fs
- L'identifiant du langage est x86 :L2 :32 :default
- L'identifiant du compilateur est « gcc » (Gnu Compiler Collection) qui est un logiciel libre qui permet de compiler plusieurs langages de programmation tels que le langage

C, C++, Fortran, Objective-C, Java et Ada.

- Le processeur de type « x86 ».
- Little-Endian (petit-boutienne) qui nous informe que l'orientation démarre avec les octets de poids faibles. Nous illustrons la notion de gros-boutienne et petit-boutienne par la figure ci-dessous :

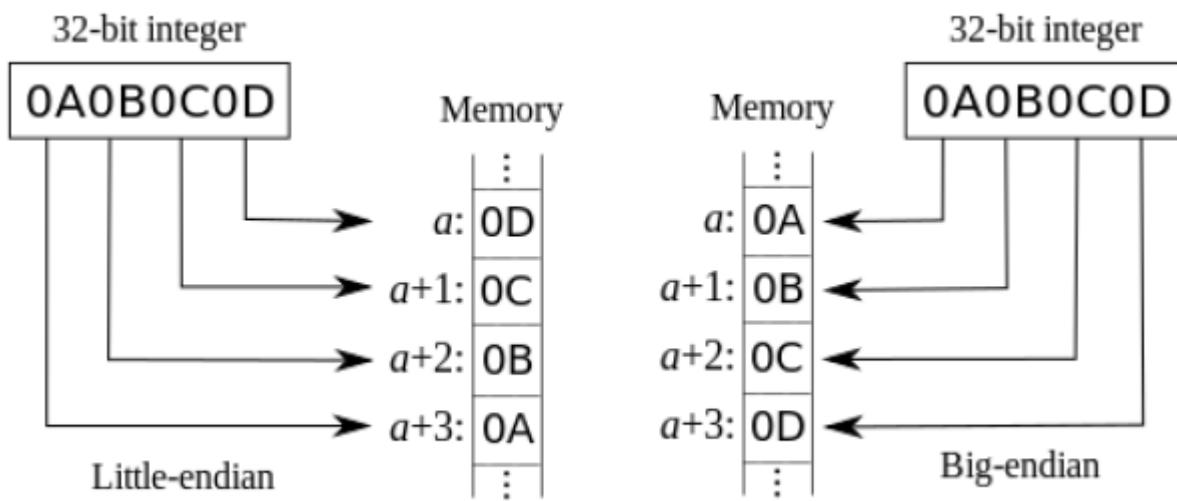


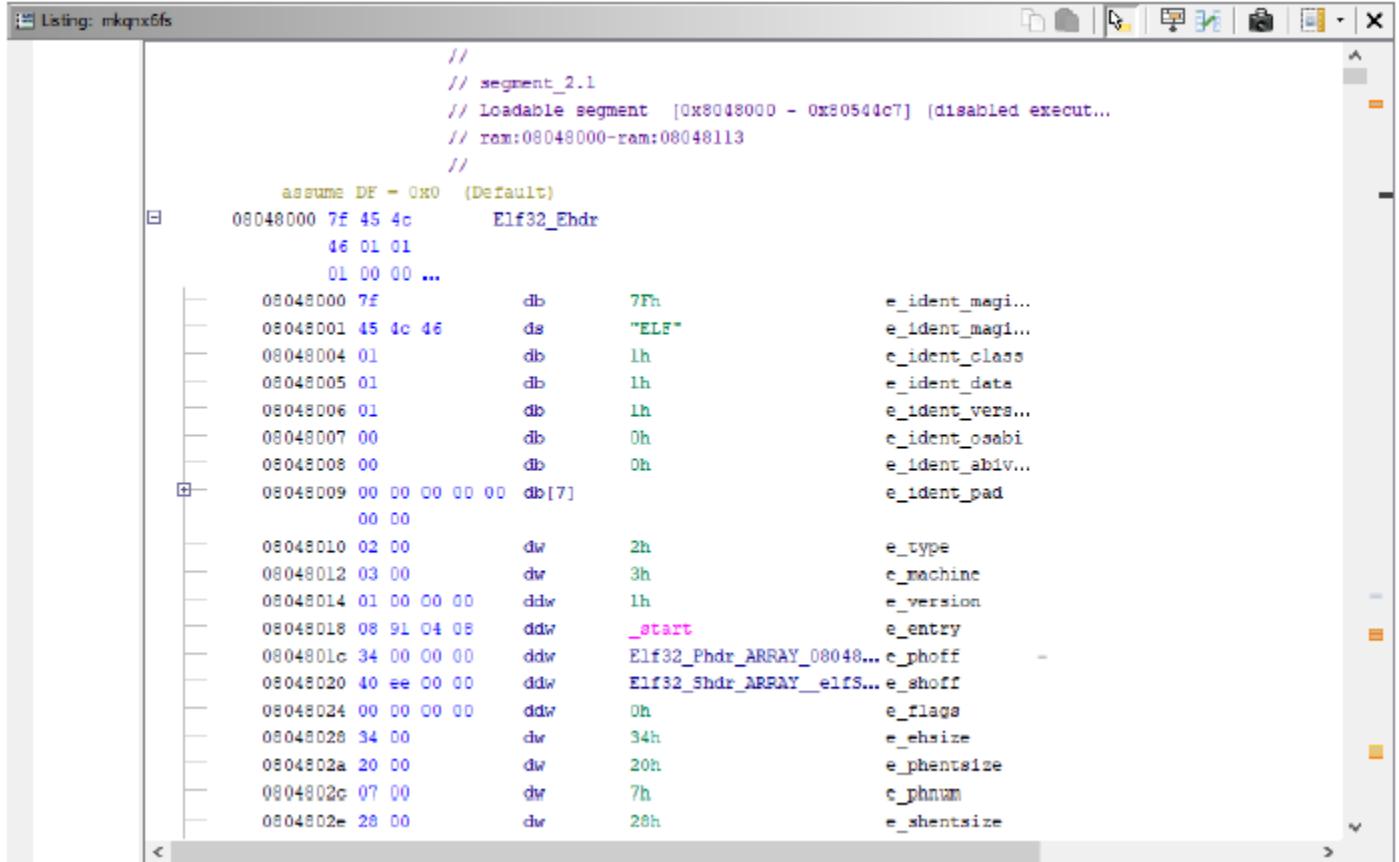
FIGURE 36 – Les différences entre les structures little endian et big endian

- Les informations relatives à la taille d'une adresse (32 bits), l'adresse minimum (08048000) et l'adresse maximum (.note.gnu.build-id : :0804815f).
- La taille de la commande « mkqnx6fs » en octets ou bytes est de 70 775 octets.
- Le nombre de blocs mémoire : 31
- Le nombre d'instructions : 180
- Le nombre de données définies : 443
- Le nombre de fonctions : 242
- Le nombre de symboles : 307
- Le nombre de type de données : 40
- Le nombre de types de données : 2
- La version de GHIDRA utilisée : 9.2.2
- ELF File Type : l'élément sur lequel on va effectuer l'opération de Reverse-Engineering est au format *executable*.

4.6 Analyse du code de « mkqnx6fs » et "fs-qnx6.so"

A l'ouverture de GHIDRA, la fenêtre principale appelée *Listing Window* affiche du code écrit en assembleur qui contient des données, des commentaires, des appels de fonctions

et qui peut également afficher des images. Elle est illustrée dans l'image ci-dessous :



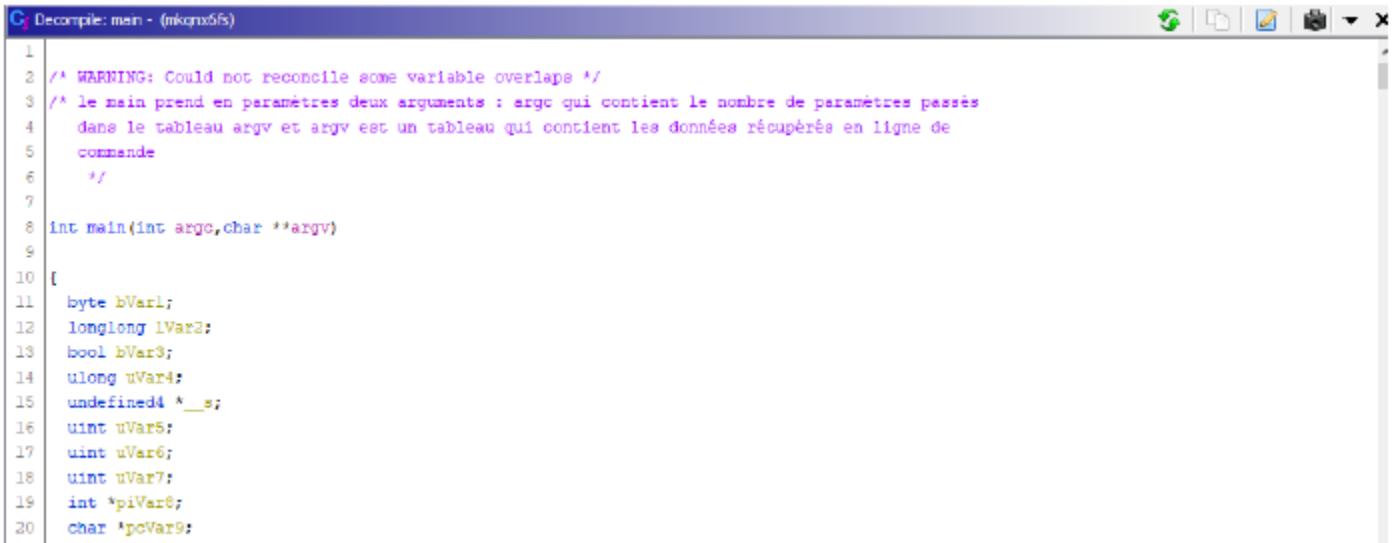
The screenshot shows the GHIDRA interface with the 'Listing' window open. The title bar says 'Listing: mkqnx6fs'. The window displays assembly code for the ELF header. The code starts with several double slashes (//) followed by comments about segments and memory regions. It then defines the ELF header structure with labels like 'Elf32_Ehdr' and various fields such as 'e_ident', 'e_type', 'e_machine', etc. The assembly code is color-coded: comments are purple, labels are blue, and registers/variables are green.

```
//  
// segment_2.1  
// Loadable segment [0x8048000 - 0x80544c7] (disabled execut...  
// ram:08048000-ram:08048113  
//  
assume DF = 0x0  (Default)  
08048000 7f 45 4c      Elf32_Ehdr  
    46 01 01  
    01 00 00 ...  
08048000 7f          db      7Fh      e_ident_mag...  
08048001 45 4c 46    ds      "ELF"  
08048004 01          db      1h       e_ident_class  
08048005 01          db      1h       e_ident_data  
08048006 01          db      1h       e_ident_vers...  
08048007 00          db      0h       e_ident_osabi  
08048008 00          db      0h       e_ident_abiv...  
08048009 00 00 00 00 00 db[7]    e_ident_pad  
    00 00  
08048010 02 00        dw      2h       e_type  
08048012 03 00        dw      3h       e_machine  
08048014 01 00 00 00  ddw     1h       e_version  
08048018 08 91 04 08  ddw     _start   e_entry  
0804801c 34 00 00 00  ddw     Elf32_Phdr_ARRAY_08048... e_phoff  
08048020 40 ee 00 00  ddw     Elf32_Shdr_ARRAY__elf$... e_shoff  
08048024 00 00 00 00  ddw     0h       e_flags  
08048028 34 00        dw      34h     e_ehsize  
0804802a 20 00        dw      20h     e_phentsize  
0804802c 07 00        dw      7h       e_phnum  
0804802e 28 00        dw      28h     e_shentsize
```

FIGURE 37 – Listing Window sur GHIDRA

La deuxième fenêtre importante est la fenêtre du décompilateur qui se situe à droite de la fenêtre des Listings vue ci-dessus. Lorsqu'on clique sur une fonction dans la fenêtre des Listings qui contient du code en assembleur, le résultat de la décompilation est affiché dans la fenêtre de décompilation et il correspond à du code en langage C. Le code en C se trouvant dans la fenêtre du décompilateur est alors plus lisible et plus compréhensible que le code en assembleur se trouvant dans la fenêtre des listings car le langage C est d'un niveau plus haut que l'assembleur , c'est-à-dire qu'il fait plus abstraction des caractéristiques techniques du matériel utilisé pour exécuter le programme, tels que les registres et drapeaux du processeur.

La fenêtre du décompilateur est illustrée ci-dessous :



The screenshot shows the GDB decompiler window titled "Décompile: main - (mkqnx6fs)". The code displayed is:

```
1  /* WARNING: Could not reconcile some variable overlaps */
2  /* le main prend en paramètres deux arguments : argc qui contient le nombre de paramètres passés
3  dans le tableau argv et argv est un tableau qui contient les données récupérées en ligne de
4  commande
5  */
6
7  int main(int argc,char **argv)
8
9  {
10    byte bVar1;
11    longlong lVar2;
12    bool bVar3;
13    ulong uVar4;
14    undefined4 *_s;
15    uint uVar5;
16    uint uVar6;
17    uint uVar7;
18    int *piVar8;
19    char *pcVar9;
```

FIGURE 38 – Decompiler Window

Sur la partie gauche de la fenêtre des listings, nous avons le menu *Program Trees* qui contient les différentes parties de l'exécutable par exemple le « .bss », « .got », etc. Il est illustré ci-dessous :

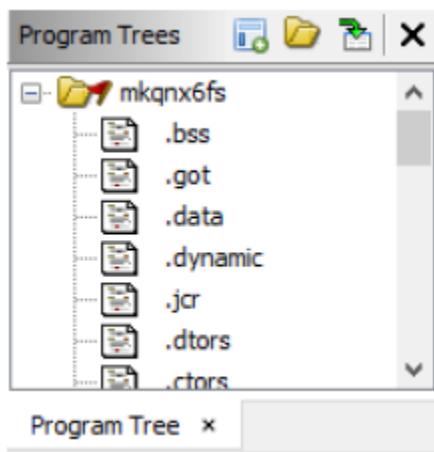


FIGURE 39 – Program Trees Menu

Sous le *Program Trees Menu*, on trouve le menu *Symbol Tree* qui contient les *imports*, *exports*, les fonctions, les labels, les classes et les namespaces trouvés en faisant l'analyse de la commande *mkqnx6fs*. Il est illustré ci-dessus :

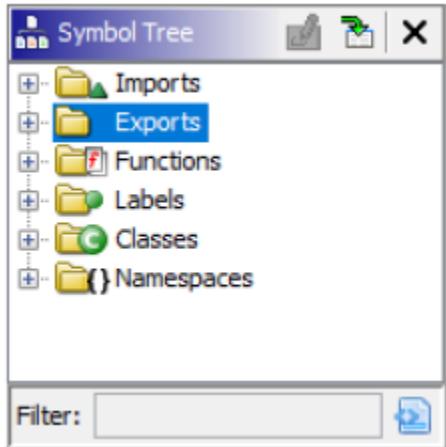


FIGURE 40 – Symbol Tree Menu

Sous le *Symbol Tree Menu*, on trouve le *Data Type Manager* où on peut gérer le type des données. Il est illustré ci-dessous :

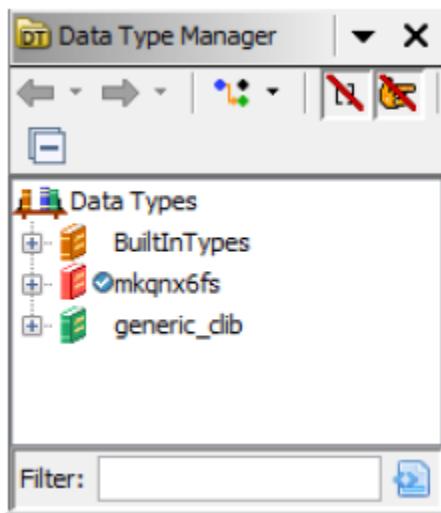


FIGURE 41 – Data Type Manager

Après avoir analysé GHIDRA, le travail va consister tout d'abord à analyser le code du main qui se trouve dans /exports. Il va falloir renommer les variables, changer les types des fonctions et des variables, ajouter des commentaires, etc.

4.7 Résultats obtenus du reverse engineering

Tout d'abord, nous avons commencé par analyser le main de la commande `mkqnx6fs` pour ensuite comprendre les différentes fonctions utilisées par ce programme. Au fur et à mesure de l'ajout des commentaires et du renommage des différentes variables, le code s'est éclairci même si cela a constitué un travail **fastidieux**. Tout d'abord, la commande `mkqnx6fs` traitent les options saisies par l'utilisateur.

```
113     percentToReserve = local_88;
114 switchD_0804af02_caseD_0:
115     local_88 = percentToReserve;
116     local_110 = (char *)argv;
117     local_10c = argv;
118             /* Si une option est trouvée, getopt() renvoie le caractère de l'option. Le
119             troisième paramètre correspond aux différentes options que l'on peut passer à
120             la commande mkqnx6fs [-Bq] [-b blocksize] [-e endian] [-g groups]
121             [-i inodes] [-n blocks] [-O options] [-o options]
122             [-r percent] [-T type] [-u uuid] host */
123     option = getopt(argc, argv, ":Bb:e:g:i:n:O:o:qr:T:u:v:");
124             /* Si toutes les options de la ligne de commande ont été analysées, getopt()
125             renvoie -1.
126             Alors, optind devient l'index du premier élément de argv qui ne soit pas
127             une option. */
128 if ((option != -1) || (optind < argc)) {
129     bVar34 = option + 1U < 0x77;
130     bVar29 = option + 1U == 0x77;
131     percentToReserve = local_88;
132     switch(option) {
133         case 0x3f:
134             /* getopt renvoie 0x3f (?) si getopt a rencontré un caractère d'option qui n'est pas
135             dans optstring */
136             fatal("unknown option '\\%c\\'");
137         case 0x3a:
138             /* getopt renvoie 0x3a (:) si il manque un argument et si le premier caractère de
139             optstring est un (':') */
140             fatal("missing argument for '\\%c\\'");
141     switchD_0804af02_caseD_42:
142             /* 0x42 option B      0x54 option -T type (desktop, runtime, media) 0x4f option -O
143             options (quiet ou close) */
144     rewriteMode = 1;
145     percentToReserve = local_88;
146 }
```

FIGURE 42 – Traitement des options de la commande `mkqnx6fs`

En analysant comment `mkqnx6fs` traite les options, nous avons pu déterminer que par défaut, la taille d'un bloc était de 1024 octets. Au minimum, la taille d'un bloc est de 512 octets.

```
|251     }
|252         /* Par defaut la taille d'un block est de 1024 */
|253         fsBlockSize = 0x400;
|254         percentToReserve = local_88;
|255     }
```

FIGURE 43 – Traitement des options de la commande mkqnx6fs

Traitement du nombre d'inodes spécifié par l'utilisateur.

```
case 0x69:
        /* option -i Set the maximum number of inodes in the filesystem. Each unique
           file or directory requires an inode. */
    piVar6 = (int *)__get_errno_ptr(local_110,local_10c);
    *piVar6 = 0;
        /* String to unsigned long (le nombre d'inodes)
           le deuxième param de strtoul , local_24 correspond à l'adresse du premier
           caractère invalide qui n'as pas pu être converti en ul
           Par exemple : 54kb local_24 pointe sur k */
    nbInodes = strtoul((char *)byteSizedBlockBuffer,local_24,10);
    if ((nbInodes != 0xffffffff) || (*piVar6 != 0x22)) {
        cVar3 = *local_24[0];
            /* Si la valeur indiqué est nbInodes suivi de 'k' ou 'K' alors il s'agit de
               kbits on fait donc un shift de 10 pour convertir la valeur en bits */
        if ((cVar3 == 'k') || (cVar3 == 'K')) {
            nbInodes = nbInodes << 10;
            local_24[0] = local_24[0] + 1;
        }
        else {
            /* SI la valeur indiquée est nbInodes suivi de 'm' ou 'M' alors il s'agit de
               mB on fait donc un shift de 20 pour convertir la valeur en bits */
            if ((cVar3 == 'm') || (cVar3 == 'M')) {
                nbInodes = nbInodes << 0x14;
                local_24[0] = local_24[0] + 1;
            }
        }
    }
```

FIGURE 44 – Traitement du nombre d'inodes spécifié

La fonction *cksblk* (probablement une diminution de *check Super Block*) utilisée par *mkqnx6fs* nous a permis de comprendre en partie la structure d'un Super Block. Elle nous a notamment permis d'obtenir le *magic number* permettant de vérifier que le système de fichiers correspond bien à QNX6. Elle nous a également permis d'identifier en partie la structure du Super Block.

```
21 sVar1 = pread64(*(int *)(param_1 + 0x248),&magicNumber,0x200,CONCAT44(offset,firstSuperBlockAddr))
22 ;
23         /* Check le numero magique contenu dans le super bloc, si = 0x22111968 alors
24             il s'agit bien d'un FS QNX6
25
26             Si 0x200 (512) correspond à la taille d'un super block( ! différent de la
27             zone du super block ! ) */
28 if (sVar1 == 0x200) {
29     isQNX6FS = (uint)(magicNumber == 0x22111968);
30     if (magicNumber == 0x22111968) {
31         uVar3 = crc32(&checksum,0x1f8);
32         uVar4 = (uint)CONCAT21((short)(local_418 >> 0x10),(char)local_418) << 8;
```

FIGURE 45 – Vérification du Super Block

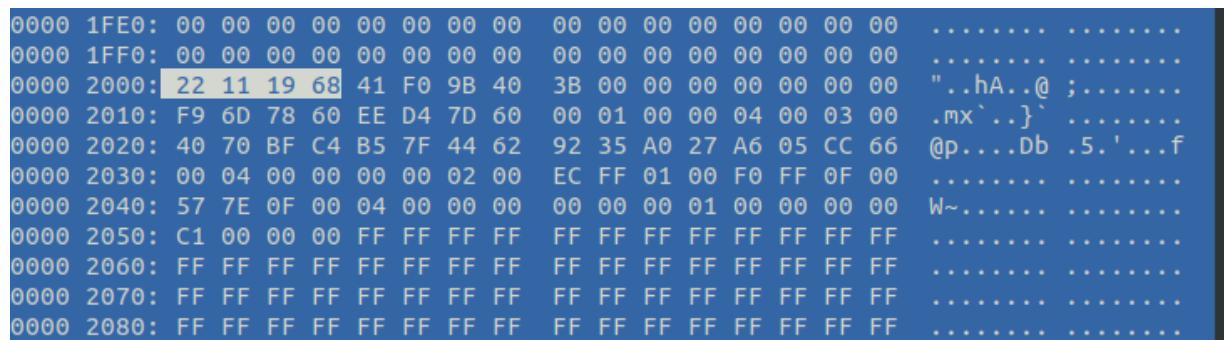
L'étude de la commande *mkqnx6fs* nous a permis d'en apprendre plus sur la structure du système de fichiers QNX6. Cependant, celle-ci s'est révélée insuffisante pour com-

prendre en détails le système de fichiers QNX6. L'étude par reverse engineering d'une telle commande est trop complexe pour des étudiants comme nous qui ne sont pas familiers avec la méthode du reverse engineering. Il a donc fallu apprendre cette méthode d'analyse depuis le départ ce qui s'est avéré très compliqué et très chronophage. Le reverse engineering complet de la commande *mkqnx6fs* et *fs-qnx6.so* aurait nécessité de meilleures compétences dans cette méthode d'analyse.

5 Étude du système de fichiers en analysant l'hexadécimal de ce dernier

Hormis le reverse engineering, l'une des méthodes qui s'est avérée payante pour comprendre le fonctionnement du système de fichiers QNX6 a été d'analyser le système de fichiers en visualisant le code hexadécimal au travers d'un logiciel comme VbinDiff. En créant un fichier comme étant un Block Device que l'on formate par la suite en système de fichiers QNX6 grâce à la commande mkqnx6fs(voir 7.2.1). Nous avons pu visualiser le code hexadécimal du système de fichiers. Cela nous a permis notamment de pouvoir valider les informations que nous avions obtenu jusqu'à présent. Dans l'exemple, le fichier servant de bloc device a une taille de 1Go.

Dans l'exemple ci-dessous, on examine le code hexadécimal du système de fichiers à partir de l'offset 0x2000. Cet offset correspond selon notre analyse par reverse engineering de mkqnx6x à l'offset du commencement du Super Block. En effet, si l'on regarde à partir de cette offset les quatre premiers octets correspondent à "22 11 19 68", on remarque qu'il s'agit du *magic number* permettant d'identifier le système de fichiers en notation *little endian*. De plus, si on regarde à partir de l'offset 0x2030, on peut lire 00 04 00 00 soit 0x400 en notation *little endian*. Cette valeur, qui vaut 1024 en décimal, correspond bien à la taille par défaut d'un bloc choisi lors de la création du système de fichiers par la commande mkqnx6fs. Cette méthode d'analyse du système de fichiers nous a aussi permis d'identifier le champs *mtime* à partir de l'offset 0x2010 noté en *little endian* ainsi que le champs *atime* à partir de l'offset 0x2014.



```
0000 1FE0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0000 1FF0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
0000 2000: 22 11 19 68 41 F0 9B 40 3B 00 00 00 00 00 00 00 "...hA...@ ;.... .
0000 2010: F9 6D 78 60 EE D4 7D 60 00 01 00 00 04 00 03 00 .mx`...}` .... .
0000 2020: 40 70 BF C4 B5 7F 44 62 92 35 A0 27 A6 05 CC 66 @p....Db .5.'....f
0000 2030: 00 04 00 00 00 00 02 00 EC FF 01 00 F0 FF 0F 00 ..... .
0000 2040: 57 7E 0F 00 04 00 00 00 00 00 00 01 00 00 00 00 W~.... .
0000 2050: C1 00 00 00 FF .....
0000 2060: FF .....
0000 2070: FF .....
0000 2080: FF .....
```

FIGURE 46 – Visualisation d'un super bloc en hexadécimal

Le logiciel vBinDiff permet notamment de souligner les différences entre deux fichiers. Cela a été très utile pour visualiser les différences entre deux versions du système de fichiers. Voici un exemple où l'on a deux versions du même système de fichiers. Dans la deuxième version, on a rajouté deux fichiers nommés fic3.txt et fic4.txt.

| | | | | |
|----------------|-------|-------------------------|-------------------------|---------------------|
| 0280 | 28C0: | 04 80 01 00 06 70 68 6F | 74 6F 73 00 00 00 00 00 |pho tos..... |
| 0280 | 28D0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 28E0: | 05 80 01 00 05 6E 6F 74 | 65 73 00 00 00 00 00 00 |not es..... |
| 0280 | 28F0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2900: | 04 00 00 00 FF 00 00 00 | 01 00 00 00 23 76 2B B1 | #v+..... |
| 0280 | 2910: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2920: | 06 80 01 00 02 6D 31 00 | 00 00 00 00 00 00 00 00 |m1..... |
| 0280 | 2930: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2940: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2950: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2960: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2970: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2980: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2990: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 29A0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| imgqnx8 | | | | |
| 0280 | 2820: | 01 00 00 00 02 2E 2E 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2830: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2840: | 02 00 00 00 05 2E 62 6F | 6F 74 00 00 00 00 00 00 |bo ot..... |
| 0280 | 2850: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2860: | 03 00 00 00 05 6E 76 66 | 69 63 00 00 00 00 00 00 |nvf ic..... |
| 0280 | 2870: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2880: | 01 80 01 00 08 64 6F 73 | 73 69 65 72 31 00 00 00 |dos sier1..... |
| 0280 | 2890: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 28A0: | 03 80 01 00 08 64 6F 73 | 73 69 65 72 32 00 00 00 |dos sier2..... |
| 0280 | 28B0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 28C0: | 04 80 01 00 06 70 68 6F | 74 6F 73 00 00 00 00 00 |pho tos..... |
| 0280 | 28D0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 28E0: | 05 80 01 00 05 6E 6F 74 | 65 73 00 00 00 00 00 00 |not es..... |
| 0280 | 28F0: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2900: | 04 00 00 00 FF 00 00 00 | 01 00 00 00 23 76 2B B1 | #v+..... |
| 0280 | 2910: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2920: | 06 80 01 00 02 6D 31 00 | 00 00 00 00 00 00 00 00 |m1..... |
| 0280 | 2930: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2940: | 05 00 00 00 08 66 69 63 | 33 2E 74 78 74 00 00 00 |fic 3.txt..... |
| 0280 | 2950: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |
| 0280 | 2960: | 06 00 00 00 08 66 69 63 | 34 2E 74 78 74 00 00 00 |fic 4.txt..... |
| 0280 | 2970: | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | |

FIGURE 47 – Différence entre deux versions du système de fichiers

FIGURE 48 – Différence entre deux versions du système de fichiers

Ci-dessus, un inode qui a été modifié entre les deux versions du système de fichiers. Si on s'intéresse à la structure de l'inode (voir chapitre Inode), on remarque que les octets modifiés (en rouge) correspondent au *atime* (access time) qui correspond à la date/heure d'accès à la ressource liée à l'inode en question. En effet, dans la structure d'un inode, le *atime* est un *unisigned int* sur 4 octets du 24ème au 28ème octet à partir du début de l'inode et se lit en mode *Little endian*. Dans la première version, on a donc une date d'accès de 0x607DDB39, ce qui donne :

Convert epoch to human-readable date and vice versa

[Timestamp to Human date](#)[\[batch convert\]](#)

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds.

Assuming that this timestamp is in **seconds**:

GMT: Monday 19 April 2021 17:17:45

Your time zone: lundi 19 avril 2021 19:17:45 [GMT+02:00 DST](#)

Relative: 14 days ago

FIGURE 49 – Date d'accès à la ressource liée à l'inode dans la version antérieure du système de fichier

Dans la deuxième version, l'access time vaut 0x607DD4EB. Si on convertit cette valeur en temps, on remarque que la ressource liée à l'inode a été accédée environ deux heures après le premier accès. Entre la première et la deuxième version du système de fichiers, la ressource liée à cette inode a donc été consultée.

[Timestamp to Human date](#)[\[batch convert\]](#)

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds

Assuming that this timestamp is in **seconds**:

GMT: Monday 19 April 2021 19:07:23

Your time zone: lundi 19 avril 2021 21:07:23 [GMT+02:00 DST](#)

Relative: 14 days ago

FIGURE 50 – Date d'accès à la ressource liée à l'inode dans la version postérieure du système de fichier

Toujours avec deux versions du même système de fichiers, avec la deuxième version postérieure à la première, on remarque des modifications au niveau du Super Block. Les quatre premiers octets en rouge correspondent au *checksum*, il a donc été modifié. Le numéro de série permettant d'identifier le Super Block actif a également été modifié. (pour plus d'informations, voir le chapitre sur les Super Blocks).

Dans la première version, la partie correspondante au nombre d'inode libre est égale à 0x1FFEC. Dans la version postérieure, ce même champs vaut 0x1FFEB. On a donc

un inode de libre en moins dans la version postérieure du système de fichiers. Il y a donc probablement un fichier ou un dossier en plus qui a été créé entre la première et la deuxième version. En effet, si on continue à chercher les différences entre ces deux versions du système de fichiers, on remarque qu'un inode a bien été ajouté :

FIGURE 51 – Différence du Super Block entre deux versions du même système de fichier

6 Partitionnement

6.1 Définition d'une partition

Parmi les supports de stockage les plus utilisés, nous citons : les supports de stockage mobiles (clé USB), les disques durs, les cartes mémoire, les CD et DVD et les serveurs. Le choix du support de stockage est toujours défini en fonction des besoins. Les critères de sélection sont : la capacité de stockage, le degré de sûreté et la mobilité. Chaque support de stockage possède plusieurs sections appelées partitions. L'organisation des supports de stockage en partitions permet au système d'exploitation de mieux gérer les informations et d'optimiser les opérations d'entrée/sortie.

Il existe une partition spéciale appelée « partition d'amorçage » qui contient un programme qui est exécuté à l'initialisation du matériel pour continuer le processus de démarrage.

6.2 Les différents types de partitionnement

Il existe deux types de partitionnement, tous les deux conçus par la société Intel : le partitionnement de type MBR (Master Boot Record) et le partitionnement de type GPT (GUID Partition Table). Le partitionnement de type MBR a été conçu en 1983 et a été conçu pour les ordinateurs dont l'espace de stockage ne dépasse pas 2 To et est mieux adapté aux ordinateurs personnels (PC). Contrairement au partitionnement MBR, le partitionnement GPT a été conçu en 2002 et est un partitionnement qui est utilisé sur les machines dont la capacité de stockage est supérieure à 2 To. Le partitionnement MBR étant limité à l'adressage à 32 bits, les ordinateurs actuels font pour la plupart du partitionnement de type GPT. Dans les prochaines années, la transition du partitionnement MBR vers le partitionnement GPT verra la disparition du premier type de partitionnement.

6.3 Le partitionnement sous les différents systèmes d'exploitation

Chaque système d'exploitation a une manière différente de partitionner les supports de stockage.

Dans notre cas, où nous étudions le système d'exploitation QNX, comme nous le verrons plus tard, une partition QNX est composée du numéro du disque dur (par exemple /dev/hd0 est le premier disque dur, /dev/hd1 est le deuxième disque dur, etc.) suivi par le nom de la partition (par exemple "t6" pour la partition DOS, "t79" pour la partition QNX, etc.).

Pour les systèmes d'exploitation de type Unix ou Gnu/Linux, les partitions sont désignées par la notation sdXN où X désigne le nom du support de stockage et N désigne le numéro de la partition. Par exemple, la notation "sdb6" fait référence à la sixième partition du support de stockage.

Sous le système d'exploitation Windows, les partitions sont désignées par les disques C, D, E où le disque C (disque dur interne principal) stockent les applications installées, les

profils utilisateurs et les applications systèmes et dossiers propres au système d'exploitation Windows. Le disque D fait référence aux données (data). Il permet de stocker des photos, vidéos, musiques et des documents qui ne sont pas des programmes systèmes. Le disque E permet lui de faire référence aux supports de stockage externe détectés par l'ordinateur. Tout comme le disque D, le disque E permet de stocker divers types de fichiers, images, vidéos, musiques, documents, etc.

Sous le système d'exploitation développé par la société Apple : MacOS, les partitions ont un nom sous une forme bien précise : diskNsM où N désigne le numéro du support de stockage et M désigne le numéro de la partition. Par exemple, la notation "disk5s6" permet de référencer la sixième partition du cinquième support de stockage.

6.4 La table de partitionnement Master Boot Record (MBR)

Avant d'expliquer plus en détails les différences entre le partitionnement MBR et le partitionnement GTP, nous devons donner quelques définitions concernant le BIOS (Basic Input Output System) et l'UEFI (Unified Extensible Firmware Interface). Le BIOS est un programme qui est disponible sur la carte mère de l'ordinateur et il s'exécute au démarrage du matériel pour l'initialiser. L'UEFI est aussi un programme informatique qui représente une version du BIOS. La différence entre le BIOS et l'UEFI du point de vue du partitionnement est que l'UEFI utilise une table de partitionnement de type GPT (GUID Partition Table).

Dans chaque support de stockage, la première zone du support de stockage est appelée table de partitionnement MBR (Master Boot Record). Cette zone contient une table appelée « table de partitions » qui contient des informations sur les partitions logiques et un programme appelé « code d'amorçage » et qui est exécuté en même temps que le BIOS au démarrage de l'ordinateur. On rappelle que le MBR n'est valable que pour les machines dont l'espace de stockage ne dépasse pas 2 To. A chaque démarrage de l'ordinateur, le BIOS charge la table de partitionnement MBR qui représente les 512 premiers octets du disque.

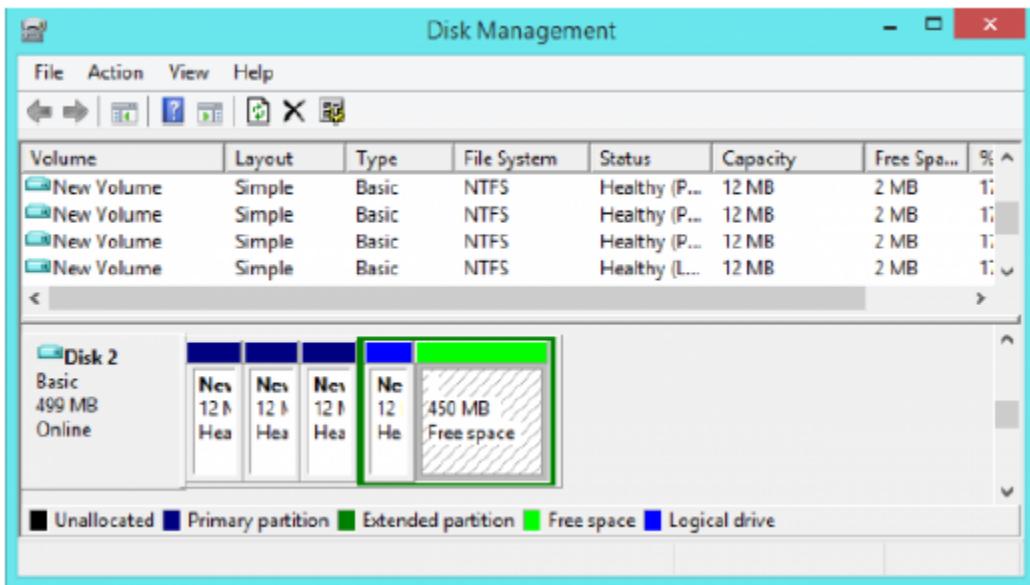


FIGURE 52 – Partitionnement MBR

Le standard GPT permet de surmonter les limitations du partitionnement MBR. Il permet de décrire la table de partitionnement d'un disque et exécute un code en même temps que l'UEFI (qui remplace le BIOS). Chaque partition possède un unique identifiant (GUID – Global Unique Identifier). Pour le partitionnement GPT, les disques n'ont pas de limite d'espace de stockage et ils peuvent gérer un grand nombre de partitions qui est déterminé par le système d'exploitation. Le système d'exploitation Windows permet par exemple de gérer jusqu'à 128 partitions sur un support de stockage de type GPT.

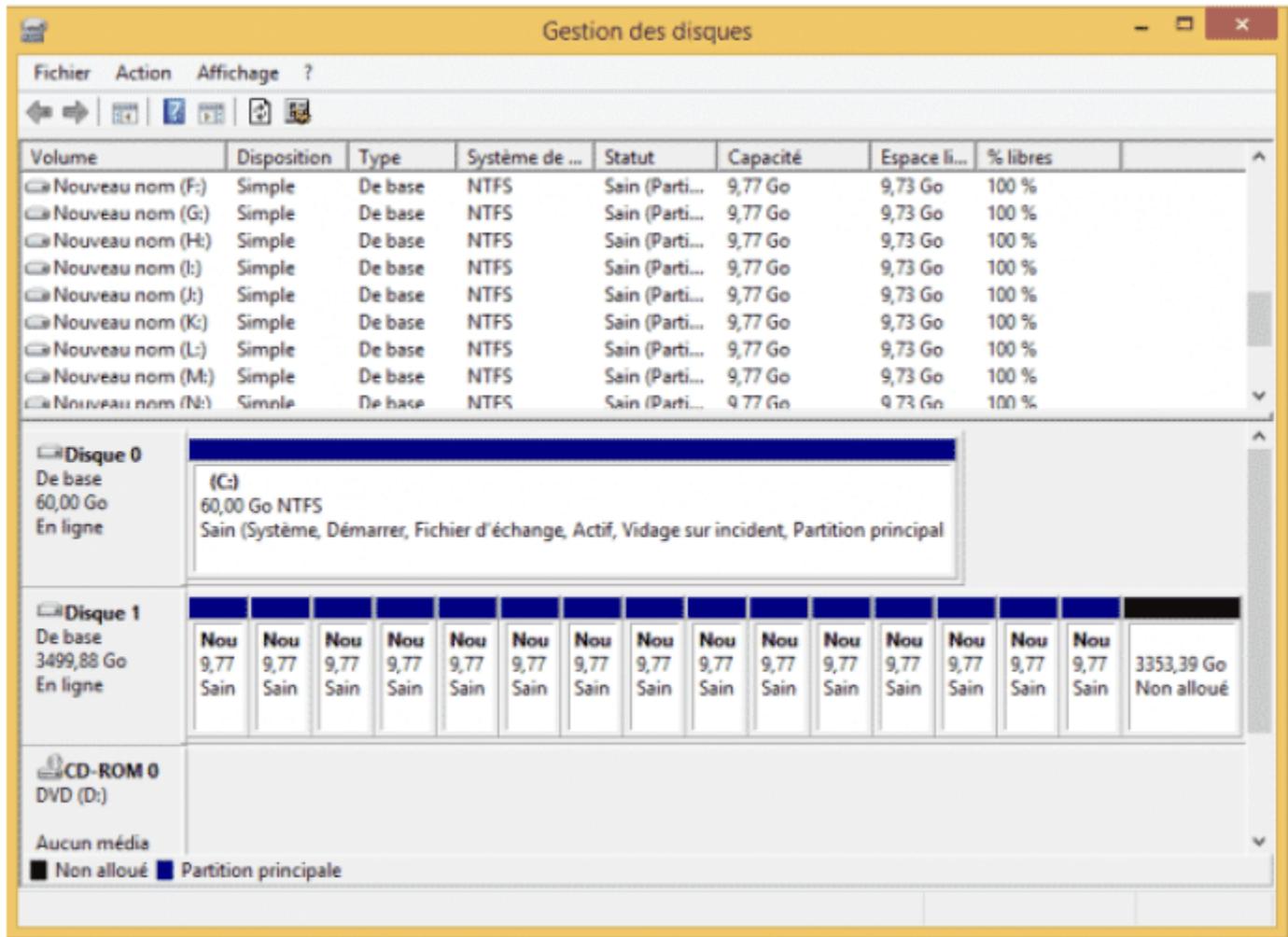


FIGURE 53 – Partitionnement GPT

6.5 Partitionnement de QNX

Le système d'exploitation QNX utilise le partitionnement du disque c'est-à-dire qu'il permet à plusieurs systèmes de fichiers différents de partager le même disque. Chaque partition du système d'exploitation QNX est représentée sous la forme d'un fichier de type *Block-Special* où chaque type de partitions est ajouté au nom du disque qui lui correspond. Comme nous l'expliquerons prochainement, un fichier de type *Block-Special* ou *Block-Special File* est un fichier qui fait référence et qui contient des informations sur un disque physique *Physical Disk*.

6.5.1 Quelques exemples de partitions présentes dans le système d'exploitation QNX :

a) /dev/hd0 : Premier disque dur « First hard disk».

b) /dev/hd0t6 : « Dos partition on the first hard disk », La partition Dos du premier disque dur. Dans cet exemple, l'élément qui nous permet d'affirmer que le disque « hd0t6 » fait référence à la partition du Dos est le type de partition qui est présent dans le nom du disque : 6 permet d'identifier la partition Dos.

c) /dev/hd0t79 : Partition QNX dans le premier disque dur. L'élément qui permet d'affirmer que le disque fait référence à la partition QNX est la présence du type de la partition dans le nom du disque (dans ce cas 79).

d) /dev/hd1 : Deuxième disque dur « Second Hard Disk ».

e) /dev/hd1t79 : « QNX partition on the second hard disk », la partition QNX dans le second disque dur. Dans le nom du disque faisant référence à la partition QNX, « hd1 » permet de référencer le deuxième disque dur et « t79 » nous indique que c'est une partition QNX qui est référencée dans le second disque dur.

Nous allons résumer les différents types de partitions présents sous le système d'exploitation QNX ainsi que le numéros permettant de les identifier dans le tableau suivant :

| Type | Filesystem |
|------|--|
| 1 | DOS (12-bit FAT) |
| 4 | DOS (16-bit FAT ; partitions < 32M) |
| 5 | DOS Extended Partition (enumerated but not presented) |
| 6 | DOS 4.0 (16-bit FAT ; partitions >= 32M) |
| 7 | OS/2 HPFS |
| 7 | Previous QNX version 2 (pre-1998) |
| 7 | Windows NT |
| 8 | QNX 1.x and 2.x (« qny ») |
| 9 | QNX 1.x and 2.x (« qnz ») |
| 11 | DOS 32-bit FAT ; partitions up to 2047G |
| 12 | Same as Type 11, but uses Logical Block Address Int 13h extensions |
| 14 | Same as Type 6, but uses Logical Block Address Int 13h extensions |
| 15 | Same as Type 5, but uses Logical Block Address Int 13h extensions |
| 77 | QNX POSIX partition (secondary) |
| 78 | QNX POSIX (secondary) |
| 79 | QNX POSIX partition |
| 99 | UNIX |
| 131 | Linux (Ext2) |
| 175 | Apple Macintosh HFS or HFS Plus |
| 177 | QNX Power-Safe POSIX partition (secondary) |
| 178 | QNX Power-Safe POSIX partition (secondary) |
| 179 | QNX Power-Safe POSIX partition |

7 Architecture du système de fichiers de QNX6

7.1 Définition d'un Block Device ou Block Special File

Sous le système d'exploitation UNIX, il existe deux types de fichiers matériels (Device files) : les Character Special Files et les Block Special Files (ou Block Devices).

Les Character Special Files fournissent un accès direct non bufférisé au hardware (matériel) alors que les Block Devices offrent un accès direct bufférisé au hardware (matériel). Utilisé dans le système d'exploitation UNIX et à la différence d'un fichier normal, un fichier de type Block Device ou Block Special File est un fichier qui fait référence et qui fournit un accès au Device (matériel). Il offre aussi un degré de sécurité en ne divulguant pas les caractéristiques matérielles (Hardware) du Device. Ce type de fichiers apparaît dans le système de fichiers comme étant un fichier normal mais il est le plus souvent utilisé par des programmes informatiques qui souhaitent interagir directement avec le matériel (Device). Les Block Devices font généralement références à des disques durs (Disk Devices) pour lesquels le kernel fournit de la mise en cache (caching). La mise en cache rend le fichier presque inutilisable car elle empêche les applications ou les programmes informatique de connaître le contenu du disque (Disk) à un instant précis. Les Block Devices font référence aux différents fichiers de périphériques (Device Files) qui se trouvent dans le répertoire de fichiers : /dev.

Un secteur est la plus petite unité addressable qui compose un Block Device qui dans la plupart des cas a une taille de 512 octets.

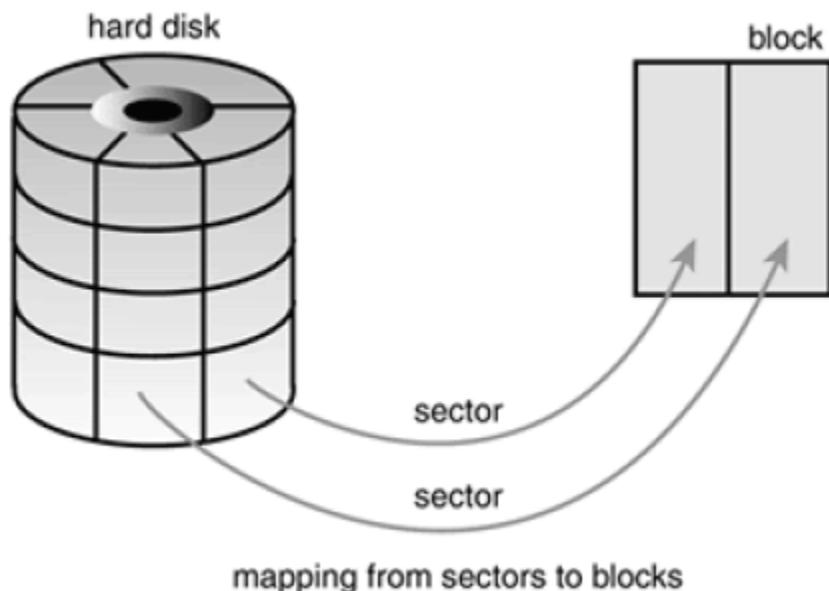


FIGURE 54 – Structure d'un Block Device

7.1.1 Interprétation d'un fichier normal comme étant un fichier de type Block Device

a) Nous allons dans un premier temps créer un fichier de taille 1GB appelé «diskimage» en utilisant la commande «dd» :

```
# dd if=/dev/zero of=/root/diskimage bs=1048576 count=1024
1024+0 records in
1024+0 records out
#
```

FIGURE 55 – Crédit d'un fichier avec la commande dd

b) Il faut ensuite formater le fichier pour lui donner une structure de système de fichiers QNX6 en utilisant la commande «mkqnx6fs» :

```
# mkqnx6fs /root/diskimage
Format fs=qnx6: 1048560 blocks, 131072 inodes, 4 groups
#
```

FIGURE 56 – Formatage du fichier permettant de lui donner une structure de système de fichiers QNX6

Le système de fichiers obtenu après le formatage a bel et bien un format de type QNX6. Il est composé de 1048560 blocks, 131072 inodes et 4 groups.

c) Création d'un répertoire servant de point de montage en utilisant la commande «mkdir» :

```
# mkdir /mnt/mymountpoint
```

FIGURE 57 – Crédit d'un répertoire servant de point de montage

d) On monte le système de fichiers créé sur le point de montage grâce à la commande «mount» :

```
# mount -t qnx6 /root/diskimage /mnt/mymountpoint
```

FIGURE 58 – Montage du système de fichiers créé sur le point de montage

e) On vérifie que tout a bien fonctionné :

```
#mount
/dev/hd0179 on / type qnx6
/root/diskimage on /mnt/Mymountpoint type qnx6"
```

7.2 Blocs

L'espace d'un périphérique QNX est divisé en blocs de taille fixe, déterminée au moment de la création du système de fichiers (avec la commande `mkqnx6fs` par exemple). La taille d'un bloc peut être égale à 512, 1024, 2048 ou 4096 octets. Puisque les pointeurs pointant sur les blocs ont une taille de 32 bits. La taille maximale adressable dans un système de fichiers QNX6 est $2^{32} * 4096$ octets soit 16384 Go.

7.3 Super Block

L'élément le plus important dans le système de fichiers QNX6 est le Super Block. C'est le point d'entrée permettant de récupérer les données. Il possède toutes les informations globales sur le système de fichiers. Dans le système de fichiers QNX6, le premier Super Block se situe à l'offset 0x2000. Avant cet offset, il s'agit de la partie réservée au Boot Block. Dans le système de fichiers QNX6, les Super Blocks ont une taille de 0x1000 octets. Ci-dessous, les informations que l'on peut récupérer du super block en lisant à partir de l'offset 0x2000 :

| Structure d'un Super Block | | | | |
|----------------------------|---|---|------------|--------------------|
| Offset : 0x2000 | Indice en octets | Commentaire | Endianness | Type |
| | 0-> 4 : Numéro Magique | Numéro Magique permettant d'identifier le système de fichier QNX6 : 0x68191122, QNX4 : 0x002f | Little | Unsigned int |
| | 4-> 8 : Checksum | | Big | Unsigned int |
| | 8-> 16 : Numéro de série | Permet d'identifier le SB actif | Little | Unsigned long long |
| | 16-> 20 : Change Time | | Little | Unsigned int |
| | 20-> 24 : Access Time | | Little | Unsigned int |
| | 24-> 28 : Flags | | Little | Unsigned int |
| | 28-> 30 : Version1 | | Little | Unsigned short |
| | 30-> 32 : Version2 | | Little | Unsigned short |
| | 32-> 48 : Volume ID | | Little | Unsigned char X 16 |
| | 48-> 52 : Block Size | | Little | Unsigned int |
| | 52-> 56 : Nombre d'inodes | Nombre d'inodes total | Little | Unsigned int |
| | 56-> 60 : Nombre d'inodes libres | | Little | Unsigned int |
| | 60-> 64 : Nombre de blocks | Nombre de blocs total | Little | Unsigned int |
| | 64-> 68 : Nombre de blocks libres | | Little | Unsigned int |
| | 68-> 72 : Alloc Group | | Little | Unsigned int |
| | 72-> 152 : Root Node | Contient le Root node constitué de 16 pointeurs qui permettent d'obtenir tous les inodes | | |
| | 232-> 312 : Long File Node | Les noms longs sont gérés différemment | | |

FIGURE 59 – Structure d'un Super Block

Le Super Block nous permet notamment d'identifier le type du système de fichiers à partir du numéro magique ou encore la taille d'un bloc dans le système de fichiers. Cette dernière information est extrêmement importante puisqu'elle est indispensable pour calculer les adresses des blocs. Par défaut, la taille d'un bloc est de 1024 octets. Le nombre de blocs dans le système de fichiers ainsi que la taille d'un bloc vont nous permettre d'obtenir l'adresse d'un deuxième Super Block. Voici la formule permettant d'obtenir l'adresse du second Super Block :

TAILLE BOOT BLOCK (0x2000) + ZONE SUPER BLOCK (0x1000) + (BLOCKSIZE X NOMBRE DE BLOCS)

Le second Super Block possède exactement la même structure que le premier. Après avoir obtenu les deux Super Block, nous pouvons identifier le Super Block "Actif" également appelé "Working Super Block" et le super Block "Stable" ou BackUp Super Block. Pour identifier lequel des deux Super Block correspond au Super Block Actif, il suffit de comparer les numéros de série des deux super blocs : celui ayant le numéro de série le plus élevé est le Super Block actif. (En pratique, le numéro du Super Block actif est égal au numéro du Backup Super Block + 1).

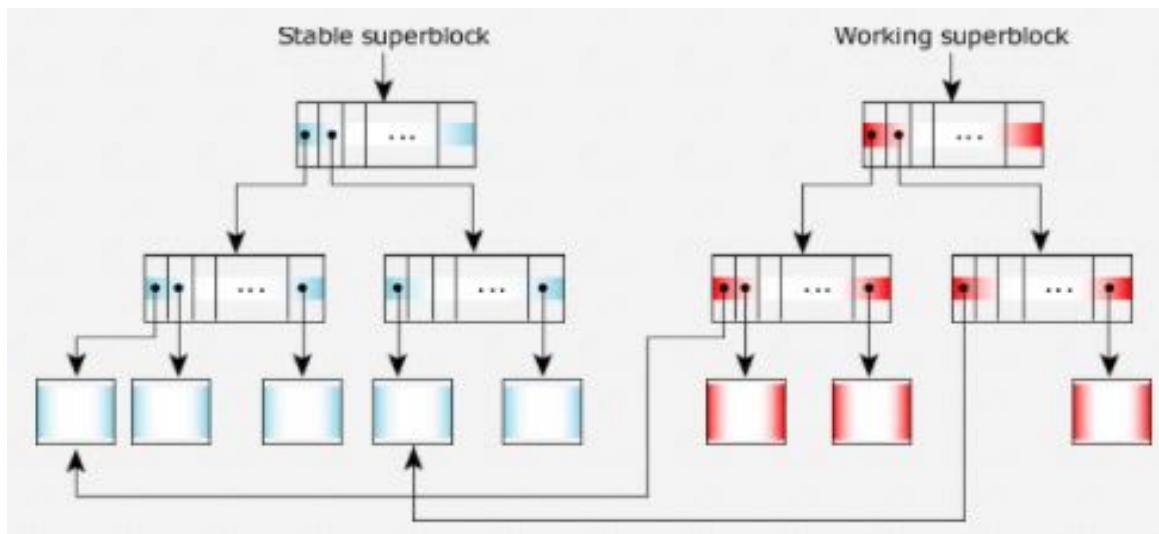


FIGURE 60 – Schéma des Super Blocks :

https://developer.blackberry.com/playbook/native/documentation/com.qnx.doc.neutrino.sys_arch/topic/fsys_cow_filesystem.html

Si certaines données sont modifiées, elles sont écrites dans un ou plusieurs blocs inutilisés et les données originales restent inchangées. La liste des pointeurs sur les blocs est modifiée pour faire référence aux nouveaux blocs utilisés. Le système de fichiers met ensuite à jour l'inode pour faire référence aux nouveaux pointeurs. Lorsque l'opération est terminée, les données d'origine et les pointeurs qui pointent vers elles restent intacts. Cependant, un nouvel ensemble de blocs, de pointeurs et d'inode apparaît pour les données modifiées. **Cela nous permettra notamment, dans l'ingest module, de récupérer les données supprimées**

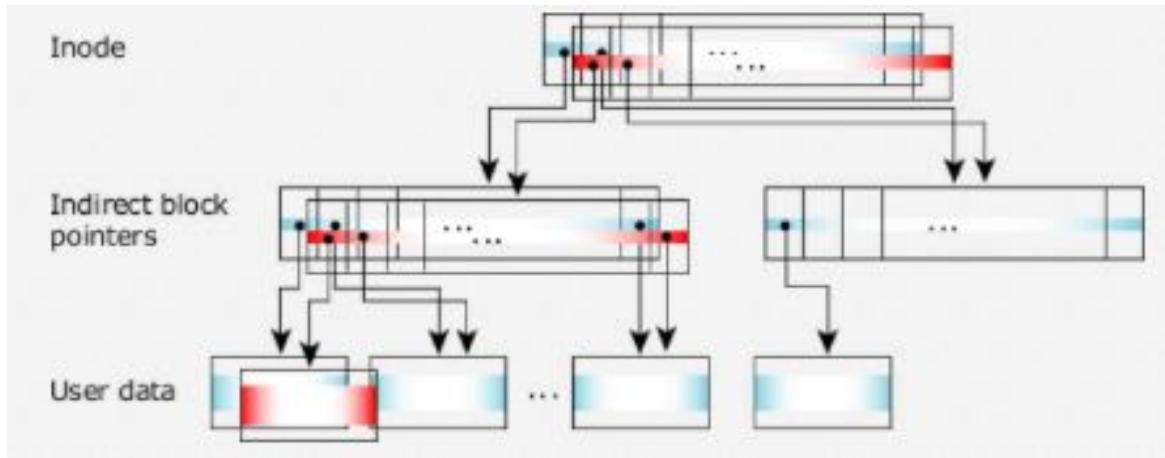


FIGURE 61 – Schéma des Super Blocks :

https://developer.blackberry.com/playbook/native/documentation/com.qnx.doc.neutrino.sys_arch/topic/fsys_cow_filesystem.html

Le Super Block Actif ou "Working Super Block" contient toutes les données modifiées à un instant T. Il s'agit de la version courante qui contient les données que pourrait voir un utilisateur qui montrait (avec `mount -t qnx6...`) le périphérique sur une machine QNX. Quand un utilisateur accède aux données sur sa machine QNX, il voit donc la version du Working Super Block. Le BackUp Super Block ou "Stable Super Block" contient quant à lui une version antérieure stable pas encore modifiée des données.

Un snapshot est une sauvegarde de l'état d'un système à un instant donné. Dans le système de fichiers QNX6, voici comment la prise d'un snapshot est effectuée :

- Blocage du système de fichiers QNX6 dans un état stable afin d'empêcher toute activité
- Écriture de tous les blocs copiés et forçage des données à être synchronisé
- Reconstruction du super bloc, enregistrement du nouvel emplacement des inodes, **incrémentation du numéro de série**
- Calcul du checksum
- Switch entre la vue "Active" et la vue "Stable". Les anciennes versions des blocs occupés sont libérées et peuvent de nouveau être utilisées.

Voici le code hexadécimal d'un Super Block. On retrouve la structure du Super Block telle qu'elle est décrite précédemment. On peut notamment vérifier visuellement un certain nombre de données. En effet, on remarque facilement que le "Magic Number" correspond bien à celui d'un système de fichiers QNX6. On constate que la date de création du système de fichiers vaut 0X608BCCC7 ce qui donne le 30/05/2021 à 9h24 après conversion en date et en heure. La valeur en hexadécimal correspond au nombre de secondes écoulées depuis EPOCH, ce qui représente une date de référence dans les systèmes UNIX.

FIGURE 62 – Visualisation d'un Super Block en hexadecimal

On apprend également en visualisant le Super Block que la taille de bloc utilisée est de 1024 octets. Le nombre de blocs total est de 0xfffff0 ce qui donne 1 048 560 blocs.

Sachant qu'un bloc a une taille de 1024 octets, l'espace utilisé par tous les blocs est de 1073725440 octets. A cela, si on ajoute la zone du Boot Block (0x2000 octets) ainsi que les zones des Super Blocks (0x1000 octets par Super Block), on obtient une taille totale de $1073725440 + 8192 + 4096 + 4096 = 1\ 073\ 741\ 824$ octets soit 1024 Mo (1 Go).

Cela concorde bien avec le fait que lors de la création du fichier servant de block device, nous avons défini la taille de ce dernier à 1 Go.

7.4 Inodes

Chaque répertoire et fichier du système de fichiers est représenté par un inode. La structure d'un inode contient des pointeurs vers des blocs du système de fichiers qui contiennent les données contenues dans l'objet (le dossier ou le fichier) et toutes les métadonnées relatives à un objet, à l'exception des noms longs de plus de 27 caractères qui sont gérés différemment. Les métadonnées comprennent notamment, la date de création, la date de modification, le UID, la taille de l'objet et son type (fichier, répertoire, supprimé). Un inode est composé de 16 pointeurs vers 16 blocs contenant les données de l'objet ou sur des blocs de pointeurs intermédiaires dans le cas d'objets de grandes tailles. Dans le cas d'un dossier, ces données correspondent aux **ID des inodes** représentant les sous-dossiers ou fichiers contenus dans le dossier. Les noms des objets contenus dans le dossier sont aussi présents. Dans le cas où 16 blocs de données ne seraient pas suffisants pour adresser toutes les données de l'objet, le système de fichiers utilise un adressage indirect sous la forme d'un arbre de la même forme que le Root Node. Chaque inode possède un Identifiant, le répertoire racine aura pour identifiant 1.

Cela nous permettra dans l'ingest module de reconstruire l'arborescence à partir du

répertoire racine.

La liste de tous les inodes qui représentent les objets peut être construite à partir des **Root Node** se trouvant dans les Super Blocks.

| Root Node | | | |
|------------------|---|------------|------------------------|
| Indice en octets | | Endianness | Type |
| 0 -> 8 | Taille (unsigned long long) | Little | 1 X unsigned long long |
| 8->72 | 16 pointeurs sur d'autres inodes intermédiaires | Little | 16 X unisgned int |
| 72->73 | Level de l'inode | Little | 1 X unsigned char |
| 73->74 | Mode | Little | 1 X unsigned char |
| 74->80 | Reservé | Little | 6 X unsigned char |
| | | | |

FIGURE 63 – Structure du Root Node

Les 16 pointeurs se trouvant dans le Root Node ne pointent pas forcément directement sur des inodes représentant un fichier ou un dossier. En effet, si c'était le cas, il pourrait donc y avoir que 16 objets dans le système de fichiers. Chaque pointeur pointe en réalité **sur des blocs de pointeurs intermédiaires**. Pour savoir combien de niveaux de blocs de pointeurs séparent les 16 pointeurs du Root Node et les inodes, on utilise le champ "level" du Root Node. Pour calculer les adresses à partir d'un pointeur, il suffit d'appliquer la formule suivante :

PTR (indice d'un bloc) X BLOCKSIZE + OFFSET où l'offset correspond à la fin du premier Super Block. On rappelle que la fin du premier Super Block est égale à la taille du Boot Block (0x2000) + la zone du Super Block (0x1000) donc 0x3000 dans le cas où il n'y a rien avant le Boot block.

A partir des 16 pointeurs se trouvant dans le Root Node, on lit dans les 16 blocs contenant des pointeurs intermédiaires. Puisqu'un pointeur est de 4 octets, il y a donc BLOCKSIZE/4 pointeurs par bloc. Les 16 pointeurs font donc référence à 16^* (BLOCKSIZE/4) pointeurs. Dans le cas le plus courant où la taille d'un bloc est de 1024, cela fait $256 * 16 = 4096$ pointeurs. Cela peut ne pas être suffisant s'il y a plus d'inodes. Il se peut donc que ces pointeurs fassent référence à leur tour, à d'autres blocs de pointeurs. Si c'est le cas cela fait 1 048 576 pointeurs pour un système de fichiers utilisant une taille de bloc de 1024 octets. On pourrait ensuite continuer...

Voici le schéma récapitulatif montrant comment retrouver les inodes à partir d'un Root Node :

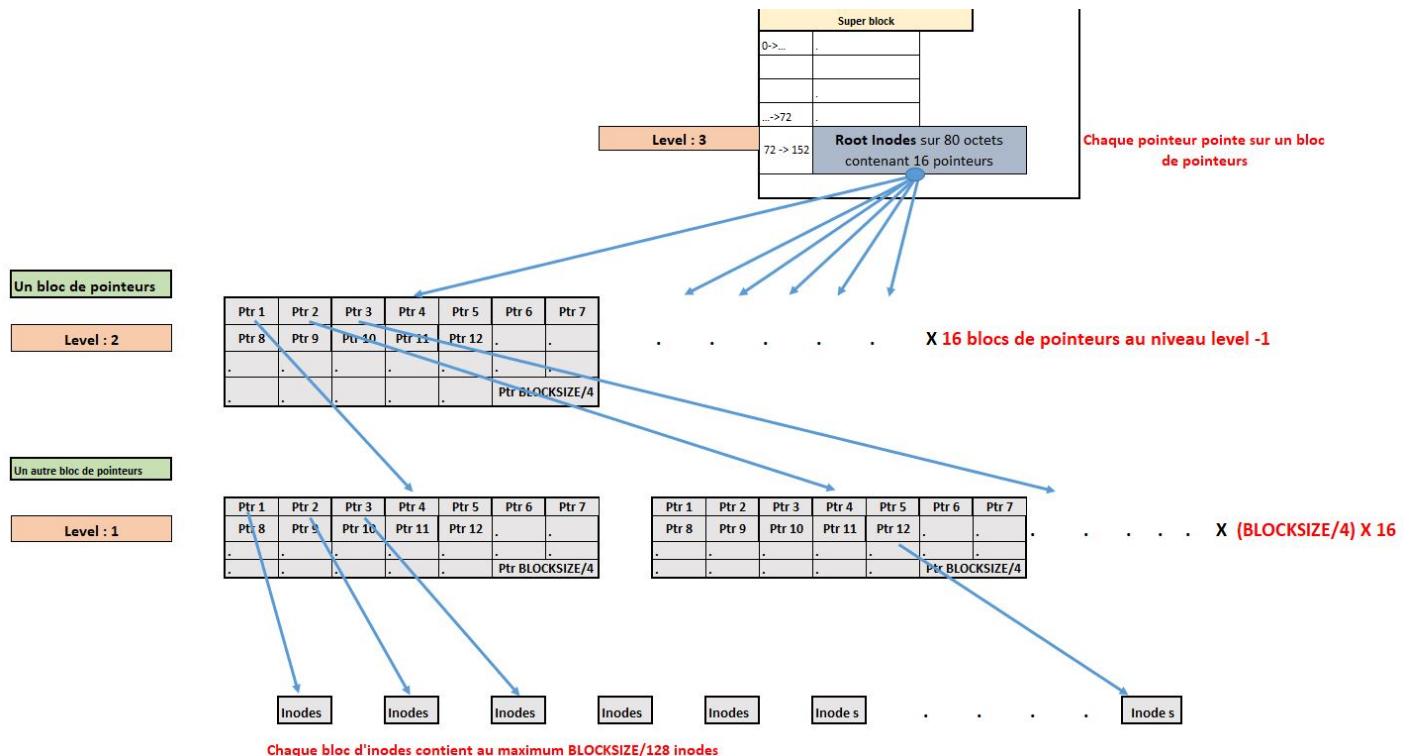


FIGURE 64 – Obtention des inodes à partir du Root Node

Pour déterminer le moment durant lequel les pointeurs ne pointent plus sur d'autres blocs de pointeurs mais bel et bien sur des blocs d'inode, on utilise le "level" ou niveau du Root Node de départ. Si le Root Node est de niveau 2, alors les blocs de pointeurs intermédiaires obtenus à partir des 16 pointeurs du Root Node seront de niveau 1. On décrémente donc le level pour chaque niveau de blocs de pointeurs jusqu'à arriver à 0. Une fois arrivé à 0, les pointeurs ne pointent plus sur d'autres blocs de pointeurs mais sur des blocs d'inode. Un inode faisant 128 octets, il y a donc $BLOCKSIZE/128$ inodes au maximum par bloc. Le niveau dans le Root Node dépend donc du nombre d'inode et donc, d'objets dans le système de fichiers : Plus il y a de dossiers et de fichiers, plus il sera nécessaire d'avoir un haut "level" pour obtenir plus d'inode.

Enfin, voici la structure d'un inode représentant un fichier ou un répertoire :

| Structure d'un inode | | | |
|----------------------|--|-----------|--------------------|
| Indice en octet | Description | Endianess | Type |
| 0 -> 8 | Taille | Little | Unsigned long long |
| 8->12 | uid | Little | Unsigned int |
| 12->16 | gid | Little | Unsigned int |
| 16->20 | ftime | Little | Unsigned int |
| 20->24 | mtime | Little | Unsigned int |
| 24->28 | atime | Little | Unsigned int |
| 28->32 | ctime | Little | Unsigned int |
| 32->34 | mode | Little | Unsigned int |
| 34->36 | ext_mode | Little | Unsigned short |
| 36->100 | 16 pointeurs pointant sur des blocs de données | Little | 16 X Unsigned int |
| 100->101 | file levels | Little | Unsigned char |
| 101->102 | status (1= dossier, 2 = supprimé, 3 = fichier) | Little | Unsigned char |
| 104->128 | 0 | Little | 6 X Unsigned int |
| | | | |

FIGURE 65 – Structure d'un inode

Comme on peut le constater grâce au schéma ci-dessus, un inode est composé des métadonnées ainsi que 16 pointeurs pointant sur les données de l'objet. Si 16 pointeurs ne sont pas suffisants à adresser toutes les données, alors des niveaux de blocs de pointeurs intermédiaires sont utilisés de la même façon que pour récupérer les inodes à partir du Root Node. Si les 16 pointeurs suffisent, alors le champ *file levels* est égal à 0 et les données sont directement accessibles via ces 16 pointeurs.

Dans le cas d'un fichier, ces pointeurs pointent sur les blocs contenant les données du fichier. Dans le cas d'un dossier, ces pointeurs pointent sur des blocs contenant les noms des sous-dossiers et fichiers enfants ainsi que leurs **ID d'inodes**. Les blocs de ce type auront la forme suivante :

Un bloc pointé par un inode de type dossier

| Objet 1 | | | Objet 2 | | |
|--------------------|--------------------|----------------------|--------------------|--------------------|----------------------|
| 0->4 | 4->5 | 5-> 32 | 0->4 +32 | 4->5 +32 | 5-> 32 +32 |
| Id de l'inode | Taille du nom | Nom de l'objet | Id de l'inode | Taille du nom | Nom de l'objet |
| Objet 3 | | | Objet 4 | | |
| 0->4 +64 | 4->5 +64 | 5-> 32 +64 | 0->4 +96 | 4->5 +96 | 5-> 32 +96 |
| Id de l'inode | Taille du nom | Nom de l'objet | Id de l'inode | Taille du nom | Nom de l'objet |

Au maximum BLOCKSIZE/32 objets par bloc

FIGURE 66 – Bloc contenant les objets enfants d'un répertoire

On remarque qu'avec cette structure de données, que les noms ne peuvent pas dépasser 27 caractères. En effet, les objets ayant des noms longs de plus de 27 caractères sont gérés différemment. Dans ce cas-là, le nom de l'objet sera remplacé par un identifiant qui servira à retrouver le nom long (voir le chapitre concernant les noms longs). Ci-dessous, un bloc pointé par un pointeur de l'inode du dossier racine dans un périphérique utilisant le système de fichiers QNX6 :

FIGURE 67 – Bloc contenant les objets enfants du répertoire racine

Ci-dessous, un inode que l'on visualise avec un outil permettant de visualiser le code hexadécimal. On retrouve la structure de l'inode décrite précédemment. D'un seul coup d'oeil, on peut remarquer que le fichier est de taille 0x29 (42 octets) et que la valeur du filelevels est 0. Puisque ce champ est à 0, il n'y a pas de pointeurs intermédiaires entre les pointeurs contenus dans l'inode et les données. On remarque aussi que parmi les 16 pointeurs de l'inode, seulement un est utilisé (les autres sont à 0xffffffff). Tout cela est logique, puisqu'on a vu que la taille du fichier (42 octets) est inférieure à la taille d'un bloc (1024 octets par défaut). On a donc besoin que d'un seul pointeur vers un bloc pour adresser toutes les données du fichier. Cela explique donc qu'il n'y a qu'un pointeur servant à adresser les données et qu'il n'y ait pas de pointeurs intermédiaires. Enfin, le status vaut 2, cela nous indique qu'il s'agit d'un fichier ou d'un dossier **supprimé**.

FIGURE 68 – Visualisation d'un inode en hexadécimal

7.5 Récupération des longs noms de fichiers

Comme nous l'avons vu précédemment, les fichiers ayant des noms longs sont traités différemment. La liste des noms longs peut être récupérée à partir d'un Root Node ("Long file Node" sur le schéma) contenu dans le Super Block. Ce "Long file Node" possède la même structure que le Root Node permettant de récupérer les inodes. Les 16 pointeurs faisant référence aux noms longs ou à des blocs de pointeurs intermédiaires selon la valeur du champ "level". La méthode pour retrouver les noms longs est similaire à celle pour retrouver les inodes.

| Structure d'un Super Block | | | | |
|----------------------------|---|---|------------|---------------------------|
| Offset : 0x2000 | Indice en octets | Commentaire | Endianness | Type |
| | 0-> 4 : Numéro Magique | Numéro Magique permettant d'identifier le système de fichier QNX6 : 0x68191122, QNX4 : 0x002f | Little | Unsigned int |
| | 4-> 8 : Checksum | | Big | Unsigned int |
| | 8-> 16 : Numéro de série | Permet d'identifier le SB actif | Little | Unsigned long long |
| | 16-> 20 : Change Time | | Little | Unsigned int |
| | 20-> 24 : Access Time | | Little | Unsigned int |
| | 24-> 28 : Flags | | Little | Unsigned int |
| | 28-> 30 : Version1 | | Little | Unsigned short |
| | 30-> 32 : Version2 | | Little | Unsigned short |
| | 32-> 48 : Volume ID | | Little | Unsigned char X 16 |
| | 48-> 52 : Block Size | | Little | Unsigned int |
| | 52-> 56 : Nombre d'inodes | Nombre d'inodes total | Little | Unsigned int |
| | 56-> 60 : Nombre d'inodes libres | | Little | Unsigned int |
| | 60-> 64 : Nombre de blocks | Nombre de blocs total | Little | Unsigned int |
| | 64-> 68 : Nombre de blocks libres | | Little | Unsigned int |
| | 68-> 72 : Alloc Group | | Little | Unsigned int |
| | 72-> 152 : Root Node | Contient le Root node constitué de 16 pointeurs qui permettent d'obtenir tous les inodes | | |
| | 232-> 312 : Long File Node | Les noms longs sont gérés différemment | | |

FIGURE 69 – Structure d'un super Block

La différence avec le Root Node est que chaque pointeur fait référence à un bloc de pointeurs d'une taille fixe de 512 octets. Puisque chaque pointeur est un "Unsigned int" de 4 octets, on a donc $512/4$ soit 128 pointeurs par bloc.

Au niveau level-1, on a donc 16 blocs de pointeurs avec 128 pointeurs par bloc soit $128*16$ pointeurs de niveau level-1.

Au niveau level-2, $16*128$ blocs de pointeurs et donc $16*128*128$ pointeurs de niveau level-2. Une fois qu'on arrive au niveau 0, de la même façon que pour les inodes, on peut lire le nom long.

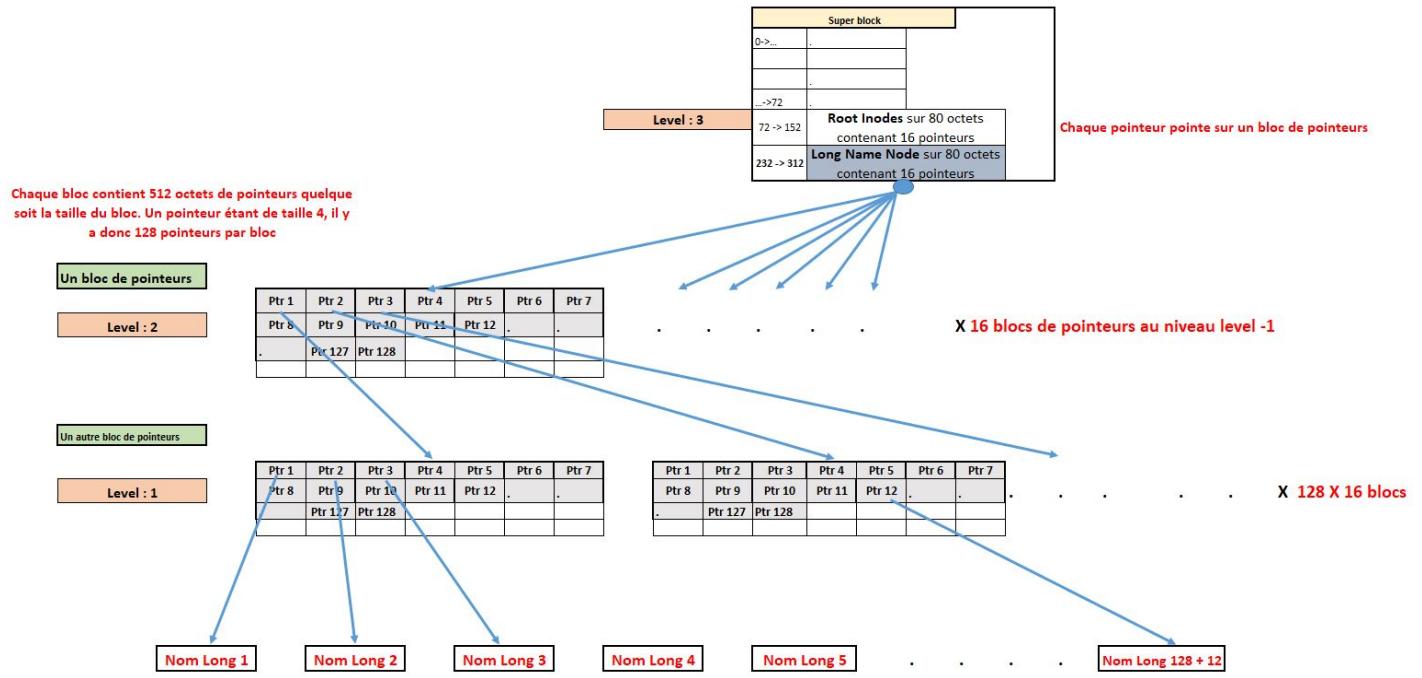


FIGURE 70 – Récupération des noms longs à partir du "Long Name Root Node"

Les deux premiers octets renseignent sur la taille du nom. Les octets suivant correspondent au nom. Chaque pointeur pointe sur un bloc dans lequel se trouve un nom long. Cependant l'espace utilisé par le nom long dans le bloc est de maximum 512 octets quelque soit la taille des blocs dans le système de fichiers. **Puisque 2 octets sont utilisés pour renseigner la taille, on en déduit que la taille maximum d'un nom dans le système de fichiers QNX6 est de 510 octets. Puisque chaque caractère est de type unsigned char (1 octet), La taille maximum d'un nom de fichier ou de dossier dans le système de fichiers QNX6 est de 510 caractères.**

| Structure d'un nom long | | | |
|-------------------------|-----------------------------|------------|------------------------|
| Indice en octets | Description | Endianness | Size/Type |
| 0 -> 2 | Taille du nom | Little | 1 X Unsigned short |
| 2 -> TAILLE +2 | Nom long du fichier/dossier | Little | TAILLE X Unsigned char |

Chaque pointeur pointe sur un bloc dans lequel se trouve un nom long. Cependant l'espace utilisé par le nom long dans le bloc est de maximum 512 octets quelque soit la taille des blocs dans le système de fichiers.

FIGURE 71 – Structure d'un nom long

7.6 Récupération et organisation des données

L'ID d'un inode fait référence à sa position quand on lit tous les inodes à partir du Root Node. Le tout premier inode obtenu en lisant les pointeurs du Root Node aura pour ID 1.

La méthode la plus simple pour récupérer les données est de lire tous les objets à partir de l'inode ayant l'ID 1 qui correspond au dossier Racine. La liste des inodes peut être obtenue à partir du Root Node comme expliqué dans le chapitre précédent. Les pointeurs contenus dans l'inode correspondant à un dossier pointent sur le nom des objets enfants ainsi que leurs **ID d'inodes**. On peut donc facilement obtenir toute l'arborescence de fichiers et d'objets.

Dans le cas d'un inode correspondant à un fichier, les pointeurs contenus dans l'inode pointent sur les données dans le cas d'un fichier de petite taille, ou sur des blocs de pointeurs intermédiaires dans le cas d'un fichier volumineux. On rappelle que l'inode contient 16 pointeurs sur les données ou sur des blocs de données intermédiaires. Avec une taille de bloc de 1024 octets, au maximum 256 pointeurs (de taille 4 octets) peuvent être contenus dans un seul bloc. Chacun de ces pointeurs faisant référence à un bloc (de 1024 octets), on a donc avec un seul niveau (level=1) de pointeur intermédiaire, $16 * 256 * 1024$ (4 Mo) octets de données qui peuvent être adressés.

Un niveau fait référence à un niveau de bloc de pointeurs intermédiaire. Si les 16 pointeurs font directement référence aux données alors le niveau est 0. Si les pointeurs font référence à des blocs de pointeurs qui eux-mêmes font référence aux données alors le niveau est de 1.

| Nombre de niveaux intermédiaires | Taille du fichier maximal |
|----------------------------------|---|
| 0 | Au maximum $16 * 1024 : 16 \text{ Ko}$ |
| 1 | Au maximum $16 * 256^1 * 1024 : 4 \text{ Mo}$ |
| 2 | Au maximum $16 * 256^2 * 1024 : 1 \text{ Go}$ |
| 3 | Au maximum $16 * 256^3 * 1024 : 256 \text{ Go}$ |

On peut directement remarquer qu'il y a une logique entre la numérotation des inodes et l'arborescence. En effet, nous avons remarqué grâce à l'analyse hexadécimale que le numéro des inodes semblerait dépendre de l'ordre de création des objets associés.

On sait que le dossier racine a l'identifiant 1. Or, en analysant l'hexadécimal, on remarque que les fichiers dans le répertoire racine ont des numéros proches de 0. Nous avons donc réalisé plusieurs tests pour essayer de trouver la logique dans l'organisation des inodes. Pour cela, nous avons créé différents fichiers dans le répertoire racine ainsi que des dossiers.

Nous avons donc remarqué que le fichier boot présent de base correspond à l'ID 2 (le deuxième inode). Si on crée un fichier, on remarque que celui-ci sera associé au troisième inode et ainsi de suite. Admettons que l'on ait 4 fichiers dans le répertoire racine, alors le premier fichier aura pour ID 3, le deuxième aura pour ID 4...ect. A chaque fois que l'on crée un fichier, l'ID est incrémenté de 1 par rapport au dernier fichier créé **dans le répertoire**.

Dossier courant ".." : ID 1
Fichier ".boot" : ID 2
Premier fichier "presentation.txt" : ID 3
Troisième fichier "t.txt" : ID 5

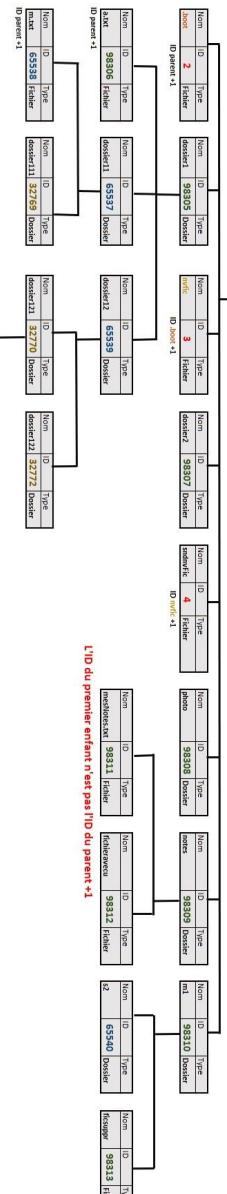
FIGURE 72 – Numéros des inodes dans un même dossier

On voit bien sur le schéma que les fichiers dans un même dossier ont bien des ID d'inode qui se suivent. On a donc le fichier .boot qui correspondra au deuxième inode dans la liste des inodes. Le fichier "présentation.txt" qui correspondra au 3ème inode ect... Cependant cela est vrai seulement pour les fichiers. Pour les dossiers, il semblerait que leurs ID ne suivent pas la même logique. En effet, si on regarde le schéma de plus près, on se rend compte que le dossier "photos" dans le dossier racine a l'ID : 01 80 01 00, ce qui donne 0x18001 (98305) puisque la lecture se fait en little endian. Le dossier "photos" correspond donc au 98305 ème inode. Si on regarde le prochain dossier dans le même répertoire, on se rend compte qu'il s'agit du 98306 ème inode. Cependant, il semblerait que cela ne soit pas toujours le cas. En effet, nous avons remarqué lors de plusieurs observations que les ID des inodes des dossiers ne suivaient pas forcément cette logique. Nous nous sommes donc intéressés à la manière dont les inodes étaient organisés par rapport à l'arborescence telle que la voit un utilisateur. Une explication textuelle serait très compliquée et peu compréhensible.

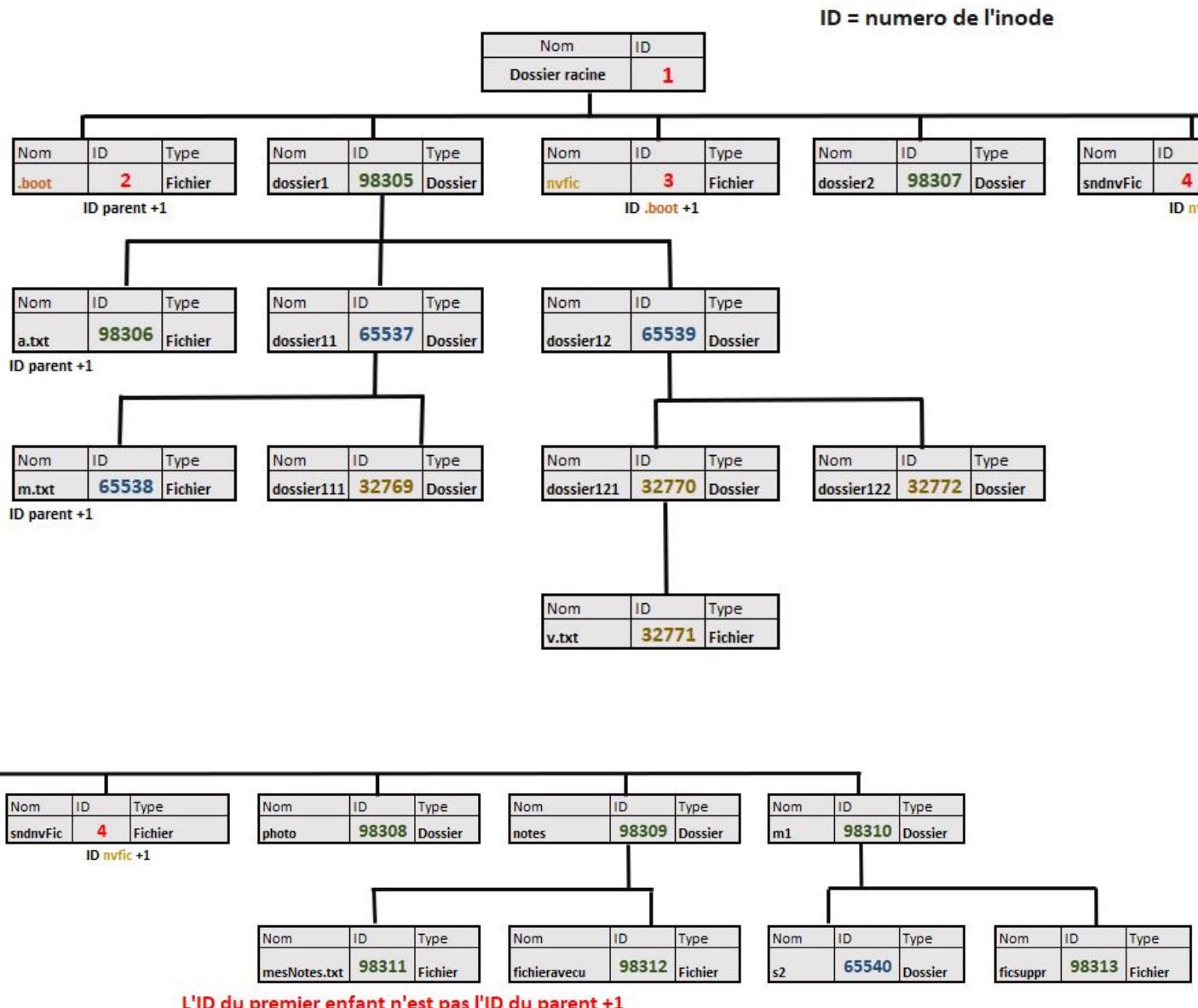
Ci-dessous, un schéma permettant de faire le lien entre la vue de l'arborescence telle que la verrait un utilisateur avec les numéros d'inode.

Lien entre l'emplacement des inodes et une arborescence de fichiers/dossiers

ID = numero de l'inode



Lien entre l'emplacement des inodes et une arborence de fichiers/dossiers



Voici le schéma représentant une arborescence de fichiers et de dossiers et qui fait le lien entre l'**ID des inodes** (leur numéro dans la liste des inodes) et leur emplacement réel dans les dossiers. L'ID de l'inode est affecté par ordre chronologique de création du fichier dans le dossier.

Rappel : L'ID d'un inode fait référence à sa position quand on lit tous les inodes à partir du Root Node. Le tout premier inode à être obtenu en lisant les pointeurs du

Root Node aura pour ID 1.

Dans le cas de "a.txt", l'ID est égale à l'ID de son parent puisqu'il s'agit d'un fichier et qu'il a été créé juste après avoir créé le répertoire parent. Si on regarde le répertoire "notes", le fichier mesNotes.txt n'a pas un ID égal à l'ID de son parent+1. Cela vient du fait que le dossier "m1" a été créé avant. Ce dernier a donc pris l'ID : ID "notes" + 1. Après avoir créé le dossier "m1", on a créé le fichier mesNotes.txt dans le dossier notes. Le fichier mesNotes.txt a donc pris l'ID du dossier m1 +1. Cela montre bien que l'ID des inodes est affecté de façon chronologique par rapport à la date de création des objets. Lors de nos tests, le premier sous-dossier créé dans le répertoire racine possède TOUJOURS l'ID 98305. De la même manière, le premier sous-sous dossier du répertoire racine possède TOUJOURS l'ID 65537.

Pour finir, voici cette même arborescence mais sous la forme d'un dictionnaire que nous utilisons pour l'ingest module que nous avons développé. Il montre le nom des fichiers et des dossiers et pour chacun d'eux, leur répertoire parent :

```
INFO: {98305L: {'Name': 'dossier1', 'ROOT_INODE': 1L}, 65537L: {'Name': 'dossier11', 'ROOT_INODE': 98305L}, 32769L: {'Name': 'dossier111', 'ROOT_INODE': 65537L}, 2L: {'Name': '.boot', 'ROOT_INODE': 1L}, 98307L: {'Name': 'dossier2', 'ROOT_INODE': 1L}, 65539L: {'Name': 'dossier12', 'ROOT_INODE': 98305L}, 32770L: {'Name': 'dossier121', 'ROOT_INODE': 65539L}, 3L: {'Name': 'nvfic', 'ROOT_INODE': 1L}, 98306L: {'Name': 'a.txt', 'ROOT_INODE': 98305L}, 65538L: {'Name': 'm.txt', 'ROOT_INODE': 65537L}, 32771L: {'Name': 'v.txt', 'ROOT_INODE': 32770L}, 98309L: {'Name': 'notes', 'ROOT_INODE': 1L}, 4L: {'Name': 'deuxiemenuveauefichieravecunnomtrestreslong.txt', 'ROOT_INODE': 1L}, 32772L: {'Name': 'dossier122', 'ROOT_INODE': 65539L}, 98308L: {'Name': 'photos', 'ROOT_INODE': 1L}, 65540L: {'Name': 's2', 'ROOT_INODE': 98310L}, 98311L: {'Name': 'mesNOTES.txt', 'ROOT_INODE': 98309L}, 98310L: {'Name': '1', 'ROOT_INODE': 1L}, 98313L: {'Name': 'deleted-ficsuppr', 'ROOT_INODE': 98310L}, 98312L: {'Name': 'fichieravecunnomtrestrestreslong.txt', 'ROOT_INODE': 98309L}, 98315L: {'Name': 'deleted-fic2suppr', 'ROOT_INODE': 98310L}}
```

7.7 Récupération des données effacées

Grâce à l'étude par reverse-engineering ainsi qu'à l'analyse hexadécimale du système de fichiers, nous avons pu comprendre comment été gérée la suppression des données dans un système de fichiers QNX6. Cela nous a permis par la suite de développer une fonction pour retrouver certaines données effacées par l'utilisateur mais qui sont toujours présentes sur le périphérique. En effet, quand l'utilisateur supprime des fichiers ou des dossiers, ces derniers peuvent être récupérés dans la majorité des cas. Nous avons identifié plusieurs niveaux de suppression dans le système de fichiers.

7.7.1 Lecture du BackUp Super block

Comme nous l'avons vu précédemment, le système de fichiers QNX6 est composé de deux super Blocks. L'un correspondant au super Block Actif ou "Working Super Block" qui contient toutes les données modifiées à un instant T et l'autre correspondant au BackUp Super Block ou "Stable Super Block" qui contient une version antérieure stable pas encore modifiée.

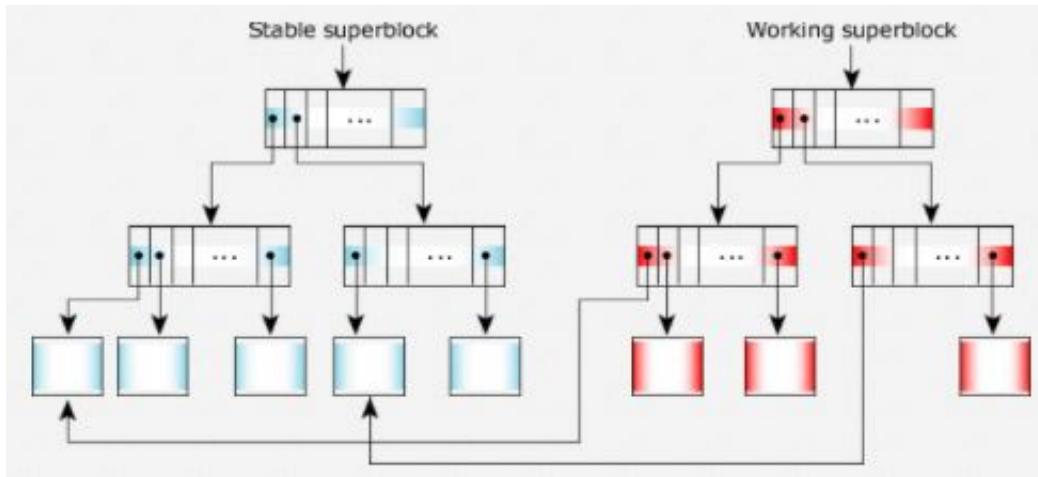


FIGURE 73 – Schéma des super blocs :

https://developer.blackberry.com/playbook/native/documentation/com.qnx.doc.neutrino.sys_arch/topic/fsys_cow_filesystem.html

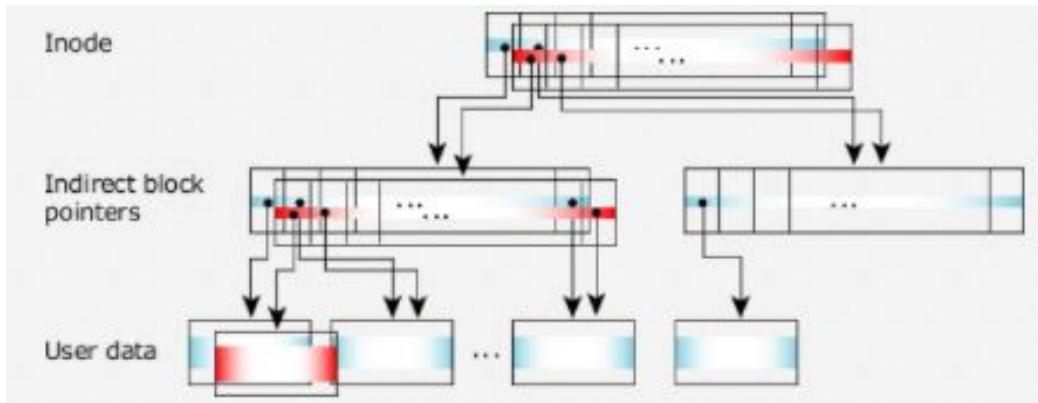


FIGURE 74 – Schéma des super blocs :

https://developer.blackberry.com/playbook/native/documentation/com.qnx.doc.neutrino.sys_arch/topic/fsys_cow_filesystem.html

Au moment du traitement de l’arborescence à partir de l’inode 1 (qui correspond au dossier racine), un nom de fichier ou dossier supprimé n’aura plus de lien vers ses données. Plus précisément l’ID de l’inode du fichier aura disparu. On ne pourra donc plus faire le lien entre, le nom du fichier ainsi que le répertoire dans lequel il se trouve et les données du fichier.

FIGURE 75 – Bloc de données pointé par l'inode d'un dossier contenant des fichiers et des répertoires

Le schéma ci-dessus montre un bloc pointé par un inode correspondant à un dossier Y qui contient plusieurs fichiers et sous dossiers.

Pour chaque élément dans ce dossier Y, on peut donc récupérer les noms ainsi que l'ID des inodes appartenant aux différents dossiers et fichiers. Ces inodes permettent par la suite de récupérer les données d'un fichier ou d'obtenir les sous éléments (dossiers et fichiers) dans le cas d'un répertoire. Chaque bloc de ce type possède $\text{BLOCKSIZE}/32$ éléments de taille 32 (Pour plus d'informations, voir chapitre Inode). Les quatre premiers octets d'un élément correspondent à l'ID de l'inode d'un fichier ou d'un sous-répertoire. Cependant, dans le cas d'un fichier ou dossier supprimé, l'ID de l'inode disparaît et le lien avec les données du fichier ne peut plus être fait.

| | | | |
|-------|--|---|---|
| 03 00 | Dans le cas d'un dossier ou fichier supprimé l'ID de l'inode à disparaître | 61 74 69 6f 6e 2e 74 78 74 00 00 00 |notes..... presentationattestation.png..... perso |
| 00 00 | | 05 70 65 72 73 6f 00 00 00 00 00 00 |t.txt..... |
| 00 00 | | 00 00 00 00 06 00 00 00 0d 69 6d 61 | gevbin.png.....m1 |
| 00 00 | | 00 00 00 00 00 00 00 00 00 00 00 00 |logo.jpg.....k.txt |
| 00 00 | | 00 00 00 00 00 00 00 00 00 00 00 00 |7.txt..... |

FIGURE 76 – Dans le cas d'un dossier ou d'un fichier supprimé, l'ID de l'inode a disparu

En effet, on constate que l'ID de l'inode vaut 0, ce qui est incorrect. Cependant, le nom est toujours présent ainsi que les données même si le lien entre les deux ne peut plus être fait. Une méthode pour retrouver l'ID de l'inode en question est de lire exactement le même inode du dossier parent (celui du dossier Y dans notre cas) identifié lui aussi par un ID mais à partir de la liste des inodes récupérés du **BackUp Super Block**.

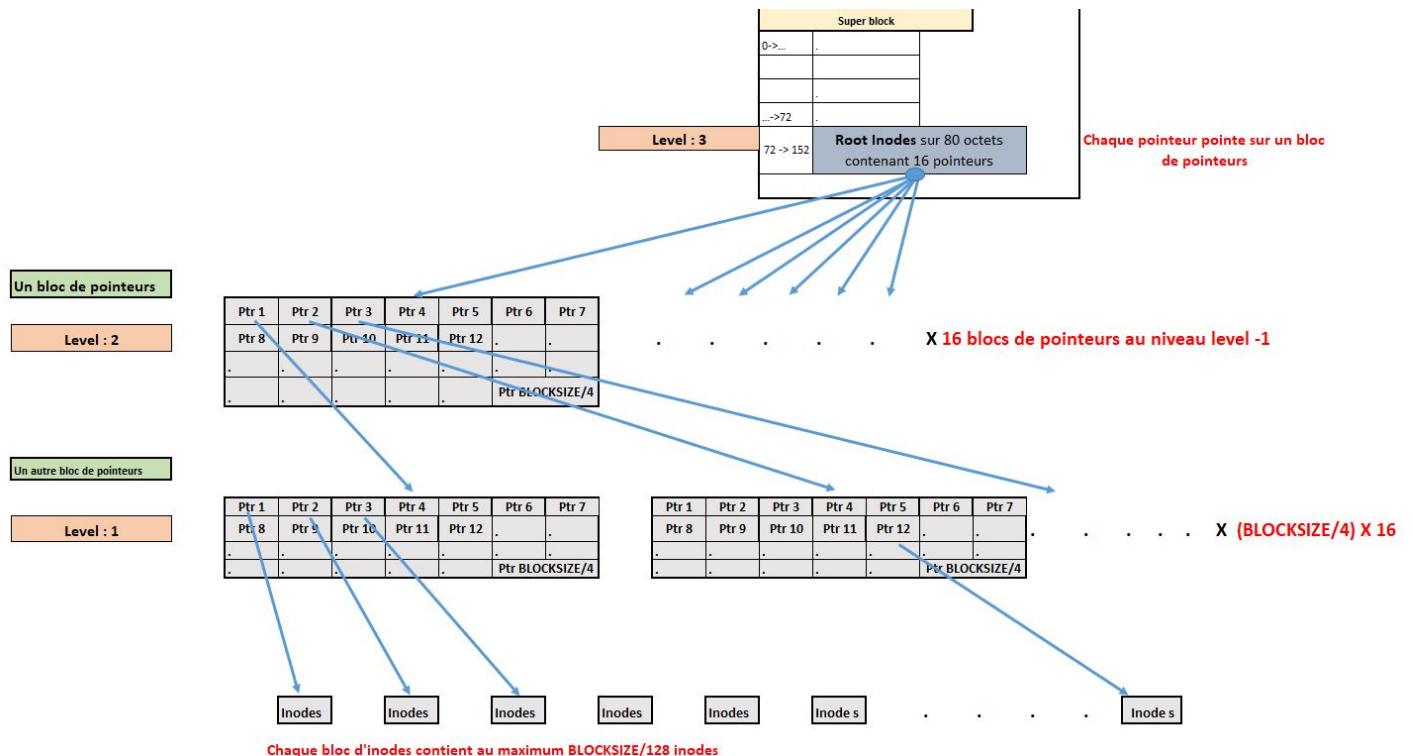


FIGURE 77 – Organisation des inodes à partir des Root Nodes d'un Super Block

Dans l'exemple, on admet que le dossier Y a pour ID 54. Au moment de lire les ID des fichiers et des sous-dossiers contenus dans le dossier Y, certains ID sont à 0. On va donc regarder pour l'inode ayant l'ID 54 dans la liste des inode du **Backup Super Block** pour retrouver les IDs manquants. En effet, le Backup Super Block (appelé aussi stable Super Block) contient une version "antérieure" et "stable" de la structure. En lisant donc le même inode, le même pointeur dans la liste des pointeurs de l'inode et en lisant le même fichier ou dossier mais à partir du Backup Super Block, on peut retrouver l'ID d'un inode correspondant à un fichier ou dossier supprimé. Dans notre ingest module, c'est la méthode la plus fiable pour retrouver des données supprimées.

7.7.2 Autre méthode pour récupérer l'ID d'un Inode

Dans le cas où la méthode précédente qui consiste à lire le Backup Super Block n'a pas fonctionné, alors il se peut aussi que l'ID de l'inode soit 1 BLOCKSIZE octets plus loin. En effet, dans certains cas, quand on supprime un fichier ou un dossier, l'ID de l'inode est juste déplacé BLOCKSIZE octets plus loin.

FIGURE 78 – Récupération de l'ID de l'inode en lisant BLOCKSIZE octets plus loin

Dans ce cas-là, on se rend compte que l'ID de l'inode est égal à 0 pour le fichier "fic2suppr". Cependant, si on regarde BLOCKSIZE octets plus loin (1024 octets dans ce cas-là), on s'aperçoit que l'ID de l'inode de "fic2suppr" est présent.

7.7.3 Récupération des données sans Inode ID

Enfin, si les deux méthodes précédentes n'ont pas fonctionné, on peut aussi regarder tous les inodes ayant un "status" égal à 0x2 et qui n'ont pas été déjà traités. Un inode ayant pour status 2 correspond à l'inode d'un dossier ou d'un fichier supprimé. Cependant avec cette méthode, on peut, certes, retrouver les données, mais il est impossible de faire le lien avec le nom du fichier ou le chemin du répertoire parent.

8 Développement de l'ingest Module

Code source de l'ingest module :

<https://github.com/jdbonfils/QNX6-Files-System-Reader-Ingest-Module>

Certaines fonctions développées sont inspirées des projets suivants :

<https://nop.ninja/> - **Mathew Evans**

<https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/linux-kaunetem/-/tree/505a666ee3fc611518e85df203eb8c707995ceaa/fs/qnx6>

<https://github.com/ReFirmLabs/binwalk/issues/365>

8.1 Définition des classes

Grâce à toutes les informations exposées ci-dessus que nous avons obtenues grâce au reverse engineering des drivers, à l'analyse hexadécimale du système de fichiers et à la documentation, nous avons pu développer l'ingest module retrouvant les données d'un périphérique QNX6. De plus, le module est aussi capable de récupérer les données supprimées jusqu'à un certain stade. En effet, comme expliqué précédemment, il existe plusieurs niveaux de suppression des données. Les noms longs sont également gérés par l'ingest module (Voir 7.7).

La prise en main de la librairie "The Sleuth Kit" pour le développement en JPython d'ingest module a été assez compliquée. En effet, la librairie dispose de multiples classes différentes et de nombreux concepts très abstraits. De plus, la documentation sur la librairie est très maigre rendant la compréhension des classes très difficile. La meilleure méthode pour apprendre à utiliser cette librairie que nous avons trouvée consiste à regarder le code des autres ingest modules développés par la communauté et d'essayer de comprendre les classes et les fonctions utilisées. Le but étant de maîtriser au mieux la librairie pour avoir la meilleure interaction possible avec l'utilisateur.

Le module développé est un **Data-source ingest module**. Pour le développement de ce dernier, nous avons donc défini trois classes. La première classe,

QNX6ReaderIngestModuleFactory qui contient les informations de description de l'ingest module. La seconde classe, **QNX6_FS** prend en paramètres de son constructeur un **AbstractFile** qui est un type propre à Autopsy. La classe prend également un objet de type logger qui permet d'écrire dans les fichiers de logs. Cet **AbstractFile** correspond au périphérique QNX6 à analyser. La classe **QNX6_FS** contient un certain nombre de fonctions permettant de réaliser différentes opérations sur le système de fichiers et renvoyer un résultat. Enfin, la classe **QNX6ReaderIngestModule** définie de la même façon que dans le chapitre "Autopsy" avec une méthode **Strartup** et une méthode **process** où toutes les opérations sur la Data-Source sont faites. Cette classe appelle les fonctions de **QNX6_FS** pour obtenir les différents résultats de l'analyse du système de fichiers et se charge de les afficher et de le gérer.

8.2 Développement du parser pour le système de fichiers QNX6

Nous avons défini un certain nombre de constantes dans cette classe telle que le Magic Number associé au système de fichiers QNX6 ou encore la taille du Super Block :

```
QNX6_BOOTBLOCK_ZONE = 0
QNX6_BOOTBLOCK_SIZE = 0x2000
QNX6_SPBLOCK_SIZE   = 0x200
QNX6_SPBLOCK_ZONE   = 0x1000
QNX6_MAGIC_ID        = 0x68191122
QNX6_PTR_MAX_LEVELS = 5
QNX6_SHORT_NAME_MAX = 27
QNX6_LONG_NAME_MAX  = 510
```

FIGURE 79 – Constantes définies dans la classe QNX6_FS

8.2.1 Fonction : parseSuperBlock

Tout d'abord, nous avons repris en partie une fonction trouvée sur internet permettant de parser un superBlock QNX6 : <https://github.com/ReFirmLabs/binwalk/issues/365>

La fonction prend en paramètres l'offset à partir duquel démarre le super Block. Par défaut, la valeur est fixée à 0x2000. A partir de là, on construit un dictionnaire contenant les informations du Super Block. De même que pour le Super Block, nous avons développé une fonction permettant de parser un inode et un Root Node en un dictionnaire contenant les différentes données.

```

299     #Lit le super block commençant à l'offset
300     def readSuperBlock(self, offset = 0x2000):
301         buffer = jarray.zeros( self.QNX6_SPBLOCK_SIZE, "b")
302         self.devQNX6.read(buffer,offset,self.QNX6_SPBLOCK_SIZE)
303         spBlock = {}
304         spBlock["magic"] = unpack('<I', buffer[:4])[0]
305         spBlock['checksum'] = (unpack('>I', buffer[4:8]))[0]
306         spBlock['serialNum'] = unpack('<Q', buffer[8:16])[0]
307         spBlock['ctime'] = unpack('<I', buffer[16:20])[0]
308         spBlock['atime'] = unpack('<I', buffer[20:24])[0]
309         spBlock['flags'] = unpack('<I', buffer[24:28])[0]
310         spBlock['v1'] = unpack('<H', buffer[28:30])[0]
311         spBlock['v2'] = unpack('<H', buffer[30:32])[0]
312         spBlock['volumeid'] = unpack('<16B', buffer[32:48])
313         spBlock['tailleBlock'] = unpack('<I', buffer[48:52])[0]
314         spBlock['nbInodes'] = unpack('<I', buffer[52:56])[0]
315         spBlock['nbInodesLibres'] = unpack('<I', buffer[56:60])[0]
316         spBlock['nbBlocks'] = unpack('<I', buffer[60:64])[0]
317         spBlock['nbBlocksLibres'] = unpack('<I', buffer[64:68])[0]
318         spBlock['allocgroup'] = unpack('<I', buffer[68:72])[0]
319         spBlock['SB_end'] = offset + self.QNX6_SPBLOCK_ZONE;
320         spBlock['RootNode'] = self.parseQNX6RootNode(buffer[72:152])
321         spBlock['Bitmap'] = self.parseQNX6RootNode(buffer[152:232])
322         spBlock['Longfile'] = self.parseQNX6RootNode(buffer[232:312])
323     return spBlock

```

FIGURE 80 – Fonction permettant de lire un Super Block

8.2.2 Fonction : getInodesFromRootNode

La fonction getInodesFromRootNode() prend en paramètres :

- La liste des 16 pointeurs contenus dans le Root Node (le Root Node se situe dans le Super Block). C'est à partir de ces 16 pointeurs que l'on va générer la liste de tous les inodes.
- La taille de bloc ainsi qu'un offset sont aussi passés en paramètres. Ces valeurs vont notamment nous permettre de calculer l'adresse des blocs pointés par les différents pointeurs.
- Le "level" du Root Node nous permettant de savoir combien de niveaux de blocs de pointeurs intermédiaires séparent le Root Node des inodes (voir 7.4).

La première partie de la fonction permet de lire les différents niveaux de blocs de pointeurs pour arriver jusqu'aux inodes. Chaque pointeur faisant 4 octets, il y a donc $\text{BLOCKSIZE}/4$ pointeurs par bloc. Une fois que l'on arrive au niveau 0, on a donc la liste des pointeurs pointant directement sur les blocs contenant les inodes. Puisque chaque inode fait 128 octets, il y a donc $\text{BLOCKSIZE}/128$ inodes par bloc. Enfin pour chaque inode, on parse les 128 octets correspondant à l'inode grâce à la fonction "parseInodeEntry".

```

63 ▼   def getInodesFromRootNode(self, listIndBlocks, tailleBlock, offset, level):
64       inodeTree = {}
65       buff = jarray.zeros( tailleBlock, "b")
66       #On lit les pointeurs jusqu'à arriver au niveau 0
67 ▼     for lvl in range(level,0,-1):
68         listIndBlocksTMP = []
69 ▼       for indiceBlock in listIndBlocks:
70           if(self.checkQNX6ptr(indiceBlock)):
71               ptr = indiceBlock * tailleBlock + offset
72               self.devQNX6.read(buff,ptr,tailleBlock)
73               listIndBlocksTMP += list(unpack('<'+str(tailleBlock//4)+'I', buff))
74       listIndBlocks = deepcopy(listIndBlocksTMP)
75       #Des qu'on arrive au niveau des inodes entries on lit tailleBlock/128 inodes entries par block
76       nbInode = int(tailleBlock/128)
77 ▼     for indiceBlock in listIndBlocks:
78         ptr = indiceBlock * tailleBlock + offset
79         self.devQNX6.read(buff,ptr,tailleBlock)
80 ▼       for i in range(0,nbInode):
81           try:
82               inodeTree[len(inodeTree)+1] = self.parseInodeEntry(buff[i*128:(i+1)*128])
83 ▼             except:
84                 inodeTree[len(inodeTree)+1] = None
85                 break
86
86     return inodeTree

```

FIGURE 81 – Fonction permettant de générer la liste des inodes à partir du Root Node

On construit, grâce à cela, un dictionnaire d'inodes où chaque inode est identifié par son ID (sa position dans la liste). Cet ID est très important puisqu'il nous permettra, par la suite, de faire le lien entre un nom de fichier et, ses données ainsi que ses métadonnées. Par exemple, le premier inode parsé correspond au dossier racine. Voici la fonction "parseInodeEntry" générant un dictionnaire à partir de 128 octets passés en paramètres :

```
def parseInodeEntry(self, ie): #qnx6_inode_entry 128b
    IE = {}
    IE['size'] = unpack('<Q', ie[0:8])[0]
    IE['uid'] = unpack('<I', ie[8:12])[0]
    IE['gid'] = unpack('<I', ie[12:16])[0]
    IE['ctime'] = unpack('<I', ie[16:20])[0]
    IE['mtime'] = unpack('<I', ie[20:24])[0]
    IE['atime'] = unpack('<I', ie[24:28])[0]
    IE['ctime'] = unpack('<I', ie[28:32])[0]

    IE['mode'] = unpack('<H', ie[32:34])[0]
    IE['ext_mode'] = unpack('<H', ie[34:36])[0]
    IE['block_ptr'] = unpack('<16I', ie[36:100])
    IE['filelevels'] = unpack('<B', ie[100:101])[0]
    IE['status'] = unpack('<B', ie[101:102])[0]
    IE['unknown2'] = unpack('<2B', ie[102:104])
    IE['zero2'] = unpack('<6I', ie[104:128])

    if(IE['size'] == 0):
        return None

    return IE
```

FIGURE 82 – Fonction permettant de parser un inode

Voici à quoi ressemble la liste des inodes que l'on obtient grâce à la fonction. :

FIGURE 83 – Une petite partie de la liste des inodes parsés

8.2.3 Fonction : getDirTree

Cette fonction est probablement la plus importante de tout le programme. Elle permet de créer une liste de dictionnaires dont un tuple est de la forme : 16 : {nom : "toto.txt", ROOT_ID : 15}. où 16 correspond à l'ID de l'inode associé au fichier "toto.txt" se trouvant dans un dossier associé à l'inode ayant l'ID 15. Cette fonction va permettre de générer l'arborescence de fichiers et de dossiers à partir du répertoire racine. Avec ce dictionnaire contenant la liste des noms des fichiers associés à leurs ID d'inodes ainsi que leur dépendance, il sera par la suite assez simple de reconstruire toute l'arborescence et de facilement retrouver tous les fichiers.

Voici la première partie de la fonction :

```
#Recupere l arborescence de fichiers et repertoires a parti du premier inode qui c
def getDirTree(self,inodeTree,backUpInodeTree,LongNameTree,blocksize,offset):
    delCmp = -1
    dirTree = {}
    buff = jarray.zeros( 32, "b")
    inodeEntryIdList = []
    #Premier Inode = repertoire racine
    inodeEntryIdList.append(1)
    while(inodeEntryIdList): #Tant qu il reste des dossiers a traiter dans l arbo
        cInodeId = inodeEntryIdList.pop(0)
        InodeEntry = inodeTree[cInodeId]
        #Si l inode est un repertoire
        if((InodeEntry != None) and (self.InodeEntry_ISDIR(InodeEntry['mode']))):
            objects = []
            #Pour chaque pointeur de l inode on calcul l adresse du block
            for indPtr in range(0,len(InodeEntry['block_ptr'])):
                ptr = InodeEntry['block_ptr'][indPtr]
                if(ptr != 0xffffffff):
                    addr = ptr * blocksize + offset
                    #Pour chaque block on lit les 32 repertoires ou fichiers
                    for i in range(0,blocksize/32):
                        obj = self.getDataInodeId( addr+(i*32),longNameTree)
```

FIGURE 84 – Fonction permettant de récupérer la liste des dépendances entre les dossiers et fichiers

On part de l'inode ayant l'ID 1 qui correspond au répertoire racine. Si l'inode courant est bien un dossier, on lit sa liste de pointeurs pour retrouver ses données. On rappelle que les "données" d'un dossier correspondent à la liste des "nom + IDinode" des enfants du dossier (voir 7.4). Pour chaque bloc pointé par un pointeur de l'inode correspondant au dossier, on lit BLOCKSIZE/32 "sous-blocs" de 32 octets. Chacun de ces sous-blocs correspond justement au nom+ID d'un enfant du dossier. La fonction getDataInodeId() permet justement de récupérer ces données.

```

#Recupere le nom de l objet et l id de l inode pointant vers les donnees de l objet
def getDataInodeId(self,addr,LongNameTree,namePrefix= "") :
    obj = {}
    buff = jarray.zeros( 32, "b")
    self.devQNX6.read(buff,addr,32)
    obj['PTR'] = unpack('<I', buff[0:4])[0]
    if(unpack('<B', buff[4:5])[0]<=self.QNX6_SHORT_NAME_MAX): #Les fichiers longs sont traites differemment
        obj['Name'] = namePrefix +"".join("%c" % i for i in unpack('<27B', buff[5:32] ) ).replace("\x00",""))
    else:
        longnameKey = unpack('>I', buff[5:9])[0]+1
        if(longnameKey in LongNameTree):
            obj['Name'] = namePrefix+longNameTree[unpack('>I', buff[5:9])[0]+1] #self.LongNames[unpack('<I', ra
            #elif(unpack('<I', buff[12:16])[0] in longNameTree):
            #    objects[str(ptr)+"."+str(i)]['Name'] = longNameTree[unpack('<I', buff[12:16])[0] ] #self.LongN
            else: #Si un fichier avec un nom long a ete supprime alors on le nomme noname
                obj['Name'] = namePrefix+"noname"
    return obj

```

FIGURE 85 – Code de la fonction permettant de récupérer le nom d'un objet et l'ID de son inode associé

L'ID de l'inode associé au nom est noté obj["PTR"] dans la fonction. Cette fonction prend en paramètres l'adresse du "sous-bloc" de 32 octets. A partir de là, la fonction récupère l'ID de l'inode noté obj["PTR"] ainsi que le nom du dossier/fichier associé. Si la taille du nom est supérieure à 27 caractères, alors le nom n'est pas directement accessible. En effet, le nom est remplacé par un identifiant qui nous permet d'obtenir le nom long à partir du "longNameTree" (voir 7.5 et 8.2.7). La fonction renvoie donc un tuple du dictionnaire, par exemple : {nom : "toto.txt", ID : 16}. Attention l'ID est l'ID de l'inode associé au fichier toto.txt et pas au répertoire parent.

8.2.4 Récupération des données

C'est dans la fonction getDirTree que le programme essaie deux des trois méthodes exposées dans le chapitre 7.7 pour retrouver les fichiers et dossiers supprimés. Une fois que l'on a récupéré l'ID de l'inode et le nom de l'enfant (dossier ou fichier) courant du répertoire courant que l'on traite, on vérifie si son ID est égal à 0. Si c'est le cas, alors le lien avec les données a été supprimé. Plus précisément, puisqu'on n'a pas l'ID de l'inode de l'enfant (0 n'est pas un ID valide), on ne peut plus retrouver son inode associé nous permettant de récupérer les métadonnées et les données.

```

#Pour chaque block on lit les 32 répertoires ou fichiers
for i in range(0,blocksize/32):
    obj = self.getDataInodeId( addr+(i*32),longNameTree)
    #Si l id de l inode est égale à 0 c est que le lien vers les données a disparu il s a
    if(obj['PTR'] == 0L or obj['PTR'] == 0):
        #Dans ce cas la on lit le même inode mais dans le backUpInodeTree grâce à l id de
        newAddr = backUpInodeTree[cInodeId]['block_ptr'][indPtr] * blocksize + offset
        obj = self.getDataInodeId(newAddr+(i*32),longNameTree,"deleted-")
        if(obj['PTR'] == 0L or obj['PTR'] == 0): #Si le pointeur est toujours égale à 0 on
            obj = self.getDataInodeId(addr+(i*32)+blocksize,longNameTree,"deleted-") #On
            if(obj['PTR'] == 0xffffffff or obj['PTR'] not in inodeTree or
               inodeTree[obj['PTR']] == None or inodeTree[obj['PTR']]['status'] != 2):
                continue

    objects.append(obj)

```

FIGURE 86 – Suite de la fonction permettant de récupérer la liste des dépendances entre les dossiers et fichiers

Rappel : L'ID d'un inode fait référence à sa position quand on lit tous les inodes à partir du Root Node. Le tout premier inode a été obtenu en lisant les pointeurs du Root Node aura pour ID 1.

La première méthode pour retrouver ce lien est de lire exactement le même inode du dossier parent, lire le même pointeur dans sa liste de pointeurs et lire le même enfant mais à partir du **Backup Super Block**. En effet, comme expliqué précédemment, le Backup Super block maintient une version antérieure du système de fichiers (Attention toutes les données ne sont pas dédoublées pour autant, les pointeurs contenus dans les inodes pointent vers des blocs différents pour les objets modifiés). Cependant puisque plusieurs niveaux de suppression existent, il se peut que la lecture du même inode et du même enfant à partir du Backup Super Block ne donne rien. Dans ce cas-là, si l'ID est toujours égal à 0, on applique la deuxième méthode identifiée pour retrouver des fichiers ou des dossiers. Cette méthode consiste, comme expliqué dans le chapitre précédent, à consulter les 32 octets formant le nom et l'ID de l'inode mais avec un offset de BLOCKSIZE de décalage par rapport à la position normale.

FIGURE 87 – Récupération de l'ID de l'inode en lisant BLOCKSIZE octets plus loin

En effet, lors des différents tests que nous avons menés, nous nous sommes rendu compte que dans certains cas, pour les objets supprimés, le lien vers l'ID de l'inode permettant de faire le lien avec les données était tout simplement "décalé" de BLOCKSIZE octets. Cette méthode fonctionne dans certains cas mais nécessite des vérifications pour s'assurer qu'il s'agit bien d'un ID valide et qu'il correspond bien au bon fichier. En effet, il pourrait très bien y avoir complètement autre chose BLOCKSIZE octets plus loin. On s'assure, tout d'abord, que l'ID de l'inode ne soit pas égal à 0xFFFFFFFF. Puisque cet ID d'inode fait bien référence à un inode se trouvant dans la liste des inodes parsés précédemment, ensuite que cet inode dans la liste ne soit pas égal à None. Enfin, on s'assure que le champ "status" de l'inode identifié par l'ID est bien égal à 0x2. Un inode ayant un "status" égal à 0x2 signifie que l'objet associé à cet inode a été supprimé. Dans ce cas-là, les données n'ont pas physiquement été effacé du périphérique. C'est le lien entre les données et le nom ainsi que le chemin du fichier qui a été supprimé. Même si un utilisateur montant le périphérique QNX6 ne verrait pas le fichier, les données sont toujours présentes dans ce cas-là.

Enfin dans les cas où l'ID de l'inode est valide, on ajoute le couple "nom + ID de l'inode" nommé temporairement "obj" dans le programme, à une liste temporaire.

```

#L'objet ayant nom "." correspond au repertoire parent
for obj in objects:
    if(obj['Name'] == "."):
        rootID=obj['PTR'] #Recupere l id du repertoire parent
        break;
#Pour chaque objets on cree un element du dictionnaire identifie par l id de l inode pointant vers les donnees
for obj in objects:
    if((obj['Name'] != ".") and (obj['Name'] != "") and (obj['Name'] != """) and (obj['Name'] != """)) :
        dirTree[ obj['PTR'] ] = {'Name':obj['Name'], 'ROOT_INODE':rootID}
        if obj['PTR'] >= 1:
            inodeEntryIdList.append(obj['PTR'])

return dirTree

```

FIGURE 88 – Construction du chemin de chaque fichier et dossier

Pour finir, on recherche l'ID de l'inode du répertoire courant nous permettant, par la suite, de construire le chemin de chaque fichier et dossier enfant. L'ID du dossier parent nous permet d'avoir toutes les relations de dépendance entre les fichiers/sous-dossiers et les dossiers. Pour cela, on recherche parmi la liste temporaire construite juste avant, l'objet ayant pour nom "..". Cet objet correspond au dossier courant. (Ce n'est donc pas réellement un enfant).

Une fois cela fait, pour chaque enfant, c'est-à-dire chaque couple nom+ID de l'inode se trouvant dans le répertoire courant, on crée un élément d'une liste. **Chaque élément est de la forme :**

id inode : { 'Name' : 'nom objet', 'ROOT_INODE' : ID parent}.

Cela signifie que le fichier/dossier s'appelant "nom objet" est associé à l'inode ayant l'ID "id inode" et il se situe dans le dossier associé à l'inode ayant comme ID "ID parent". Cette liste nous permet donc de faire le lien entre chaque objet dans l'arborescence et pour chaque objet d'obtenir ses données et ses métadonnées grâce à son ID d'inode.

Voilà à quoi ressemble la liste retournée par le programme :

```
INFO: {98305L: {'Name': 'dossier1', 'ROOT_INODE': 1L}, 0L: {'Name': 'deleted-', 'ROOT_INODE': 32772L}, 65537L: {'Name': 'dossier11', 'ROOT_INODE': 98305L}, 32769L: {'Name': 'dossier111', 'ROOT_INODE': 65537L}, 2L: {'Name': '.boot', 'ROOT_INODE': 1L}, 98307L: {'Name': 'dossier2', 'ROOT_INODE': 1L}, 65539L: {'Name': 'dossier12', 'ROOT_INODE': 98305L}, 32770L: {'Name': 'dossier121', 'ROOT_INODE': 65539L}, 3L: {'Name': 'nvfic', 'ROOT_INODE': 1L}, 98306L: {'Name': 'a.txt', 'ROOT_INODE': 98305L}, 65538L: {'Name': 'm.txt', 'ROOT_INODE': 65537L}, 32771L: {'Name': 'v.txt', 'ROOT_INODE': 32770L}, 98309L: {'Name': 'notes', 'ROOT_INODE': 1L}, 4L: {'Name': 'deuxiemenuveauaufichieravecunnomtrestreslong.txt', 'ROOT_INODE': 1L}, 32772L: {'Name': 'dossier122', 'ROOT_INODE': 65539L}, 98308L: {'Name': 'photos', 'ROOT_INODE': 1L}, 65540L: {'Name': 's2', 'ROOT_INODE': 98310L}, 98311L: {'Name': 'mesNOTES.txt', 'ROOT_INODE': 98309L}, 98310L: {'Name': 'm1', 'ROOT_INODE': 1L}, 98312L: {'Name': 'fichieravecunnomtrestrestreslong.txt', 'ROOT_INODE': 98309L}}
```

FIGURE 89 – Exemple de dirTree représentant les dépendances entre les objets

8.2.5 Fonction : getDirsAndFiles

Cette fonction permet de construire l'arborescence de fichiers et de dossiers à partir de la liste des dépendances ("dirTree") obtenues grâce à la fonction précédente. La fonction permet notamment de faire le lien entre le dirTree et la liste des inodes.

```
INFO: {98305L: {'Name': 'dossier1', 'ROOT_INODE': 1L}, 0L: {'Name': 'deleted-', 'ROOT_INODE': 32772L}, 65537L: {'Name': 'dossier11', 'ROOT_INODE': 98305L}, 32769L: {'Name': 'dossier111', 'ROOT_INODE': 65537L}, 2L: {'Name': '.boot', 'ROOT_INODE': 1L}, 98307L: {'Name': 'dossier2', 'ROOT_INODE': 1L}, 65538L: {'Name': 'dossier12', 'ROOT_INODE': 98307L}, 32770L: {'Name': 'dossier121', 'ROOT_INODE': 65539L}, 3L: {'Name': 'nvfic', 'ROOT_INODE': 1L}, 98306L: {'Name': 'a.txt', 'ROOT_INODE': 98305L}, 65538L: {'Name': 'm.txt', 'ROOT_INODE': 65537L}, 32771L: {'Name': 'v.txt', 'ROOT_INODE': 32770L}, 98309L: {'Name': 'notes', 'ROOT_INODE': 1L}, 4L: {'Name': 'deuxiemeenouveaufichieravecunnomrestreslong.txt', 'ROOT_INODE': 1L}, 32772L: {'Name': 'dossier122', 'ROOT_INODE': 65539L}, 98308L: {'Name': 'photos', 'ROOT_INODE': 1L}, 65540L: {'Name': 's2', 'ROOT_INODE': 98310L}, 98311L: {'Name': 'mesNOTES.txt', 'ROOT_INODE': 98309L}, 98310L: {'Name': 'm1', 'ROOT_INODE': 1L}, 98312L: {'Name': 'fichieravecunnomrestreslong.txt', 'ROOT_INODE': 98309L}}
```

Liste des dépendances (dirTree)

Liste des inodes

FIGURE 90 – Lien entre dirTree et inodeTree

```
#Pour chaque objet dans le dirTree on recuperes les donnees de l'objet grace a l'inode tree
def getDirsAndFiles(self,inodeTree,dirTree,blksize=1024,blkOffset=0):
    dirList = []
    fileList = []
    for keyObj in dirTree: #Pour chaque objet dans le dirTree on recuperes les donnees de l'objet grace a l'inode tree
        if(keyObj > 0):
            if(inodeTree[keyObj] != None and self.InodeEntry_ISDIR(inodeTree[keyObj]['mode'])): #Il s'agit d'un repertoire
                dirList.append(self.getDirFromInodeId(inodeTree,dirTree,keyObj))
                continue
            if(inodeTree[keyObj] != None and not self.InodeEntry_ISDIR(inodeTree[keyObj]['mode'])): #Il s'agit d'un fichier
                fileList.append(self.getFileFromInodeId(inodeTree,dirTree,keyObj,blksize,blkOffset))

    return dirList,fileList
```

FIGURE 91 – Fonction getDirsAndFiles

Pour chaque objet dans la liste des dépendances, le programme appelle une fonction différente selon s'il s'agit d'un fichier ou d'un dossier. La fonction prend en paramètres inodeTree qui est la liste des inodes et dirTree la liste des dépendances entre les objets. La plupart des fonctions ont un paramètre blocksize et offset permettant de calculer l'adresse des blocs pointés suivant la formule : INDICE BLOCK X TAILLE BLOCK + OFFSET. L'offset est presque toujours égal à 0x3000. Cette valeur correspond à l'offset du tout premier bloc dans le système de fichiers. C'est aussi l'offset où se termine le premier Super Block. La fonction renvoie deux listes. La première, la liste des dossiers et la deuxième, la liste des fichiers.

Les tuples sont de la forme :

```

-----Directories Extracted-----
Path : | Name : dossier1 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505655 | ctime : 1618505983 | mtime : 1618505391 | status : 3
Path : dossier1// | Name : dossier11 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505649 | ctime : 1618505983 | mtime : 1618505420 | status : 3
Path : dossier1//dossier11// | Name : dossier111 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505650 | ctime : 1618505983 | mtime : 1618505300 | status : 3
Path : | Name : dossier2 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505691 | ctime : 1618505983 | mtime : 1618505328 | status : 3
Path : | Name : .boot | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505209 | atime : 1618505209 | ctime : 1618505209 | mtime : 1618505209 | status : 1
Path : dossier1// | Name : dossier12 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505656 | ctime : 1618505983 | mtime : 1618505320 | status : 3
Path : dossier1//dossier12// | Name : dossier121 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618767334 | ctime : 1618767334 | mtime : 1618505461 | status : 3
Path : | Name : notes | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618767583 | atime : 1618767734 | ctime : 1618767638 | mtime : 1618767638 | status : 3
Path : dossier1//dossier12// | Name : dossier122 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505461 | ctime : 1618505983 | mtime : 1618505453 | status : 3
Path : | Name : photos | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618767447 | atime : 1618839736 | ctime : 1618839738 | mtime : 1618839738 | status : 3
Path : m1// | Name : s2 | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618839891 | atime : 1618839891 | ctime : 1618839891 | mtime : 1618839891 | status : 3
Path : | Name : mi | Size : 1024 | UID : 0 | GID : 0 | ftime : 1618839877 | atime : 1618859244 | ctime : 1618859231 | mtime : 1618859231 | status : 3

-----Files Extracted-----
Path : dossier1// | Name : a.txt | Size : 60 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505983 | ctime : 1618505983 | mtime : 1618505405 | status : 3
Path : dossier1//dossier11// | Name : m.txt | Size : 39 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505420 | ctime : 1618505983 | mtime : 1618505429 | status : 3
Path : dossier1//dossier12//dossier121// | Name : v.txt | Size : 16 | UID : 0 | GID : 0 | ftime : 1618505983 | atime : 1618505453 | ctime : 1618505983 | mtime : 1618505461 | status : 3
Path : | Name : deuxiemenuouveaufichieravecumnotrestreslong.txt | Size : 28 | UID : 0 | GID : 0 | ftime : 1618839783 | atime : 1618839783 | ctime : 1618839783 | mtime : 1618839810 | status : 3
Path : notes// | Name : mesNOTES.txt | Size : 117 | UID : 0 | GID : 0 | ftime : 1618767519 | atime : 1618767519 | ctime : 1618767582 | mtime : 1618767582 | status : 3
Path : notes// | Name : fichieravecumnotrestreslong.txt | Size : 94 | UID : 0 | GID : 0 | ftime : 1618767638 | atime : 1618767638 | ctime : 1618767669 | mtime : 1618767669 | status : 3

```

FIGURE 92 – Exemple de dirTree

A cela s'ajoute dans la liste des fichiers une colonne "DATA", pour chaque tuple, contenant les données du fichier. Voici le code de la fonction getFilesFromInodes() permettant de construire un tuple dans la liste des fichiers :

```

199 def getFileFromInodeId(self, inodeTree, dirTree, DataINodeID, blksize=1024, blkOffset=0):
200     InodeDataEntry = inodeTree[DataINodeID]
201     if(InodeDataEntry != None and not self.InodeEntry_ISDIR(InodeDataEntry['mode'])):
202         filename = dirTree[DataINodeID]['Name']
203         ## Create DIR List
204         dirpath = ""
205         dirID = DataINodeID
206         while True:
207             if(dirID <= 0x01):
208                 break
209             if(dirID != DataINodeID):
210                 dirpath = dirTree[dirID]['Name'] + "//" + dirpath
211                 dirID = dirTree[dirID]['ROOT_INODE']
212             if(dirID not in dirTree):
213                 dirID != 0x01
214
215         ## Create List of all physical blocks
216         PhysicalPTRs = []
217         for pointer_index in InodeDataEntry['block_ptr']:
218             ## Make sure pointer != 0xFFFFFFFF
219             if pointer_index != 0xffffffff:
220                 ## Calculate Physical Location.
221                 PhysicalPTRs += [(pointer_index*blksize)+blkOffset]
222             #Recupere les donnees du fichier
223             data = self.getDataFromPTR(PhysicalPTRs, InodeDataEntry, InodeDataEntry['filelevels'], b
224
225             return {'path':dirpath, 'name': dirTree[DataINodeID]['Name'] , 'size':InodeDataEntry['
226             return None
227

```

FIGURE 93 – Fonction permettant de récupérer un fichier à partir de l'ID de son inode

La fonction prend en paramètres l'ID de l'inode correspondant au fichier à récupérer. On vérifie, tout d'abord, que l'inode correspond bien à un fichier. La première partie permet de construire le chemin du fichier. Pour cela, on lit successivement l'ID de l'inode correspondant aux dossiers parents. Voici un exemple trivial d'une liste de dépendances (dirTree) :

```
65538 : { "Name" : "fic.txt" , "ROOT_INODE" : 65537 }
65537 : { "Name" : "toto" , "ROOT_INODE" : 98605 }
98605 : { "Name" : "home" , "ROOT_INODE" : 1 }
```

A partir de cette liste, on peut facilement reconstituer le chemin du fichier "fic.txt". L'ID de l'inode de son parent est 65537. Cela signifie que fic.txt se situe dans le dossier toto/. Ainsi de suite, le dossier toto a l'ID de son parent qui vaut 98605. Le dossier toto/ se situe donc dans le dossier home/. De cette manière, on peut reconstruire le chemin de fic.txt : /home/toto/fic.txt

C'est exactement de cette façon-là que le programme procède pour retrouver le chemin d'un fichier. Après cela, la fonction récupère la liste des pointeurs se situant dans l'inode. Ces pointeurs pointent directement ou indirectement vers les données en fonction de la taille de l'objet (voir 7.6). Pour un petit fichier, il n'est pas nécessaire d'avoir des niveaux de pointeurs intermédiaires (voir 7.6). Une fois la liste des pointeurs obtenue, la fonction récupère les données grâce à la méthode getDataFromPTR(). Enfin, la fonction construit le tuple constitué des données et métadonnées et correspondant au fichier courant. La fonction getDirFromInodeId() réalise pratiquement les mêmes opérations sauf que les pointeurs contenus dans l'inode ne sont pas analysés pour récupérer les données. En effet, puisqu'il s'agit de dossiers, les pointeurs contenus dans l'inode pointent sur la liste des enfants de ce dossier. Or, cela est inutile de les récupérer car le programme a déjà récupéré les dépendances entre les objets grâce à la liste dirTree.

8.2.6 Fonction : getDataFromPTR

La fonction getDataFromPTR permet de récupérer les données d'un fichier à partir de la liste de pointeurs contenue dans l'inode associé à ce dernier. La fonction prend donc en paramètres :

- La liste de pointeurs à lire
- Le niveau des pointeurs permettant de savoir combien de blocs de pointeurs intermédiaires séparent les pointeurs des données
- L'inode permettant notamment de connaître la taille des données à récupérer
- La taille de bloc ainsi que l'offset pour calculer les adresses

```
228 #Recupere les donnees a partir d une liste de pointeurs
229 def getDataFromPTR(self,ptrs,InodeDataEntry,level,blksize,blkOffset):
230     DATABUFF = ""
231     #Pour chaque pointeur
232     for i in range(0,len(ptrs)):
233         if level == 0: #Des que les pointeurs sont de niveau 0
234             if self.checkQNX6ptr(ptrs[i]): #On check que le pointeur soit valide
235                 if ptrs[i] != 0xffffffff and ptrs[i] != 0x0:
236                     if (InodeDataEntry['size']) >= 1024:
237                         buf = jarray.zeros( blksize, "b")
238                         self.devQNX6.read(buf,ptrs[i],blksize)
239                         DATABUFF += buf
240                     else:
241                         buf = jarray.zeros( (InodeDataEntry['size']), "b")
242                         self.devQNX6.read(buf,ptrs[i],(InodeDataEntry['size']))
243                         DATABUFF += buf
244                 else: #On lit les pointeurs qui ne sont pas de niveau 0 et on rappelle la fonction par re
245                     buf = jarray.zeros(blksize, "b")
246                     self.devQNX6.read(buf,ptrs[i],blksize) #On lit blocksize octets a partir de ptrs[i]
247                     newPTRS = unpack('<'+str(blksize/4)+'I', buf) #Un block contient blocksize/4 pointeurs
248                     level2_PTRS = []
249                     for i in range(0,len(newPTRS)):
250                         if self.checkQNX6ptr(newPTRS[i]):
251                             if newPTRS[i] != 0xffffffff and newPTRS[i] != 0x0:
252                                 level2_PTRS += [(newPTRS[i]*blksize)+blkOffset] #Calcul de l adresse
253                                 DATABUFF += self.getDataFromPTR(level2_PTRS,InodeDataEntry,level-1,blksize,blkOffset)
254
255     return DATABUFF
```

FIGURE 94 – Fonction récupérant les données à partir des 16 pointeurs contenu dans l'inode

De la même façon que pour récupérer les inodes à partir du Root Node, le système de fichiers utilise des blocs de pointeurs intermédiaires dans le cas où la taille des données est trop importante. La fonction getDataFromPTR utilise de la récursivité. Pour chaque pointeur, s'ils ont un niveau supérieur à 1, alors on les traite de façon à obtenir la liste de pointeurs du niveau juste en dessous et on rappelle la méthode getDataFromPTR par récursivité mais avec cette fois-ci, la nouvelle liste de pointeurs ET avec le "level" décrémenté de 1. Dans le cas où les pointeurs font référence à des blocs de pointeurs intermédiaires, ces blocs sont constitués de pointeurs de 4 octets les uns à la suite des

autres. On lit donc BLOCKSIZE/4 pointeurs de taille 4 par bloc de pointeurs intermédiaires.

```
buf = jarray.zeros(blksize, "b")
self.devQNX6.read(buf,ptrs[i],blksize) #On lit bloc
newPTRS = unpack('<'+str(blksize/4)+'I', buf) #Un b
level2_PTRS = []
for i in range(0,len(newPTRS)):
```

FIGURE 95 – Découpage du bloc en morceaux de taille 4 correspondants aux pointeurs

Si les pointeurs passés en paramètres de la fonction sont de niveau 0, alors on lit les blocs de données associés à ces pointeurs. Dans le cas où la taille des données est supérieure à la taille d'un bloc, on lit le bloc entièrement. Si la taille des données est inférieure à la taille du bloc, alors on lit seulement la taille des données nécessaires à partir du début du bloc. Pour chaque bloc de données, on ajoute les données à une variable DATABUFF qui contient les données du fichier.

8.2.7 Fonction récupérant les noms longs

Pour cette fonction, nous avons repris le code d'un programme python2 que nous avons trouvé. En effet, devant la difficulté à comprendre cette partie-là, nous avons préféré reprendre cette partie du code après avoir obtenu l'accord de son développeur. Voici les liens vers le programme :

<https://nop.ninja/> - Mathew Evans <https://github.com/ReFirmLabs/binwalk/issues/365>

Voici le code des fonctions permettant de récupérer les noms longs :

```
258 def getLongFileNames(self,superBlock):
259     longnames = []
260     for n in range(0, 16):
261         ptr = superBlock['Longfile']['ptr'][n]
262         if(self.checkQNX6ptr(ptr)):
263             ptrB = (ptr*superBlock['tailleBlock'])+superBlock['SB_end'];
264             longnames.append(self.parseQNX6LongFilename(ptr,superBlock['Longfile']['level'],superBlock))
265     ##Make Dictionary with all Names and INode/PTRs
266     count = 1
267     Dict = {}
268     for i in longnames:
269         if(i != None):
270             for q in i:
271                 if(q != None):
272                     Dict[count] = i[q]
273                     count = count + 1;
274     return Dict
275 #Fonction provenant de https://github.com/ReFirmLabs/binwalk/issues/365 - https://nop.ninja/ (Matthew)
276 def parseQNX6LongFilename(self,ptr_,level,blksize,blksoffset):
277     handle = jarray.zeros( 512, "b")
278     self.devQNX6.read(handle,(ptr_*blksize)+blksoffset,512)
279     LogFilenameNode={}
280     if level == 0:
281         size = unpack('<H',handle[0:2])
282         fname = unpack('<'+'str(size[0])+'B',handle[2:size[0]+2])
283         if(size[0] > 0):
284             LogFilenameNode[str(ptr_)] = str("".join("%c" % i for i in fname)).strip()
285             return LogFilenameNode
286         else:
287             return None
288     else:
289         Pointers = unpack('<128I', handle)
290         for i in range(0, 128):
291             if (self.checkQNX6ptr(Pointers[i]) != False):
292                 name = (self.parseQNX6LongFilename(Pointers[i],level-1,blksize,blksoffset))
293                 if name != None:
294                     if level >= 1:
295                         LogFilenameNode[str(Pointers[i])] = name[str(Pointers[i])]
296                     else:
297                         LogFilenameNode[str(Pointers[i])] = name
298     return LogFilenameNode
```

FIGURE 96 – Gestion des noms longs

Ces fonctions permettent de construire une liste des noms longs à partir du Root Node appelé "Longfile" et qui est situé dans le Super Block. De la même façon que pour récupérer les inodes, on lit à partir des 16 pointeurs contenus dans le Root Node, les différents niveaux de blocs de pointeurs intermédiaires jusqu'à obtenir les noms longs au niveau 0 (voir chapitre Longfile name). Dans le cas des pointeurs sur des blocs de pointeurs intermédiaires pour les Longfilename, seulement les 512 premiers octets du bloc contiennent des pointeurs. Puisque chaque pointeur est de taille 4, il y a donc 128 pointeurs par bloc de pointeurs. Une fois que l'on arrive à des pointeurs de niveau 0, chaque pointeur pointe sur un bloc contenant 1 seul nom long. On lit tout d'abord la

taille du nom constituant les deux premiers octets, puis à partir de la taille du nom, on lit le nom complet ne pouvant pas excéder 510 octets (voir chapitre Longfilename). La liste des Longfilename utilise le même système "d'ID" que pour les inodes. Le premier Longfilename parsé possède l'ID 1. Cela permet de retrouver le nom long correspondant à l'objet dans la méthode getDataInodeId(). En effet, dans cette méthode, si le nom est supérieur à 27 caractères, le nom ne sera pas directement disponible. A la place l'ID du nom long sera présent et le nom complet de l'objet pourra donc être récupéré grâce à la liste des noms longs construite par la fonction récupérant les noms longs à partir du Root Node "longfile".

8.2.8 Fonction : getDeletedContent

Cette fonction constitue la dernière méthode pour retrouver un fichier supprimé. Pour chaque inode, **s il s'agit d'un fichier supprimé ne se trouvant pas dans la liste des dépendances driTree**. On récupère alors les données et les métadonnées du fichier et on l'ajoute à une liste qui sera retournée par la suite au programme principal. Avec cette méthode, on ne peut plus retrouver le chemin ou le nom du fichier associé. Le fichier prend donc pour nom "deleted_InodeID" et pour chemin un dossier spécialement créé pour ce genre de fichiers et qui se situe dans le répertoire racine.

```
#Recupere les fichiers supprimes dont on ne peut plus recuperer le path et le name
def getDeletedContent(self,delFilesDirName,inodeTree,driTree,blksize=1024,blkOffset=0):
    deletedFiles = []
    for IEidx in inodeTree:
        IE = inodeTree[IEidx]
        if(IE != None and IE["status"] == 2 and IEidx not in driTree): #Si le status est égale à
            if(not self.InodeEntry_ISDIR(IE['mode'])): #Si c'est un fichier
                #On récupère la liste des pointeurs pointant vers les données
                PhysicalPTRs = []
                for pointer_index in IE['block_ptr']:
                    if pointer_index != 0xffffffff:
                        #On récupère la liste des pointeurs pointant vers les données
                        PhysicalPTRs += [(pointer_index*blksize)+blkOffset]
                #Recupere les données à partir des pointeurs
                data = self.getDataFromPTR(PhysicalPTRs,IE,IE['filelevels'],blksize,blkOffset)
                deletedFiles.append({'path':delFilesDirName,'name': str("deleted_") + str(IEidx) , 'data':data})
    return deletedFiles
```

Pour finir, nous avons aussi implémenté d'autres petites fonctions permettant notamment de vérifier les données. Par exemple la fonction checkQNX6ptr qui permet de vérifier que le pointeur est bien valide.

```

366     def checkQNX6ptr(self,ptr):
367         return not ((ptr+1) & ptr == 0 and ptr != 0)
368
369

```

FIGURE 97 – Vérification du pointeur

8.3 Interaction avec Autopsy

Maintenant que nous disposons des fonctions permettant de récupérer les différentes données du système de fichiers, il suffit d'appeler ces fonctions successivement dans la méthode process de l'ingest module et de les présenter à l'utilisateur via l'interface d'Autopsy. Tout d'abord, on initialise la barre de chargement dans autopsy et on fixe la tâche en cours à "Parsing Super Block". On crée ensuite dans le dossier du "Case" Autopsy un sous-dossier qui contiendra toutes les données extraites du système de fichiers.

```

80     def process(self, dataSource, progressBar):
81
82         #Permet de faire avancer la progress bar dans autopsy
83         progressBar.switchToDeterminate(100)
84         progressBar.progress("Parsing Super Block",10)
85         case = Case.getCurrentCase()
86         sKCase = case.getSleuthkitCase()
87         wDirPath = case.getModuleDirectory()
88
89         #Répertoire dans lequel extraire les données
90         realRootDir = wDirPath+"\\"+dataSource.getName()+"\\Partition0"
91         if(os.path.exists(realRootDir) == False):
92             try:
93                 os.makedirs(realRootDir)
94             except OSError as e:
95                 pass
96

```

FIGURE 98 – Méthode process de l'ingest module : Partie 1

On récupère ensuite le périphérique QNX6 correspondant à la data source ajouté par l'utilisateur et sur lequel ce dernier exécute l'ingest module. On rappelle qu'il s'agit d'un Data-source ingest module. L'objet récupéré est de type AbstractFile. On construit ensuite l'objet qnx6fs à partir de l'Abstract file correspondant au Data-Source. Cet objet qnx6fs va nous permettre de récupérer les différentes données au cours de l'analyse du système de fichiers. L'AbstractFile passé en paramètres est lu dans la classe QNX6_FS et lui permet d'accéder aux différentes données.

Pour récupérer les données à partir d'un Abstractfile voici comment on procède :

- On défini un buffer de type Java Array constitué de N "octets" : buff = jarray.zeros(N , "b")

-
- On lit les N octets à partir de l'adresse ADDR et on place les données dans le buffer créé précédemment : self.devQNX6.read(buff,ADDR,N)

```
#Recupere le diskimg au format AbstractFile
fileManager = case.getServices().getFileManager()
qnx6Img = fileManager.findFiles(dataSource, "%")[0]

#Construciton de l'objet QNX6 permettant de recuperer les infos du superblock ect ...
qnx6fs = QNX6_FS(qnx6Img, self._logger)

#Il faudrait prendre en compte la place occupe par la partition si il y en a
SBoffset = 0 + qnx6fs.QNX6_BOOTBLOCK_SIZE
#On recupere les informations du premier super block
SB = qnx6fs.readSuperBlock(SBoffset)
#La fin du super block 1 permet de calculer les addresses a partir des pointeurs se trouvant dans les inodes
SBendOffset = SB["SB_end"]
#Si il s'agit bien d'un FS QNX6
```

FIGURE 99 – Méthode process de l'ingest module : Partie 2

On calcule l'offset à partir duquel le Super Block commence et on appelle la méthode readSuperBlock() qui nous retourne le dictionnaire contenant les champs du Super Block. On appelle ensuite la méthode isQNX6FS pour s'assurer qu'il s'agit bien d'un système de fichiers QNX6. La fonction compare juste la valeur du champ SUPERBLOCK["magic"] avec 0x68191122 qui est le magic number des systèmes de fichiers QNX6.

```
#La fin du super block 1 permet de calculer les addresses a partir des pointeurs se trouvant dans les inodes
SBendOffset = SB["SB_end"]
#Si il s'agit bien d'un FS QNX6
if(qnx6fs.isQNX6FS(SB)):
    self.postMessage("QNX6 file system detected")

#Création d'un rapport contenant les informations du super blocks
self.createAndPostSBReport(dataSource.getName(), wDirPath + "\\..\\Reports", SB)
self.postMessage("File System report created")

#Identification du SuperBlock actif (Le super block ayant l'ID le plus grand est le super block actif)
#L'autre block est le backupSuperBlock qui peut être utile pour retrouver les données effacées
sndSBoffset = qnx6fs.getRndSBBlockOffset(SB)
sndSB = qnx6fs.readSuperBlock(sndSBoffset)
backUpSB = sndSB
if(qnx6fs.isQNX6FS(sndSB)):
    if(sndSB['serialNum'] > SB['serialNum']):
        backUpSB = SB
        SB = sndSB

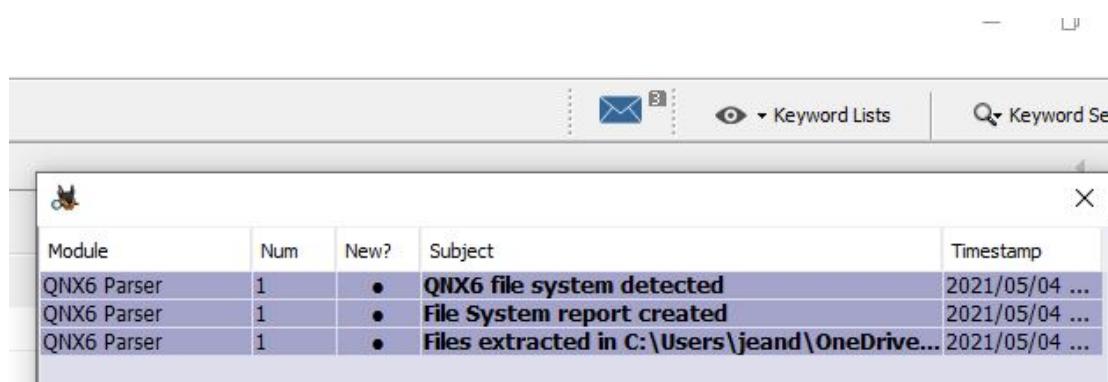
#Recuperation des inodes à partir des rootNodes du superBlock actif
progressBar.setProgress("Parsing inodes", 20)
inodeTree = qnx6fs.getInodesFromRootNodes(SB["RootNode"]['ptr'], SB["tailleBlock"], SBendOffset, SB["RootNode"]['level'])
```

FIGURE 100 – Méthode process de l'ingest module : Partie 3

S'il s'agit bien d'un système de fichiers QNX6, alors on crée un rapport au format texte sur le Super Block et qui est directement accessible dans Autopsy.
Voici à quoi ressemble le rapport :

```
-----diskimage9 QNX6FS Super Block informations-----  
  
Serial number : 0x2d  
Magic number : 0x68191122  
File system creation time : 04/30/2021, 11:24:23  
File system modification time : 04/30/2021, 11:24:23  
File system access time : 04/30/2021, 11:24:23  
Block Size : 1024 bytes  
Number of blocks : 0xfffff0  
Number of free blocks : 0xf6d91  
Number of inodes : 0x20000  
Number of free inodes : 0x1ffef
```

FIGURE 101 – Un rapport créé par l'ingest module



The screenshot shows the Autopsy interface with a message center window open. The window has a header with icons for envelope, Keyword Lists, and Keyword Search. Below the header is a list of notifications:

| Module | Num | New? | Subject | Timestamp |
|-------------|-----|------|--|----------------|
| QNX6 Parser | 1 | • | QNX6 file system detected | 2021/05/04 ... |
| QNX6 Parser | 1 | • | File System report created | 2021/05/04 ... |
| QNX6 Parser | 1 | • | Files extracted in C:\Users\jeand\OneDrive... | 2021/05/04 ... |

FIGURE 102 – Notifications dans Autopsy

Une fois que le rapport est créé, on notifie l'utilisateur de l'avancement de l'analyse en postant un message dans Autopsy. Ensuite, on récupère le second Super Block du système de fichiers. Pour récupérer l'offset de ce Super Block, il suffit d'appliquer la formule suivante :

TAILLE BLOC X NOMBRE BLOCS + INDICE DEBUT BLOCS où l'indice du début des blocs correspond à la fin du premier super bloc (0x3000 dans la plupart des cas).

Une fois que le programme a obtenu le second Super Block, il détermine lequel des deux blocs est le Super Block Actif et lequel est le Backup Super Block en comparant leurs numéros de séries ("serialNum").

```
#Recuperation des inodes a partir des rootNodes du superBlock actif
progressBar.progress("Parsing inodes",20)
inodeTree = qnx6fs.getInodesFromRootNodes(SB["RootNode"]['ptr'],SB["tailleBlock"],SBendOffset,SB['RootNode']['level'])

#Recuperation des inodes a partir des rootNodes du backup superBlock (utile pour retrouver les donnees effacees)
backUpInodeTree = qnx6fs.getInodesFromRootNodes(backUpSB["RootNode"]['ptr'],backUpSB["tailleBlock"],SBendOffset,backUpSB['RootNode']['level'])

#On recupere les inodes correspondant a des fichier dont le nom est long (traite differemment)
longNameObj = qnx6fs.getLongFileNames(SB)

#Recupere dans dirTree un dictionnaire contenant l id des dossiers et des fichiers ainsi que leurs noms et l id de leurs parents
progressBar.progress("Parsing directory structure",65)
dirTree = qnx6fs.getDirTree(inodeTree,backUpInodeTree,longNameObj,SB['tailleBlock'],SBendOffset)
```

FIGURE 103 – Méthode process de l'ingest module : Partie 4

Le programme récupère ensuite respectivement :

- La liste de tous les inodes à partir du Root Node se situant dans le Super Block Actif
- La liste de tous les inodes à partir du Root Node se situant dans le Backup Super Block
- La liste de tous les noms longs à partir du Root Node "longName"
- La liste des dépendances grâce aux différentes listes ci dessus

Toutes ces listes sont générées grâce aux méthodes situées dans la classe QNX6_FS et qui sont expliquées dans le chapitre précédent.

```
#On recupere la liste des fichiers et repertoire avec toutes les informations associees
progressBar.progress("Files and dirs recovery from inodes",80)
dirList,fileList = qnx6fs.getDirsAndFiles(inodeTree,dirTree,SB['tailleBlock'],SBendOffset)

#On cree un dossier special ou l on met les fichiers supprimees dont on a pas pu retrouver le path et le nom
retrievedContentDirName = "retrieved_content//"
dirPath = realRootDir+"\\"+retrievedContentDirName
if(not os.path.exists(dirPath)):
    try:
        os.makedirs(dirPath)
    except OSError as e:
        self.postMessage("Erreur lors de la creation de : "+dirPath )
#On recupere les fichiers supprimees dont on a pas pu retrouver le path et le nom
deletedContent = qnx6fs.getDeletedContent(retrievedContentDirName,inodeTree,dirTree,SB['tailleBlock'],SBendOffset)
```

A partir de la liste des dépendances "dirTree" et la liste des inodes "inodeTree", on récupère enfin la liste des dossiers et des fichiers où chaque tuple est dictionnaire de la forme :

Path : perso//privée// | Name : p.txt | Size : 51 | UID : 0 | GID : 0 | ftime : 1619775912
| atime : 1619775912 | ctime : 1619775921 | mtime : 1619775921 | status : 3

Enfin, on récupère une liste similaire aux deux premières grâce à la méthode getDeletedContent pour les fichiers supprimés mais dont on n'a pas pu retrouver le chemin et le nom. Ces fichiers sont placés dans un répertoire créé spécialement à la racine appelé "retrieved_content".

A partir de ces listes d'objets, on génère l'arborescence et les fichiers dans un répertoire du "Case" Autopsy. On crée d'abord tous les dossiers récupérés puis tous les fichiers se trouvant dans les dossiers créés précédemment.

```
#On cree les dossiers retrouves dans un repertoire du projet
progressBar.progress("Creation of recovered files and dirs",90)
for rep in dirList:
    dirPath = realRootDir+"\\"+os.path.join(rep["path"],rep["name"])
    if(not os.path.exists(dirPath)):
        try:
            os.makedirs(dirPath)
        except OSError as e:
            self.postMessage("Erreur lors de la creation de : "+dirPath )
            self.log(Level.INFO, os.strerror(e.errno))
        pass

#On cree les fichiers retrouves dans un repertoire du projet
for file in fileList+deletedContent:
    filePath = realRootDir+"\\"+os.path.join(file["path"],file["name"])
    if(not os.path.exists(filePath)):
        try:
            f = open(filePath, "wb+")
            if(file["data"] != None):
                f.write(file["data"])
            f.close()
        except IOError as e:
            self.postMessage("Erreur lors de la creation de : "+filePath )
            self.log(Level.INFO, os.strerror(e.errno))
        pass
```

FIGURE 104 – Génération de l'arborescence extraite

Enfin, on réalise différentes opérations pour notifier et informer l'utilisateur à travers l'interface d'Autopsy. Puis on appelle la fonction addTree permettant d'ajouter l'arborescence créée du système de fichiers sur l'interface Autopsy. Cela permet à l'utilisateur de directement pouvoir accéder à l'arborescence générée et de consulter les fichiers :

The screenshot shows the Autopsy interface. On the left, the file tree displays a 'diskimage9' source containing a 'Partition0 (11)' volume with various folders like '.boot', 'attestation.png', 'deleted-m1', 'imagevbin.png', 'logo.jpg', 'notes', 'perso', 'photos', 'presentation.txt', 'retrieved_content', and 't.txt'. Other sources listed are 'imgqnx7', 'Views', 'File Types', 'Deleted Files', 'MB File Size', 'Results', 'Extracted Content', and 'Keyword Hits'. On the right, a table view titled 'Table' shows the same list of files with columns for Name, S, C, O, and Modified Time.

| Name | S | C | O | Modified Time |
|-------------------|---|---|---|----------------|
| .boot | | | | 2021-05-04 11: |
| attestation.png | | | | 2021-05-04 11: |
| deleted-m1 | | | | 2021-05-04 11: |
| imagevbin.png | | | | 2021-05-04 11: |
| logo.jpg | | | | 2021-05-04 11: |
| notes | | | | 2021-05-04 11: |
| perso | | | | 2021-05-04 11: |
| photos | | | | 2021-05-04 11: |
| presentation.txt | | | | 2021-05-04 11: |
| retrieved_content | | | | 2021-05-04 11: |
| t.txt | | | | 2021-05-04 11: |

FIGURE 105 – Affichage de l’arborescence extraite

L’utilisateur peut, à travers l’interface Autopsy, facilement naviguer entre les dossiers et les fichiers extraits du système de fichiers QNX6. Cela permet notamment à l’utilisateur de, par la suite exécuter, d’autres ingest modules sur les fichiers extraits pour réaliser d’autres types d’opérations. (Par exemple récupérer certains types de fichiers). Dans ce cas-là, après avoir extrait les fichiers et dossiers du périphérique grâce à notre ingest module, on a exécuté un autre ingest module permettant de trier les fichiers selon leurs types :

The screenshot shows the Autopsy interface. The file tree on the left is identical to Figure 105. On the right, a table view titled 'EXIF Metadata' shows two entries: 'image7.jpg' and 'image6.jpg'. This indicates that the 'EXIF Metadata' ingest module has been run on the previously extracted files.

| Source File | S | C |
|-------------|---|---|
| image7.jpg | | |
| image6.jpg | | |

FIGURE 106 – Exécution d’un autre ingest module sur les fichiers extraits par notre ingest module

Les données extraites par notre ingest module sont ensuite triées par un autre ingest module disponible de base dans Autopsy. D’autres d’opérations peuvent être exécutées

sur les données extraites par notre ingest module grâce aux nombreux ingest modules développés par la communauté.

Voici la fonction addTree qui ajoute l'arborescence et les fichiers générés dans Autopsy :

```
#Ajoute le contenu d un repertoire dans le datasource d autopsy
def addTree(self,path,parent):
    sCase = Case.getCurrentCase().getSleuthkitCase()
    for f in os.listdir(path):
        fpath = os.path.join(path, f)
        if os.path.isfile(fpath):
            sCase.addLocalFile(f,fpath,os.path.getsize(fpath), Long=True)
        if os.path.isdir(fpath):
            rep = sCase.addLocalFile(f,fpath,os.path.getsize(fpath), Long=True)
            self.addTree(fpath,rep)
```

Le programme parcourt récursivement l'arborescence et les fichiers générés extraits du périphérique QNX6 et les ajoute à l'interface Autopsy via la méthode addLocalFile(). Cela permet donc de visualiser l'arborescence et les fichiers générés directement dans Autopsy. Pour chaque répertoire, on récupère son ID donné par Autopsy via la méthode addLocalFile() permettant par la suite de lier les enfants du dossier à ce même dossier. Pour finir, le programme crée un rapport contenant toutes les métadonnées, telles que la date de création de tous les fichiers et dossiers :

```
|-----diskimage9 QNX6FS Content Report-----
```

-----Directories Extracted-----

```
Path : | Name : photos | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:24:59 | atime : 04/30/2021,
Path : perso// | Name : privee | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:45:02 | atime : 04/30/2021,
Path : | Name : .boot | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:24:23 | atime : 04/30/2021, 11:24:23,
Path : | Name : perso | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:23 | atime : 04/30/2021, 11:37:23,
Path : | Name : notes | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:25:05 | atime : 04/30/2021, 11:25:05,
Path : | Name : deleted-m1 | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:35 | atime : 04/30/2021, 11:38:35
```

-----Files Extracted-----

```
Path : | Name : presentation.txt | Size : 53 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:25:18 | atime : 04/30/2021, 11:25:18,
Path : perso//privee// | Name : p.txt | Size : 51 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:45:12 | atime : 04/30/2021, 11:45:12,
Path : | Name : attestation.png | Size : 3244517 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:29:30 | atime : 04/30/2021, 11:29:30,
Path : photos// | Name : casBetaElevee.png | Size : 13571 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:57 | atime : 04/30/2021, 11:37:57,
Path : | Name : t.txt | Size : 928 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:07 | atime : 04/30/2021, 11:38:07,
Path : notes// | Name : notesCrypto.txt | Size : 5158 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:39 | atime : 04/30/2021, 11:37:39,
Path : | Name : imagevbin.png | Size : 145267 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:20 | atime : 04/30/2021, 11:38:20,
Path : perso// | Name : rapport.pdf | Size : 683636 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:59 | atime : 04/30/2021, 11:38:59,
Path : | Name : logo.jpg | Size : 24890 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:49 | atime : 04/30/2021, 11:38:49,
Path : deleted-m1// | Name : deleted-testAnglais.png | Size : 266607 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:42:47 | atime : 04/30/2021, 11:42:47,
Path : retrieved_content// | Name : deleted_9 | Size : 41 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:42:47 | atime : 04/30/2021, 11:42:47,
Path : retrieved_content// | Name : deleted_10 | Size : 36 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:46:44 | atime : 04/30/2021, 11:46:44,
```

FIGURE 107 – Rapport contenant les métadonnées des fichiers et des dossiers

Enfin, nous avons aussi développé d'autres fonctions, notamment pour générer des rapports dans Autopsy, qu'il n'est pas pertinent de présenter ici. Nous avons testé l'ingest module sur plusieurs cas différents. Le programme est bien capable de récupérer les

noms longs ainsi que les fichiers et dossiers supprimés. Si un dossier contenant d'autres dossiers ou fichiers a été supprimé, alors l'ingest module est capable de reconstruire toutes l'arborescence supprimée. Nous avons également testé l'ingest module pour un périphérique QNX6 contenant des fichiers de grandes tailles et l'entièreté des données a été récupéré. De plus, nous avons essayé plusieurs configurations de système de fichiers QNX6 en modifiant les paramètres de la commande "mkqnx6fs" lors de la création du fichier servant de block device et tout a fonctionné correctement. Enfin, pour les tests effectués, l'ingest module a toujours généré la bonne arborescence telle qu'elle avait été créée sur la machine QNX6.

Voici le résultat de l'exécution de l'ingest module sur plusieurs périphériques QNX6, pour chaque périphérique, des rapports sont également créés :

The screenshot shows a forensic analysis interface with a sidebar for 'Data Sources' and a main 'Listing' pane. The 'diskimage9' source contains 'Partition0 (11)' which includes 'deleted-m1 (1)', 'notes (1)', 'perso (2)', 'privée (1)', 'photos (1)', and 'retrieved_content (2)'. The 'imgpx5' source contains 'Partition0 (8)' with 'dossier1 (3)', 'dossier11 (2)', 'dossier12 (2)', 'm1 (1)', and 'notes (2)'. The 'imgpx8' source contains 'Partition0 (9)' with 'dossier1 (3)', 'dossier11 (2)', 'dossier12 (2)', 'm1 (1)', 'notes (2)', and 'retrieved_content (3)'. The 'qnxdisk2' source contains 'Partition0 (6)' with 'rvdossier (1)', 'dir2 (1)', and 'retrieved_content (1)'. The 'Results' section includes 'Extracted Content', 'Keyword Hits' (with 'Single Literal Keyword Search (0)' and 'Single Regular Expression Search (0)'), and 'Hashset Hits'.

Table Headers:

| Name | S | C | O | Modified Time | Change Time | Access Time | Created Time | Size | Flags(Dir) | Flags(Meta) | I |
|------|---|---|---|---------------|-------------|-------------|--------------|------|------------|-------------|---|
|------|---|---|---|---------------|-------------|-------------|--------------|------|------------|-------------|---|

Sample Data from Table:

| | | | | | | | | | | | |
|-------------------|--|--|--|--------------------------|--------------------------|--------------------------|--------------------------|---------|-----------|-----------|---|
| boot | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| attestation.png | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 3245056 | Allocated | Allocated | L |
| deleted-m1 | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| imagevbin.png | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 145408 | Allocated | Allocated | L |
| logo.jpg | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 25600 | Allocated | Allocated | L |
| notes | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| perso | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| photos | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| presentation.txt | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 53 | Allocated | Allocated | L |
| retrieved_content | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 0 | Allocated | Allocated | L |
| t.txt | | | | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 2021-05-04 11:35:36 CEST | 928 | Allocated | Allocated | L |

FIGURE 108 – Résultat de l'exécution de l'ingest module sur plusieurs périphériques QNX6

9 Guide d'utilisation

Vidéo de présentation et d'utilisation de l'ingest module :

https://www.youtube.com/watch?v=H9FppPDLrpY&ab_channel=JeandeBonfils

9.1 Installation

Autopsy ainsi que l'ingest module développé fonctionne uniquement sous le système d'exploitation Windows. L'ingest module peut être récupéré à l'adresse suivante :

<https://github.com/jdbonfils/QNX6-Files-System-Reader-Ingest-Module>

Une fois récupéré, placez le dossier de l'ingest module dans le dossier :
"AppData/Roaming/autopsy/python_modules"

C'est dans ce dossier que doit être installé tous les "3rd party ingest module". Au lancement d'Autopsy, le logiciel devrait automatiquement repérer le nouvel ingest module ajouté.

9.2 Utilisation

Au lancement d'Autopsy, créez un nouveau projet en cliquant sur "New case" puis saisissez les informations demandées et enfin cliquez sur "Terminer". Autopsy vous demande ensuite de choisir un type de "Data Source à ajouter". Il faut choisir "**Unallocated Space Image File**"

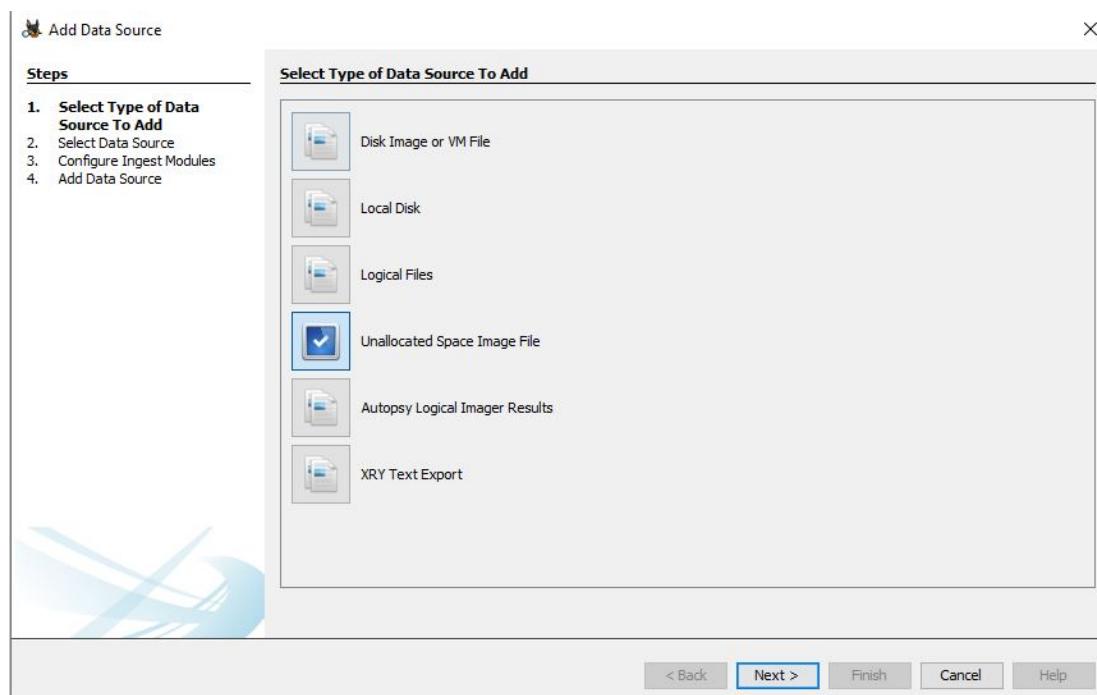


FIGURE 109 – Ajout de la source de données

Choisissez ensuite la source de données correspondant au périphérique QNX à analyser et cochez "Do not break up" en dessous :

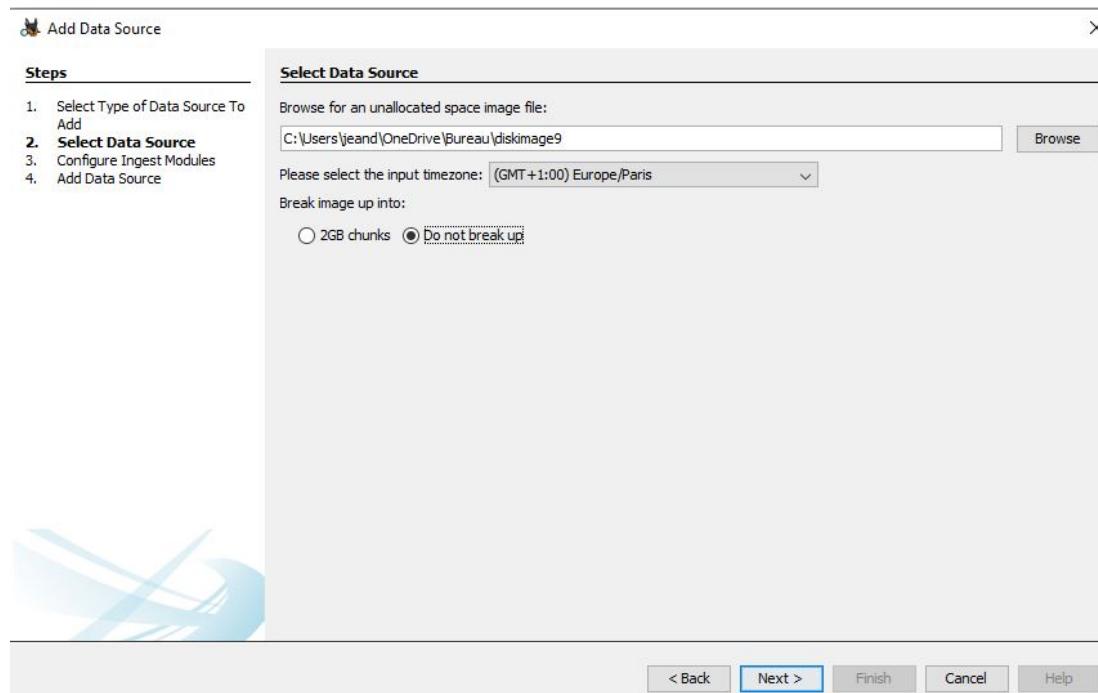


FIGURE 110 – Ajout de la source de données

La liste des ingest modules à lancer au démarrage s'affiche. Choisissez l'ingest module "QNX6 Parser" qui correspond au module développé. Enfin cliquez sur "Terminer". Le module devrait s'exécuter. Une barre de chargement en bas à droite s'affiche indiquant la tâche en cours et son avancement. Une fois que l'ingest module a terminé de s'exécuter, des messages apparaissent donnant un feedback sur l'exécution du module.

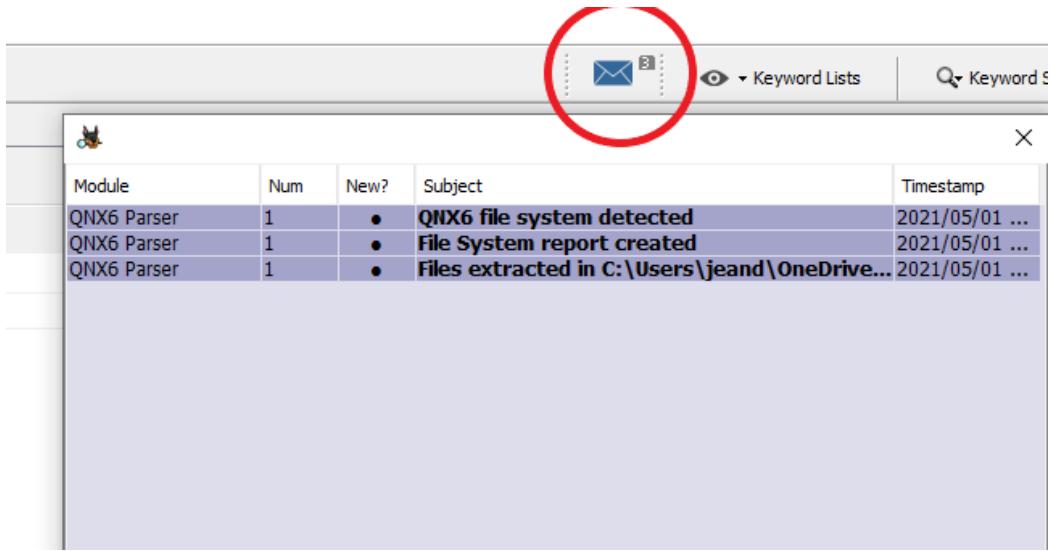


FIGURE 111 – Message donnant un feedback

Des rapports sur le périphérique QNX6 sont créés. Le premier contient les informations du Super Block et le second contient les informations sur la liste des données extraites comme les répertoires, les dossiers ainsi que ceux supprimés. Ces rapports peuvent être consultés en cliquant sur l'onglet "Report" puis en double cliquant sur l'un des rapports en question.

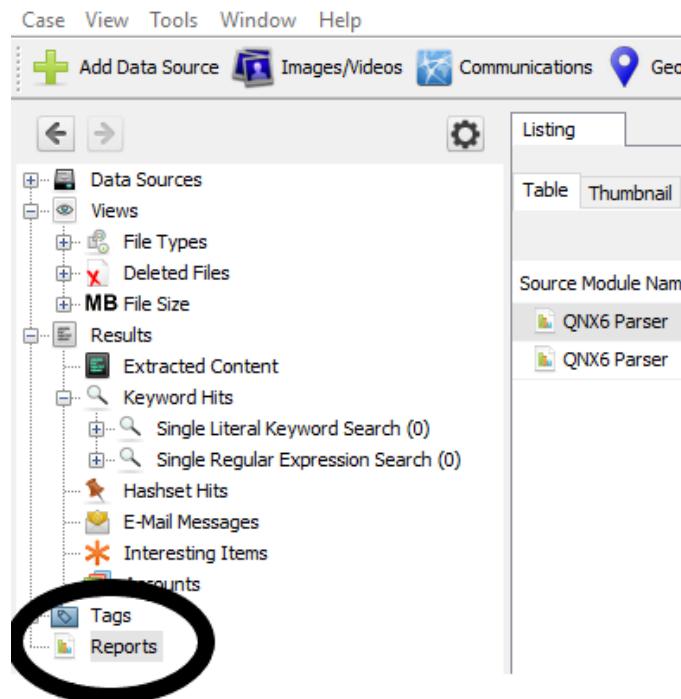


FIGURE 112 – Consultation des rapports

Voici un exemple de rapport créé :



```
diskimage9SuperBlockReport.txt - Bloc-notes
Fichier Edition Format Affichage Aide
-----diskimage9 QNXFS Super Block informations-----
Serial number : 0x2d
Magic number : 0x68191122
File system creation time : 04/30/2021, 11:24:23
File system modification time : 04/30/2021, 11:24:23
File system access time : 04/30/2021, 11:24:23
Block Size : 1024 bytes
Number of blocks : 0xfffff0

diskimage9ContentReport.txt - Bloc-notes
Fichier Edition Format Affichage Aide
-----diskimage9 QNXFS Content Report-----

-----Directories Extracted-----
Path : | Name : photos | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:24:59 | atime : 04/30/2021, 11:45:42 | ctime : 04/30/2021
Path : perso// | Name : privee | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:45:02 | atime : 04/30/2021, 11:45:20 | ctime : 04/
Path : | Name : .boot | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:24:23 | atime : 04/30/2021, 11:24:23 | ctime : 04/30/2021,
Path : | Name : perso | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:23 | atime : 04/30/2021, 11:45:04 | ctime : 04/30/2021,
Path : | Name : notes | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:25:05 | atime : 04/30/2021, 11:44:11 | ctime : 04/30/2021,
Path : | Name : deleted-m1 | Size : 1024 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:35 | atime : 04/30/2021, 11:49:31 | ctime : 04/30/2021

-----Files Extracted-----
Path : | Name : presentation.txt | Size : 53 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:25:18 | atime : 04/30/2021, 11:43:24 | ctime : 04
Path : perso// | Name : p.txt | Size : 51 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:45:12 | atime : 04/30/2021, 11:45:12 | ctime
Path : | Name : attestation.png | Size : 3244517 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:29:30 | atime : 04/30/2021, 11:29:07 | ctime
Path : photos// | Name : casBetaElevee.png | Size : 13571 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:57 | atime : 04/30/2021, 11:35:40 |
Path : | Name : t.txt | Size : 928 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:07 | atime : 04/30/2021, 11:32:27 | ctime : 04/30/2021,
Path : notes// | Name : notesCrypto.txt | Size : 5158 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:37:39 | atime : 04/30/2021, 11:44:20 | ct
Path : | Name : imagevbnn.png | Size : 145267 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:20 | atime : 04/30/2021, 11:34:48 | ctime : 0
Path : perso// | Name : rapport.pdf | Size : 683636 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:59 | atime : 04/30/2021, 11:31:01 | ctim
Path : | Name : logo.jpg | Size : 24890 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:38:49 | atime : 04/30/2021, 11:31:28 | ctime : 04/30/2
Path : deleted-m1// | Name : deleted-testAnglais.png | Size : 266607 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:39:51 | atime : 04/30/2021,
Path : retrieved_content// | Name : deleted_9 | Size : 41 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:42:47 | atime : 04/30/2021, 11:42:47 |
Path : retrieved_content// | Name : deleted_10 | Size : 36 | UID : 0 | GID : 0 | ftime : 04/30/2021, 11:46:44 | atime : 04/30/2021, 11:46:44
```

FIGURE 113 – Consultation des rapports

Ces rapports nous donnent des informations très utiles sur le périphérique, les fichiers et les dossiers. Il indique notamment la date du dernier accès, la date de modification, la date de création ou encore la taille d'un fichier ou d'un répertoire. Les fichiers ou répertoires ayant un nom commençant par "deleted-" correspondent au fichier ou dossier qui avait été supprimé mais que l'on a pu retrouver.

Les fichiers se trouvant dans le dossier "retrieved_content" correspondent au fichier dont on a pu retrouver les données mais pas le chemin ni le nom.

9.3 Visualisation de l'arborescence générée

Enfin, on peut visualiser tous les dossiers et fichiers récupérés sous la forme d'une arborescence de la même façon qu'ils sont stockés sur le système d'exploitation QNX.

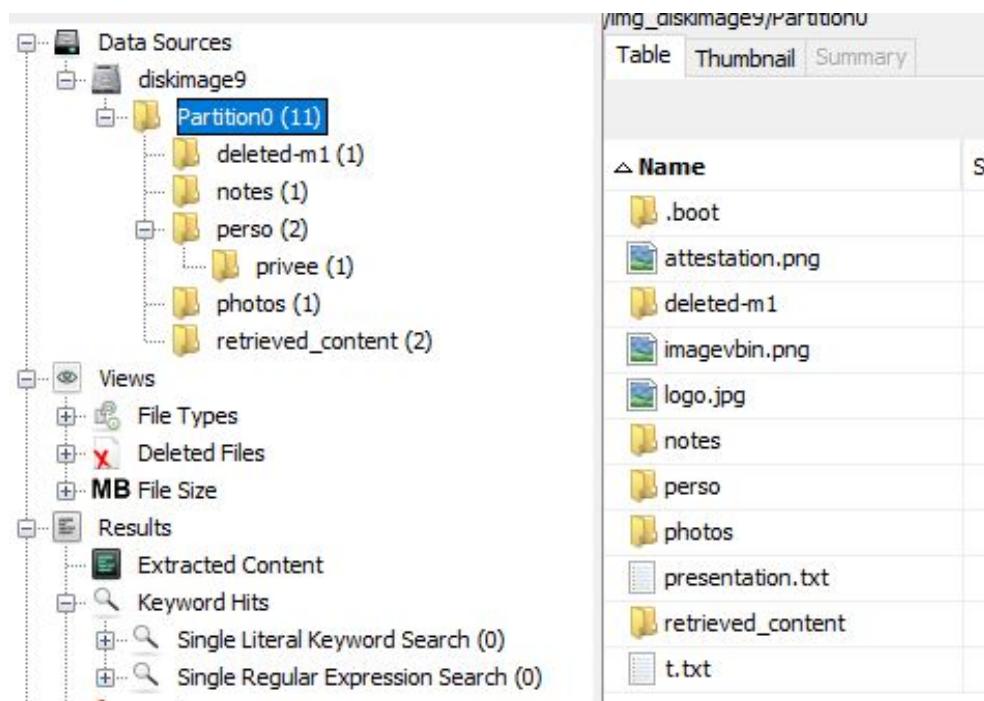


FIGURE 114 – Affichage de l’arborescence extraite

On peut facilement naviguer entre les dossiers et les fichiers retrouvés par l’ingest module. L’ingest module recrée l’arborescence telle qu’on la verrait si on faisant un "mount -t qnx6..." du périphérique sur une machine QNX. En plus de cela, l’ingest module recrée aussi les dossiers et fichier qui avait été supprimés et que l’on a pu retrouver (leurs noms commencent par deleted-). Évidemment, tous les fichiers et dossiers supprimés ne peuvent pas forcément être récupérés. Cependant, une certaine partie des dossiers et fichiers supprimés reste encore enregistrée pendant un certain temps et peut être retrouvée (voir ch. Récupération de données supprimées). Un dossier retrieved_content contient les fichiers supprimés dont on n’a pas pu retrouver le nom et le chemin.

The screenshot shows the Autopsy interface. On the left, the file tree displays a 'diskimage9' source containing 'Partition0 (11)' which includes 'deleted-m1 (1)', 'notes (1)', 'perso (2)', 'privée (1)', 'photos (1)', and 'retrieved_content (2)'. Below this are sections for 'Views' (File Types, Deleted Files, MB File Size), 'Results' (Extracted Content, Keyword Hits, Single Literal Keyword Search (0), Single Regular Expression Search (0), Hashset Hits, E-Mail Messages, Interesting Items, Accounts), 'Tags', and 'Reports'. The main area shows a 'Listing' table for '/img_diskimage9/Partition0/perso' with two entries:

| Name | S | C | O | Modified Time | Change Time | Access Time | Created Time | Size | Flags |
|-------------|---|---|---|--------------------------|--------------------------|--------------------------|--------------------------|--------|-----------|
| privée | | | | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 0 | Allocated |
| rapport.pdf | | | | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 2021-05-01 13:12:29 CEST | 684032 | Allocated |

Below the table is a toolbar with various icons for file operations. The bottom right corner features the University of Limoges logo and text: 'UNIVERSITÉ DE LIMOGES', 'FACULTÉ DES SCIENCES ET TECHNIQUES', 'MASTER 1 CRYPTIS', and 'Rapport de Droit et Conduite de Projet'.

FIGURE 115 – Affichage de l’arborescence extraite

Certains types de fichiers pris en charge par Autopsy comme les PNG, PDF ou encore JPG peuvent être directement consultés dans le logiciel. Il est aussi possible d’exporter l’arborescence générée par l’ingest module. Pour cela, il faut faire un "clique droit" sur le dossier à exporter puis "Extract Files". A la fin de l’exécution de l’ingest module, l’arborescence générée est automatiquement créée dans le dossier :
Documents/nomProjet/ModuleOutput/nomDataSource

Bien sûr, plusieurs data sources peuvent être ajoutées au "Case" et d’autres ingest modules peuvent être lancés en parallèle.

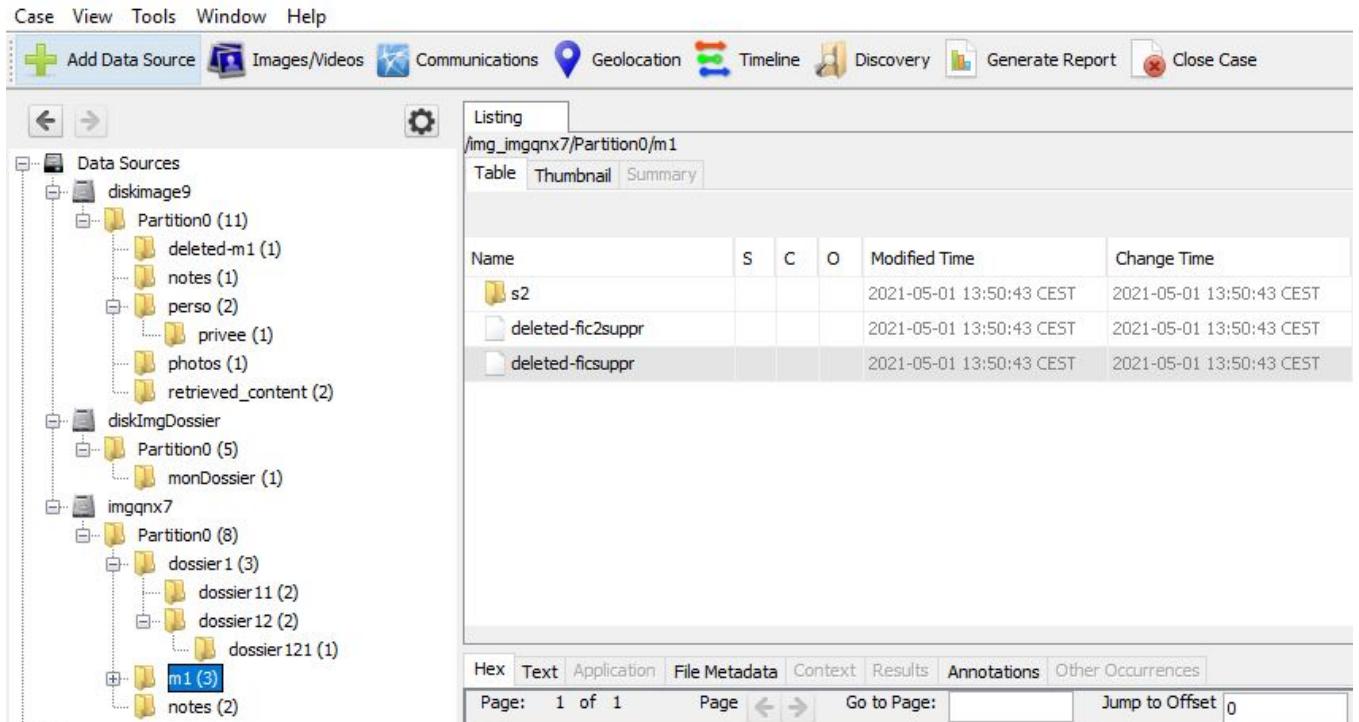


FIGURE 116 – Analyse de data sources multiples

Toute l'arborescence que ce soit les dossiers, les dossiers supprimés, les fichiers et les fichiers supprimés sont entièrement retrouvés grâce à l'ingest module. Si l'on montait le périphérique sur une machine QNX, on obtiendrait la même arborescence avec le contenu supprimé en moins.

9.4 Résultats obtenus

L'ingest module développé lors de ce projet est capable de récupérer tous les fichiers auxquels on pourrait accéder via une machine QNX. En plus de cela, certains fichiers supprimés peuvent être retrouvés. Comme vu précédemment, il existe dans le système de fichiers plusieurs niveaux de suppression jusqu'à la suppression totale des données d'un fichier. L'ingest module est donc capable de récupérer d'une part certains fichiers supprimés avec leurs noms ainsi que leur chemin et d'autre part, seulement les données pour certains fichiers. De plus, l'ingest module récupère également les fichiers possédants des noms longs. En effet, les fichiers ou dossiers avec cette particularité sont gérés dans le système de fichiers différemment. Pour finir, les fichiers sont à chaque fois récupérés en entier, même ceux de grande taille où les données sont réparties sur plusieurs blocs différents. Après plusieurs tests, il semble bien que l'ingest module soit capable de reconstituer toute l'arborescence ainsi que les fichiers tels qu'on les verrait sur une machine QNX (à cela, sont ajoutés les fichiers supprimés). Les informations générées dans les rapports telles que la date de modification semblent aussi correctes après plusieurs tests.

10 Gestion de Projet

Nous allons dans cette partie détailler la manière dont on s'est organisés pour mener à bien notre projet. Conscients au départ que ce projet nécessitait beaucoup de recherches et de curiosité afin de cerner et comprendre les spécifications et caractéristiques du système d'exploitation QNX, nous avions commencé à effectuer ces recherches dès la fin du premier semestre. Au début du deuxième semestre et après quelques semaines de recherche, nous avions effectué une réunion avec M. Maxime SAUVAYRE via Google Meet pour lui poser nos différentes questions et pour savoir ce qui nous été réellement demandé. Après cette réunion, il nous a fourni un certain nombre de documents et la machine virtuelle permettant de lancer le système d'exploitation QNX. Après cela, nous nous sommes mis d'accord pour effectuer des réunions virtuelles hebdomadaires (généralement le lundi ou mardi de chaque semaine) pendant lesquelles on se donnait des directives que chacun devait exécuter pour la semaine d'après. Plusieurs réunions ont aussi été organisées avec M. Damien SAUVERON via Discord pour lui montrer l'état d'avancement de notre projet et lui soumettre plusieurs questions dont on ne trouvait pas de réponse sur Internet et notamment des questions qui apparaissaient au fur et à mesure de l'avancement du projet.

Nous allons par la suite détailler l'ensemble du travail réalisé semaine par semaine dans un diagramme de Gant.

10.1 Planning réel et répartition des tâches

Ci-dessous, un diagramme de Gantt qui résume l'ensemble des travaux réalisés pendant toute l'année universitaire. Le diagramme de Gantt contient le résumé et les directives fixées lors des onze réunions qui ont eu lieu :

| Réunions | 26/01/2021 | 02/02/2021 | 09/02/2021 | 23/02/2021 | 02/03/2021 |
|---------------------------------|--|--|---|--|------------|
| Pour tous les membres du groupe | Installation de la machine virtuelle QNX Etudier les différents fichiers de une solution pour windows) (fs-qnx6.so ...) Installation et prendre en main IDA ou GUIDRA, regarder des tutos sur IDA , GUIDRA Revoir l'assembleur | Recherche du fonctionnement et de l'organisation des systèmes de fichiers et reverse sur les autres systèmes de fichiers.+ IDA | Installer Ghidra complètement Regarder/Lire les liens envoyés par M. Sauveron | Chercher le liens entre mount utilisé dans mkqnx6fs et le mount de fs-qnx6.so | |
| JEAN | Envoyer un mail à M Sauveron pour une réunion | Chercher sur QNX et Internet des infos sur mkqnx6fs Poser la question sur les forums | Comprendre (traduire les variables + commenter le code) du main de mkqnx6fs et les fonctions associées au main | commenter, comprendre, renommer les vars sur les fonctions utilisées par mkqnx6fs et trouver les fonctions utilisées provenant de fs-qnx6.so | |
| MALIK | | | | | |
| LISE | Regarder un moyen de tracer une commande sur QNX Trace logger | Comprendre (traduire les variables + commenter le code) de fs-qnx6::mount et fs-qnx6::q6_superblock_inoist | | | |
| MATHIS | | | | | |



| Réunions | 16/03/2021 | 23/03/2021 | 30/03/2021 | 06/04/2021 | 13/04/2021 |
|---------------------------------|---|---|--|---|--|
| Pour tous les membres du groupe | | | | | |
| JEAN | Rapport sur la structure d'un inode, comprendre la structure d'un inode et comment on retrouve les données à partir des indices et des root nodes | Finir le schéma des inodes pour retrouver les données (dossier, fichiers bitmap), commencer l'ingest module, envoyer message à M Sauveron | Continuer le code de l'ingest module(essayer de récupérer les données) | Continuer le code de l'ingest module(essayer de récupérer les données supprimées) | Continuer le code (retrouver les données effacées) |
| MALIK | Ajouter des informations sur les différents types de fichiers UNIX, et lire et faire un rapport sur les informations Faire une partie sur les partitions QNX, MBR ,etc. | Faire une partie sur les partitions QNX, MBR, Faire plus de recherches Faire une partie sur les partitions QNX sur le répertoire /dev | recherche sur partitions MBR,GPT,,, voir avec Lise parties à chercher sur QNX(pour demain) | Rechercher comment Autopsy parse les autres SF | Faire une partie détaillée sur le reverse |
| LISE | | | | | |
| MATHIS | | | | | |

| Réunions | 16/03/2021 |
|---------------------------------|---|
| Pour tous les membres du groupe | |
| JEAN | Rédaction du rapport du projet Recupération des noms longs |
| MALIK | Rédaction du rapport du projet |
| LISE | |
| MATHIS | |

FIGURE 119 – L'ensemble des réunions réalisées à partir du 21/04/2021

10.2 Réunions

Nous avions organisé des réunions virtuelles via Discord le lundi ou mardi de chaque semaine depuis le 26/01/2021 pour nous fixer des objectifs à réaliser pour la semaine d'après. Nous avions réalisé onze réunions depuis le début du projet. L'ensemble des tâches qui ont été réalisées ont été recensées dans le diagramme de Gantt illustré dans la partie précédente. Le fait d'organiser des réunions hebdomadaires nous a permis de bien avancer et de mener à bien notre projet.

10.3 Organisation

Compte tenu des conditions sanitaires actuelles, toutes nos réunions ont été réalisées via Discord. Nous annoncions à chaque fois qu'une réunion était prévue pour nous assurer que nous étions tous disponibles au créneau prévu pour la réunion. Le jour prévu pour la réunion, chacun d'entre nous présentait le travail qu'il avait fait et les recherches qu'il avait menées aux autres étudiants. A chaque fois qu'on trouvait des questions auxquelles on n'avait pas de réponses, on contactait M. Damien SAUVERON pour les lui soumettre directement.

10.4 Imprévus

Malheureusement et compte tenu des conditions sanitaires actuelles, nous étions au départ tous censés travailler sur notre projet mais la majorité du projet n'a été réalisé que par deux personnes du groupe (Jean DE BONFILS LAVERNELLE et Malik CHOUGAR). Nous avions donc dû donc faire face à une grosse charge de travail. Nous n'avions aussi malheureusement pas pu organiser de réunion en présentiel entre nous ou avec M. Damien SAUVERON : l'ensemble de nos réunions a été organisé via Discord.

10.5 Grille de participation

| Nom du groupe | | | | | Production | | |
|-----------------------|--------|---------------------------|----------------------|------------------------------|------------|---------------|------------------------------|
| Etudiant | | Evaluation sur la période | Coordination des RDV | Participation active aux RDV | Code | Rapport Final | Préparation de la soutenance |
| Nom | Prénom | | | | | | |
| de Bonfils Lavernelle | Jean | | 5 | 5 | 5 | 5 | X |
| Chougar | Malik | | 5 | 5 | 5 | 0 | X |
| Matet | Lise | | | | | | X |
| Greau | Mathis | | | | | | X |

FIGURE 120 – Grille de participation demandée

11 Conclusion

Pour conclure, le projet nous a permis de nous familiariser avec le système d'exploitation QNX qui est très peu référencé et documenté sur le WEB. Nous pensons également que l'ensemble du travail demandé a été bien mené et que les objectifs fixés par les professeurs ont été concrétisés. L'étude du système de fichiers QNX 6 s'est avérée beaucoup plus compliquée que prévue. La documentation sur QNX étant très faible, il n'a pas été facile de comprendre son fonctionnement. La majeure partie du travail aura été au travers de différent moyen, de comprendre le fonctionnement du système de fichiers. QNX est un système très fermé et l'accès à la documentation y est très restreint. L'apprentissage de la librairie "The Sleuth Kit" a aussi été compliqué. En effet, cette librairie possède énormément de classes et de concepts dont il est difficile de distinguer les utilités. De plus, il n'existe pratiquement aucun tutoriel sur le développement d'ingest module et sur l'utilisation de la librairie "The Sleuth Kit". Enfin, la rédaction du rapport s'est avérée être une tâche beaucoup plus complexe que prévue. Il est très difficile d'expliquer de manière textuelle le fonctionnement du système de fichiers d'autant plus que nous connaissons pas les termes adaptés au système de fichiers QNX6. IL est donc possible que certains termes utilisés ne soient donc pas rigoureusement corrects. Cependant ce projet nous aura permis de comprendre des concepts jusqu'alors totalement inconnus comme les inodes et de s'initier à de nouvelles méthodes d'analyse telles que le reverse engineering.

Ci-dessous, les conclusions réalisées par chaque étudiant :

- Malik CHOUGAR : Je me suis pour ma part mis au travail depuis le début du projet. Je me suis en grande partie occupé de la recherche et de la rédaction du rapport. Ce projet m'a permis d'apprendre de nouvelles notions concernant les systèmes d'exploitation en général mais surtout concernant le système d'exploitation QNX qui rappelons-le est très peu documenté et référencé sur Internet. Je pense que malgré les difficultés que nous avions rencontrées, ce fut un plaisir de travailler avec mes collègues sur le projet et d'apprendre en même temps de nouveaux concepts et notions.

- Jean DE BONFILS LAVERNELLE :

J'ai énormément apprécié l'étude du système de fichiers QNX6. Même s'il a été très compliqué de comprendre son fonctionnement, cela s'est avéré très intéressant. De plus, cela a été pour nous, un exercice très formateur. Le travail de compréhension du système de fichiers (comprendre le fonctionnement des inodes, Super Blocks ...) a été pour moi la partie la plus intéressante du projet. En effet, certains concepts étaient, pour nous, complètement étrangers et cela nous a permis de découvrir un domaine de l'informatique que je trouve extrêmement intéressant. J'ai aussi beaucoup apprécié l'analyse du code hexadécimal du système de fichiers pour comprendre son fonctionnement et notamment pour faire le lien entre les inodes et comprendre les mécanismes de suppression. Je me suis personnellement senti un peu seul au niveau de l'analyse du système de fichiers QNX6 et de la programmation de l'ingest module et regrette que l'on n'ait pas été plus sur ces tâches. D'un autre côté, je suis très content de comprendre le fonctionnement du système de fichiers. Même si le projet n'était pas simple et malgré les conditions du projet, je pense sincèrement que cela nous a apporté des connaissances et des méthodes que nous n'obtiendrons pas à travers des cours magistraux. Enfin, la rédaction du rapport s'est avérée être une tâche plus compliquée que prévue. En effet, expliquer certains concepts dont on ne connaît pas les termes exacts est très compliqué. Je pense que la meilleure façon pour expliquer le fonctionnement du système de fichiers QNX6 est d'utiliser des schémas. Je suis fier que l'on ait finalement réussi à compléter le projet dans les temps malgré tous les imprévus et la difficulté du projet. Je suis également fier de l'ingest module développé qui est même capable de récupérer certains fichiers supprimés et qui interagit avec le logiciel Autopsy. Pour conclure, ce projet a été vraiment intéressant, il nous a permis de comprendre des concepts étrangers pour nous. Je pense que le meilleur bénéfice de ce projet a été de nous faire gagner en autonomie. En effet, nous avons dû formuler des hypothèses sur le fonctionnement du système de fichiers QNX6 et par la suite les vérifier par nous-mêmes à travers différentes méthodes. Commencer un projet de ce type sans aucune connaissance et réussir à comprendre le fonctionnement du système de fichiers QNX6, par soi-même, à travers différentes méthodes a vraiment été très gratifiant.

- Lise MATET :

- Mathis GREAU :

12 Annexe

Code source de l'ingest module :

<https://github.com/jdbonfils/QNX6-Files-System-Reader-Ingest-Module>

Schémas sur le système de fichiers

https://mega.nz/file/k3oWSCAD#Qzmo3K3L6IewXJJH_zAeh_v81EwCpmkv1eWiNisQ1EQ

Fichiers servant de block device formattés en système de fichiers QNX6 et contenant des fichiers et des dossiers pouvant être récupérés à l'aide de l'ingest module développé :

<https://mega.nz/folder/MuAjSQLQ#ZpNtrxlGqVM3TwnHmRygbw>

Vidéo de Présentation de l'ingest module :

https://www.youtube.com/watch?v=H9FppPDLrpY&ab_channel=JeandeBonfils

<https://mega.nz/file/Unhy1SxK#JhoCfVPv3xk8a2MOS4RN099jdHRAPMct1gEYegqtID8>

Taches et réunions réalisées au cours du semestre 2 :

https://drive.google.com/file/d/1psq7VDIebmtF_xiJkcxKgztbh47F_GX3/view?usp=sharing

13 Bibliographie

13.1 Code utile au développement de l'ingest module

Certaines fonctions développées sont inspirées des projets suivants :

<https://nop.ninja/> - **Mathew Evans**

<https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/linux-kaunetem/-/tree/505a666ee3fc611518e85df203eb8c707995ceaa/fs/qnx6>

<https://github.com/ReFirmLabs/binwalk/issues/365>

13.2 Présentation de QNX

<https://fr.wikipedia.org/wiki/Boutisme>

<https://www.kernel.org/doc/html/latest/filesystems/qnx6.html#specification>

<http://www-igm.univ-mlv.fr>

<https://fr.wikipedia.org/wiki/POSIX>

https://fr.wikipedia.org/wiki/Threads_POSIX

http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_sys_arch%2Fproc.html

https://man7.org/linux/man-pages/man3/posix_spawn.3.html

<https://man7.org/linux/man-pages/man2/fork.2.html>

<https://man7.org/linux/man-pages/man3/exec.3.html>

13.3 Reverse engineering

https://fr.wikipedia.org/wiki/R%C3%A9tro-ing%C3%A9nierie#Br%C3%A8ve_histoire_de_la_r%C3%A9tro-ing%C3%A9nierie

<https://connect.ed-diamond.com/MISC/MISCHS-007/Introduction-au-reverse-engi>

https://fr.wikipedia.org/wiki/R%C3%A9tro-ing%C3%A9nierie_en_informatique#/media/Fichier:Retroingenierie_-_Byrne.jpeg <https://pediaa.com/what-is-the-difference-between-QNX-and-Linux.html>
http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fproc_MMUs.html

13.4 Architecture du système de fichiers de QNX6

<https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/linux-kaunetem/-/tree/505a666ee3fc611518e85df203eb8c707995ceaa/Documentation/filesystems>
<https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/linux-kaunetem/-/blob/505a666ee3fc611518e85df203eb8c707995ceaa/Documentation/filesystems/qnx6.txt> <https://www.kernel.org/doc/html/latest/filesystems/qnx6.html>
https://en.wikipedia.org/wiki/Unix_file_types#Regular_file https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxb200/dasspe.htm <http://ibgwww.colorado.edu/~lessem/psyc5112/usail/concepts/filesystems/everything-is-a-file.html> <https://www.programmersought.com/article/2919214259/> <https://www.companeo.com/sauvegarde-informatique/guide/support-de-stockage> [https://fr.wikipedia.org/wiki/Partition_\(informatique\)](https://fr.wikipedia.org/wiki/Partition_(informatique)) <https://www.malekal.com/la-partition-systeme-windows-structure>
<https://lecrabeinfo.net/differences-mbr-gpt-tables-de-partitionnement.html>
<https://www.tech2tech.fr/quelle-est-la-difference-entre-le-format-gpt-et-mbr.html>
https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_91 <https://unix.stackexchange.com/questions/259193/what-is-a-block-device>
https://en.wikipedia.org/wiki/Device_file <https://docs.freebsd.org/en/books/arch-handbook/driverbasics-block.html> <http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch13lev1sec1.html> https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxb200/dasspe.htm <https://www.jamescoyle.net/how-to/2096-use-a-file-as-a-link-on-qnx>
http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Ffsys_Partitions.html http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_user_guide%2Fstarting.html http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_utilities%2Fmkqnx6fs.html http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_user_guide%2Ffilesystems.html [http://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.hypervisor.user/topic/virt/qvm.html](http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.cookbook/topic/s2_adios_Driver_design.html) http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_utilities%2Fmount.html http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_utilities%2Ff%2Ffs-qnx6.so.html <https://www.linux.org/threads/qnx-file-systems.9025/> <https://reverseengineering.stackexchange.com/questions/8372/unpack-qnx-img-files>

13.5 Autopsy

<https://www.autopsy.com/> <https://www.sleuthkit.org/> http://www.sleuthkit.org/sleuthkit/docs/jni-docs/4.3/classorg_1_lsleuthkit_1_1datamodel_1_1_abstract_file.html http://www.sleuthkit.org/sleuthkit/docs/jni-docs/4.3/classorg_1_lsleuthkit_1_1datamodel_1_1_file_system.html http://www.sleuthkit.org/sleuthkit/docs/jni-docs/4.3/classorg_1_lsleuthkit_1_1datamodel_1_1_sleuthkit_case.html <https://github.com/tomvandermussele/autopsy-plugins> <https://github.com/markmckinnon/Autopsy-Plugins> http://sleuthkit.org/sleuthkit/docs/jni-docs/4.10.1/interfaceorg_1_lsleuthkit_1_1datamodel_1_1_content.html <https://github.com/sleuthkit/autopsy/tree/develop/pythonExamples> http://sleuthkit.org/autopsy/docs/api-docs/4.14.0/mod_python_ds_ingest_tutorial_page.html [#~:text=Overview, from%20a%20live%20Windows%20computer.&text=The%20logical%20imager%20produces%20one, Autopsy%20or%20mounted%20by%20Windows](http://sleuthkit.org/autopsy/docs/user-docs/4.12.0/logical_imager_page.html) <https://www.autopsy.com/python-autopsy-module-tutorial-2-the-data-source-ingest-module/> <https://www.autopsy.com/python-autopsy-module-tutorial-1-the-file-ingest-mo>

13.6 Reverse Engineering

<https://ghidra-sre.org/> <https://github.com/NationalSecurityAgency/ghidra> <https://www.youtube.com/channel/UC1cE-kVhqyiHCCjYwcpfj9w/search?query=ghidra> https://www.youtube.com/watch?v=fTGTnrgjuGA&ab_channel=stacksmashing https://www.youtube.com/watch?v=P8U12K7pEfU&ab_channel=0x6d696368 https://www.youtube.com/watch?v=ObLA0Za2PhY&ab_channel=JamesTate <https://www.csoonline.com/article/3393246/how-to-get-started-using-ida-pro/>

13.7 Autre

https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.sys_arch/topic/dll.html <https://askubuntu.com/questions/165219/what-exactly-happens-when-i-run-a-dll> https://www.youtube.com/watch?v=F61qtWetoew&ab_channel=media.ccc.de http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_utilities%2Ft%2Ftracerlogger.html <https://www.yumpu.com/en/document/read/25613206/system-analysis-toolkit-users-guide-qnx-softw> <http://www.mnis.fr/opensource/ocera/rtos/c2462.html> <https://www.jamescoyle.net/how-to/2096-use-a-file-as-a-linux-block-device> <https://www.cjmweb.net/vbindiff/> <https://stackoverflow.com/questions/19052977/how-to-use-dd-to-fill-a-disk-with-a-specific-char> https://fr.wikipedia.org/wiki/Server_Message_Block [https://fr.wikipedia.org/wiki/Samba_\(informatique\)](https://fr.wikipedia.org/wiki/Samba_(informatique)) https://fr.wikipedia.org/wiki/Licence_de_logiciel#Contrat_de_licence_utilisateur_final

<https://resources.infosecinstitute.com/topic/step-by-step-tutorial-on-reversing-a-binary-executable/>
<https://www.sleuthkit.org/autopsy/features.php>