Unix : construire un environnement de développement de A à Z

Warren Toomey

Unix : construire un environnement de développement de A à Z

Warren Toomey
The Unix Heritage Society (TUHS)
Traduction de l'anglais par Loïc Petitgirard[1].

Résumé

En avril 1969, alors qu'AT&T se retire du projet Multics, les chercheurs impliqués dans ce projet se voient privés de leur environnement de développement si « convivial ». Privés de leur « jouet », ces chercheurs ont commencé à prospecter pour un remplaçant. N'ayant rien trouvé d'approprié, Ken Thompson décide d'en écrire un, en partant de rien. Dès le milieu de l'année 1969 il crée un système d'exploitation autonome (self-hosting) sur un miniordinateur PDP-7 inutilisé. Il s'agit d'Unix, un système d'exploitation dont les systèmes actuels ont en partie hérité. Cet article s'intéresse à la création d'Unix, après qu'AT&T ait quitté le projet Multics, aux fonctionnalités et innovations de la version Unix créée sur le PDP-7, et au travail réalisé en 2016 pour remettre sur pied une version opérationnelle d'Unix PDP-7 sur la base du code source disponible.

Qu'est-ce qui a motivé le développement d'Unix ?

1969 n'a pas été une bonne année pour les chercheurs des Bell Labs de AT&T qui étaient impliqués dans le projet Multics. Multics était une tentative pour améliorer et développer un grand nombre de concepts du moment sur les systèmes d'exploitation (par exemple, la mémoire virtuelle, le multitâche) et de construire un système d'exploitation assurant un service continu comme pouvait l'être celui d'une entreprise électrique ou d'une compagnie téléphonique.

Mais Multics a souffert de l'effet de « second système » décrit par Brooks (1975) : il était trop technique et trop ambitieux pour atteindre tous ses objectifs sur la plateforme matérielle pour laquelle il avait été conçu, l'ordinateur General Electric GE-645. AT&T avait rejoint le projet Multics en 1964, avec General Electric et le MIT. En avril 1969, le projet

ayant dépassé les délais, AT&T décide de se retirer de Multics.

Cette décision laisse désœuvrés les chercheurs impliqués dans Multics (Ken Thompson, Dennis Ritchie, Doug McIlroy entre autres). Sandy Fraser décrit la situation aux Bell Labs dans un entretien avec Mike Mahoney[1] :

> Je rentrais dans le bâtiment et y trouvais une sorte de désarroi, un moral à coup sûr très bas. Il se trouve que j'étais le dernier arrivé. Il était clair que nous étions dans un moment de transition assez traumatisant pour bon nombre de gens… Le jouet [le GE-645] n'était plus là. La salle de l'ordinateur était vide. Les gens étaient tout simplement déprimés. Certains partaient. Il manquait clairement de l'entrain.

Pour s'être brûlé une première fois les ailes avec un projet de système d'exploitation, le management d'AT&T n'était pas prêt à soutenir de nouveaux travaux de recherches sur les systèmes d'exploitation. Mais les chercheurs avaient trouvé que le système Multics était un environnement de développement formidable et ils étaient motivés par l'idée de travailler sur une solution de remplacement adaptée à leurs besoins. Ritchie précise que[2] :

> Même si Multics ne pouvait pas accueillir beaucoup d'utilisateurs, il avait les qualités techniques pour notre projet, malgré des coûts exorbitants. Nous ne voulions pas perdre le petit créneau confortable que nous occupions, parce qu'il n'en existait pas d'autre équivalent : même les services de temps-partagé que les systèmes d'exploitation de GE allaient offrir plus tard n'existaient pas encore. Ce que nous voulions préserver, ce n'était pas seulement un bon environnement pour réaliser des programmes, mais un système autour duquel une communauté pouvait se former.

Les chercheurs du projet Multics aux Bell Labs ont tenté à de nombreuses reprises de persuader le management d'AT&T d'acheter une plateforme informatique plus modeste sur laquelle ils auraient écrit un système d'exploitation ; mais rétrospectivement, ces propositions exigeaient qu'AT&T dépense « *beaucoup trop d'argent pour trop peu de personnel avec un plan trop vague* » (Ritchie, 1980). En fin de compte, tous ces plans ont été rejetés.

Durant cette période sombre, Thompson était engagé dans deux activités qui allaient conduire au développement du système d'exploitation Unix. La première était le développement de Space Travel, une simulation informatique précise de la dynamique du Soleil, des planètes et des satellites du système solaire. Un joueur pouvait y naviguer dans un vaisseau spatial à travers le système solaire, voir le paysage et essayer d'atterrir sur les planètes et les satellites.

---

1 Entretien de M. Mahoney avec Sandy Fraser, 1989 [URL : https://www.princeton.edu/~hos/mike/transcripts/fraser.htm].

2 Entretien de M. Mahoney avec Dennis Ritchie, 1989 [URL : https://www.princeton.edu/~hos/mike/transcripts/ritchie.htm].

Space Travel a été initialement développé sur la plateforme GE-645, mais son fonctionnement était onéreux (75 $ de temps processeur pour un tour de jeu (Ritchie, 1980)) et son interface en ligne de commande laissait beaucoup à désirer. Après le retrait du projet Multics, Thompson a réécrit Space Travel pour le PDP-7, un ordinateur du laboratoire plus petit, qui avait été utilisé pour des travaux graphiques mais qui était devenu obsolète et excédait les besoins.

Ce travail a permis à Thompson de faire connaissance avec la plateforme PDP-7, qui était sévèrement contrainte à la fois en termes de mémoire (avec seulement 8,192 mots de 18-bits)[3] et d'espace disque (avec une capacité de stockage de seulement 1,024,000 mots), et une architecture du jeu d'instruction complexe. Ces restrictions ont en fin de compte influencé la conception du système Unix.

Le système de fichiers d'Unix

La seconde activité de Thompson après la fin du projet Multics était la recherche en conception et structuration des systèmes de fichiers. Thompson avait entamé cette recherche avant qu'AT&T ne se retire formellement de Multics.

> J'avais construit une simulation de haut niveau d'un système de fichiers complet. Je n'en étais pas au point où on donne des adresses, où la taille des fichiers peut être augmentée ou d'autres choses similaires. J'étais resté à un niveau supérieur. Je pense que nous n'en étions qu'à une ou deux séances de travail, Dennis [Ritchie], [Rudd] Canaday et moi. Nous ne discutions que d'idées générales sur la façon d'éviter que les fichiers ne s'emmêlent et les soucis liés à leur taille grandissante[4].

À partir de cette simulation de haut niveau d'un système de fichier, Thompson a pris le PDP-7 en main et a implémenté une structure du système de fichier. Au départ il s'agissait d'une simulation des fichiers et des actions sur ces fichiers, et au fur et à mesure le système simulé s'est étoffé pour devenir un système d'exploitation opérationnel.

> Pour faire fonctionner le système de fichiers, vous deviez créer, effacer et concaténer des fichiers, pour voir jusqu'à quel point il fonctionnait. Pour cela il fallait un scénario des commandes que vous deviez soumettre au système de fichiers, et le seul que nous avions était […] un ruban perforé qui disait : lit un fichier, écrit un fichier, ce genre de chose. Vous lanciez le scénario du ruban perforé et ça secouait un peu le disque. On ne savait pas exactement ce qui se produisait. On ne pouvait tout simplement pas le regarder, on ne pouvait pas le voir, on ne pouvait rien faire. Ensuite nous avons conçu quelques outils sur le système de fichiers. Nous avons

3 Un mot correspond à la taille de l'instruction standard utilisée par la machine.

4 Entretien de M. Mahoney avec Ken Thompson, 1989 [URL : https://www.princeton.edu/~hos/mike/transcripts/thompson.htm].

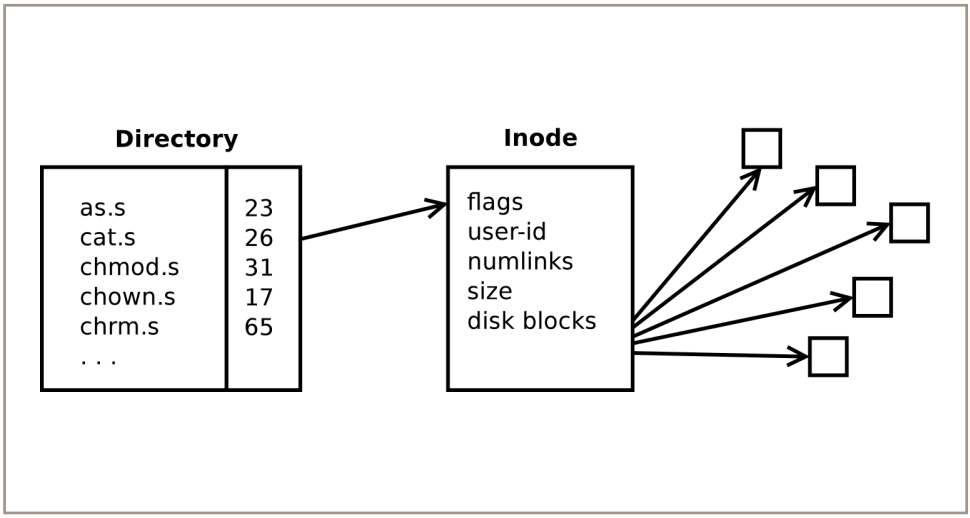

Figure 1 – Structure Inode sur Unix PDP-7

> utilisé ce ruban perforé pour charger le système de fichier et ses outils, et alors nous avons pu […] taper des commandes dans un outil qui s'appelait un interpréteur (*shell*), pour tordre le système de fichiers dans tous les sens, afin de pouvoir mesurer son fonctionnement et ses réactions. Le système de fichier primitif a duré peut-être un jour ou deux avant que nous ayons développé les outils dont nous avions besoin pour le charger[5].

Au XXIe siècle, nous voyons, à juste titre, les systèmes d'exploitation comme des systèmes extrêmement complexes, demandant des centaines de développeurs et des milliers d'heures pour être conçus. Au XXe siècle, Thompson ne le savait pas. Au lieu de cela, il raconte son été 1969 :

> Ma femme est partie en vacances en Californie pour voir mes parents. Elle est partie un mois en Californie et je me suis alloué une semaine pour chaque partie du système, le noyau, l'interpréteur, l'éditeur et l'assembleur, pour que le système puisse se reproduire lui-même. Pendant le mois où elle était absente, tout a été réécrit sous une forme qui ressemblait à un système d'exploitation, avec des outils de types connus : un assembleur, un éditeur et un interpréteur. Ça ne tenait pas encore la route, mais on y était presque[6].

En un mois de temps, Ken Thompson a pris un simulateur de système de fichiers et l'a réécrit pour en faire un système d'exploitation autonome (*self-hosting*) que nous connaissons aujourd'hui sous le nom de Unix PDP-7.

5 *ibid.*

6 *ibid.*

La structure du système de fichiers

Dès le départ, la structure du système de fichiers de l'Unix PDP-7 avait les marques distinctives qu'elle allait conserver le reste de sa vie (comme illustré sur la figure 1) :

- des *inode* (index) qui contiennent les métadonnées de chaque fichier ;
- des répertoires séparés de noms de fichiers, chacun pointant vers un *inode* ;
- un ensemble de blocs disque qui stockent les contenus de chaque fichier.

L'*inode* pouvait stocker jusqu'à sept numéros de blocs disque, pour des fichiers jusqu'à 7x64 = 448 mots de long. Pour des fichiers plus gros, le fichier était repéré dans le champ des « indicateurs » (*flags*) comme un grand fichier. Dans ce cas, les numéros de bloc pointaient vers des blocs disques qui chacun contenait 64 numéros de blocs disque, autorisant des fichiers jusqu'à 7x64x64 = 28 672 mots de longueur.

Le champ des indicateurs (*flags*) de l'*inode* identifiait aussi le type du fichier (fichier standard, répertoire ou fichier spécial), et les permissions pour les données (lecture et écriture pour le propriétaire du fichier, lecture et écriture pour tous les autres utilisateurs). Le propriétaire d'un fichier était identifié par le champ « *user-id* » ; le concept de « groupe d'utilisateurs » n'arrivera pas avant quelques années.

Le concept de lien physique a été une innovation du système de fichier de l'Unix PDP-7. Comme le nom d'un fichier était séparé des métadonnées de l'*inode*, cela permettait à deux ou plusieurs noms de fichiers de pointer vers la même métadonnée, comme l'illustre la figure 2.

Tous les noms de fichier reliés à l'*inode* étaient équivalents : aucun ne valait plus qu'un autre. Le champ « *numlink* » de l'*inode* enregistrait le nombre de noms de fichiers liés au fichier ; le fichier n'était supprimé du système de fichier que dans le cas où tous les noms de fichiers étaient déliés et que le compteur de lien tombait à zéro.

Les répertoires

Si l'organisation des contenus des fichiers et de leurs métadonnées était plutôt bien définie sur l'Unix PDP-7, on ne peut pas en dire autant de l'organisation des répertoires et de leurs relations entre eux.

> La première structure de système de fichiers ne dessinait pas une hiérarchie de répertoires mais un graphe orienté, et nous ne le limitions pas à un arbre. Nous expérimentions différentes topologies[7].

7 *ibid.*

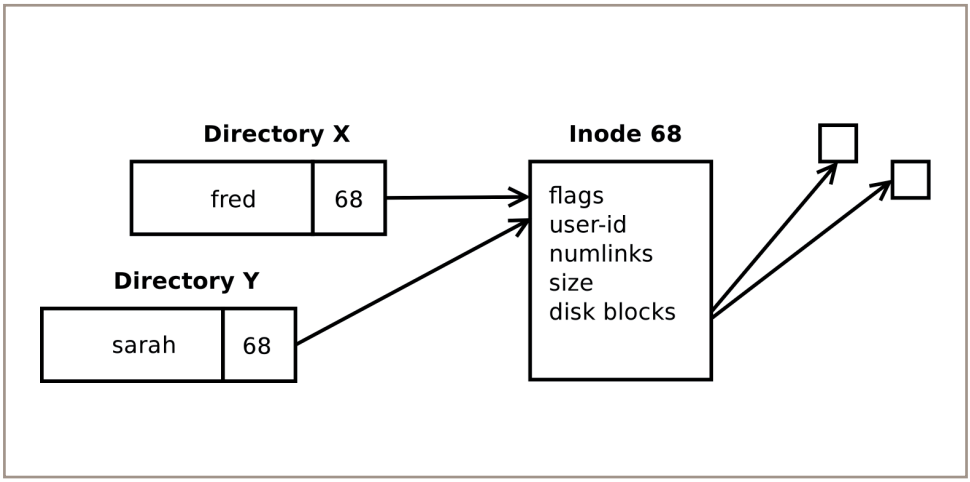

Figure 2 – Les liens physiques dans Unix PDP-7

Le noyau de l'Unix PDP-7 comprenait et gérait le concept de répertoire (contenant des noms de fichiers correspondants) et le fait qu'un répertoire puisse contenir des liens vers d'autres répertoires. Cependant, l'organisation des répertoires lui était indifférente.

Le premier système de fichiers Unix, et celui qui subsiste dans le système Unix PDP-7 restauré, contenait deux répertoires de haut niveau, « *dd* » et « *system* », comme l'illustre la figure 3. Le répertoire « *dd* » contenait les entrées du répertoire « *system* » et tous les répertoires « *home* » de l'utilisateur. Le répertoire « *system* » contenait tous les principaux fichiers exécutables du système et les fichiers spéciaux des périphériques. Aujourd'hui, « *dd* » correspondrait au répertoire « *root* », et « *system* » à une combinaison des répertoires « */bin* » et « */dev* ».

Pour naviguer dans le système de fichiers, chaque répertoire avait besoin d'un lien vers le répertoire « dd », pour y revenir. Et comme l'interpréteur était programmé pour chercher les codes binaires exécutables dans le répertoire « *system* », chaque répertoire devait contenir une entrée pointant dans le répertoire « *system* ».

Dans l'Unix PDP-7, le concept de chemin absolu comme */usr/local/bin/less* n'existait pas, ni celui de nom de chemin relatif (par exemple : *../../file ou sub-dir-1/subdir2/file*). Une fois que l'utilisateur était entré dans un répertoire, il ne pouvait faire référence qu'à des fichiers ou des répertoires visibles depuis ce répertoire. Ritchie souligne que :

> La commande « link » prenait la forme suivante :
>
> ln dir file newname

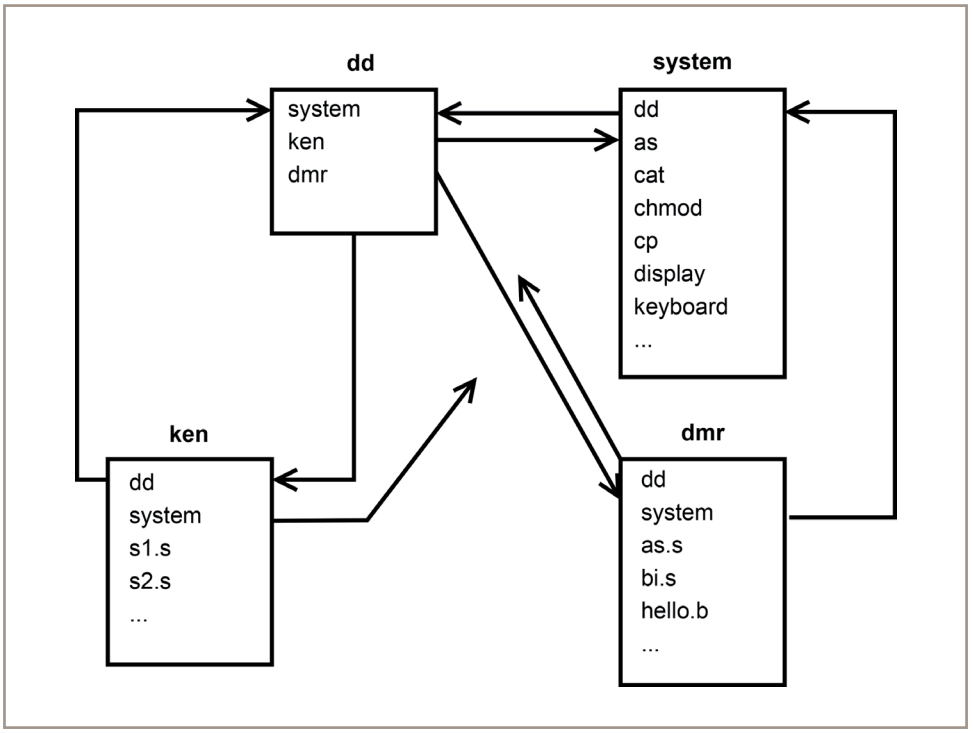

Figure 3 – Structure des répertoires sur Unix PDP-7

où « *dir* » était un fichier répertoire dans le répertoire en cours, « *file* » une entrée existant dans ce répertoire, et « *newname* » le nom du lien qui était ajouté au répertoire en cours. Parce que « *dir* » devait être dans le répertoire actuel, il est évident que les interdictions actuelles de créer des liens à des répertoires n'étaient pas imposées : le système de fichiers de l'Unix PDP-7 avait la forme d'un graphe général orienté.

Pour que chaque utilisateur n'ait pas à maintenir un lien vers tous les répertoires qui l'intéressaient, il existait un répertoire appelé « *dd* » qui contenait les entrées pour le répertoire de chaque utilisateur. Si j'étais dans mon répertoire « *home* » (« *dmr* »)[8], faire un lien vers un fichier x du répertoire « *ken* », je pouvais faire :

ln dd ken ken

ln ken x x

rm ken

Cela rendait les sous-répertoires suffisamment difficiles à utiliser pour qu'ils ne soient jamais utilisés en pratique. La convention « dd » rendait la commande « chdir » relativement commode[9]. Elle

8 NB : « *dmr* » correspond à Dennis MacAlistair Ritchie, et « *ken* » à Kenneth Thompson.

9 « chdir » pour « change directory ».

prenait plusieurs arguments et faisait passer du répertoire en cours au répertoire identifié. Ainsi :

chdir dd ken

faisait passer [du répertoire « *dmr* » au répertoire « *ken* » via le répertoire « *dd* »] (Ritchie, 1980).

Dans l'Unix PDP-11 (écrit en 1971), l'entrée « *dd* » dans chaque répertoire a évolué vers l'entrée « .. » (deux points) qui pointe vers le répertoire immédiatement supérieur. La mémoire supplémentaire disponible sur le PDP-11 permettait au noyau de gérer des chemins absolus et relatifs ; une fois que l'interpréteur pouvait trouver les exécutables dans le répertoire « */bin* », l'entrée « *system* » dans chaque répertoire n'était plus nécessaire.

Les opérations sur les fichiers et les fichiers spéciaux

L'Unix PDP-7 mettait en œuvre un ensemble d'opérations sur les fichiers immédiatement reconnaissables pour un programmeur Unix ou Linux actuel :

- Descripteur de fichier (*file descriptor*) = *open* (nom de fichier, mode)
- *close* (*file descriptor*)
- *read* (*file descriptor, buffer, amount*)
- *write* (*file descriptor, buffer, amount*)
- *seek* (*file descriptor, amount, whence*)

De cette manière, le noyau Unix s'abstrayait des détails du stockage d'un fichier et le remplaçait par un tableau linéaire de mots du PDP-7 (des octets dans les versions ultérieures d'Unix). L'Unix PDP-7 a étendu cette abstraction en introduisant le concept de « fichiers spéciaux ». Chaque fichier spécial représentait le contenu d'un périphérique auquel l'accès était possible en utilisant les opérations de fichiers ci-dessus, quelles que soient les caractéristiques du périphérique.

Le noyau d'Unix PDP-7 prenait en charge ces fichiers spéciaux, conservés dans le répertoire *system* grâce aux commandes :

- *ttyin* et *ttyout*, pour la console du PDP-7
- *keyboard* et *display*, pour l'écran Graphics-2 qui était utilisé comme un second terminal
- *pptin* et *pptout*, le dispositif de ruban perforé

Ainsi, sur un système de seulement 8 192 mots de mémoire, l'Unix PDP-7 était capable de fournir un environnement de développement multitâche pour deux utilisateurs.

Processus et contrôle de processus

L'Unix PDP-7 fournissait un environnement multitâche en divisant les 8K mots de mémoire en deux moitiés. La moitié basse de la mémoire était réservée au noyau. La moitié haute était mise de côté pour les processus en cours d'exécution. L'Unix PDP-7 basculait les pro-

cessus de la mémoire au disque durant le changement de contexte[10] entre deux processus en cours. Bien que cela permettait d'isoler les processus, le *hardware* du PDP-7 n'empêchait pas les processus d'accéder aux 4K de mots de la mémoire basse du noyau.

Ce que nous reconnaissons aujourd'hui comme l'ensemble canonique des mécanismes de contrôle de processus Unix (*fork()*, *exec()*, *exit()* et *wait()*) a évolué par étapes lorsque la version pour le PDP-7 a été développée puis réécrite pour la plateforme PDP-11.

Les commandes « *fork* » et « *exec* »

Thompson a emprunté et modifié le concept de *fork()* qu'il avait vu sur le système Project Genie sur l'ordinateur SDS930 lorsqu'il était étudiant en licence à l'Université de Berkeley. *Fork()* a été développé par Melvin Conway en 1962 comme une paire de primitives, *fork* et *join*, pour permettre la planification des processus d'un système multiprocesseur (Nyman & Laakso, 2016). Dans Unix PDP-7, *fork()* était utilisé par un processus pour créer une copie de lui-même qui pouvait ensuite être ordonnancée indépendamment.

Si *fork()* permet à un système de démarrer de nouveaux processus, ils sont cependant tous identiques. Une autre primitive était nécessaire pour permettre au système d'exécuter des programmes différents. Dans Unix aujourd'hui, c'est le mécanisme *exec()*, implémenté dans le noyau Unix, qui le réalise. Dans Unix PDP-7 ce mécanisme était implémenté dans l'interpréteur, comme illustré dans la figure 4.

Lorsque l'interpréteur du PDP-7 lisait une commande d'exécution d'un nouveau programme, il faisait d'abord une copie de lui-même avec un *fork()*. L'interpréteur originel attendait que le nouvel interpréteur s'exécute et se termine. Le nouvel interpréteur devait alors se remplacer par le nouveau programme demandé par l'utilisateur.

La fonction exécutée dans l'interpréteur se relogeait d'abord dans la partie supérieure de la mémoire, juste au-dessous des arguments du programme entrés par l'utilisateur dans la ligne de commande. La fonction relogée faisait alors un *open()* du fichier exécutable et un *read()* de son contenu, qu'elle transférait dans la mémoire utilisateur à partir de l'adresse 4096. Une fois l'exécutable chargé en mémoire, la fonction relogée faisait un *close()* du fichier et allait à l'adresse 4096 pour démarrer l'exécution du nouveau programme.

Ritchie fait remarquer que même ce simple mécanisme de contrôle de processus a évolué à partir d'une construction primitive :

10 Le « contexte » est l'ensemble des états des registres la machine, à un instant donné, qui permet l'exécution des processus.

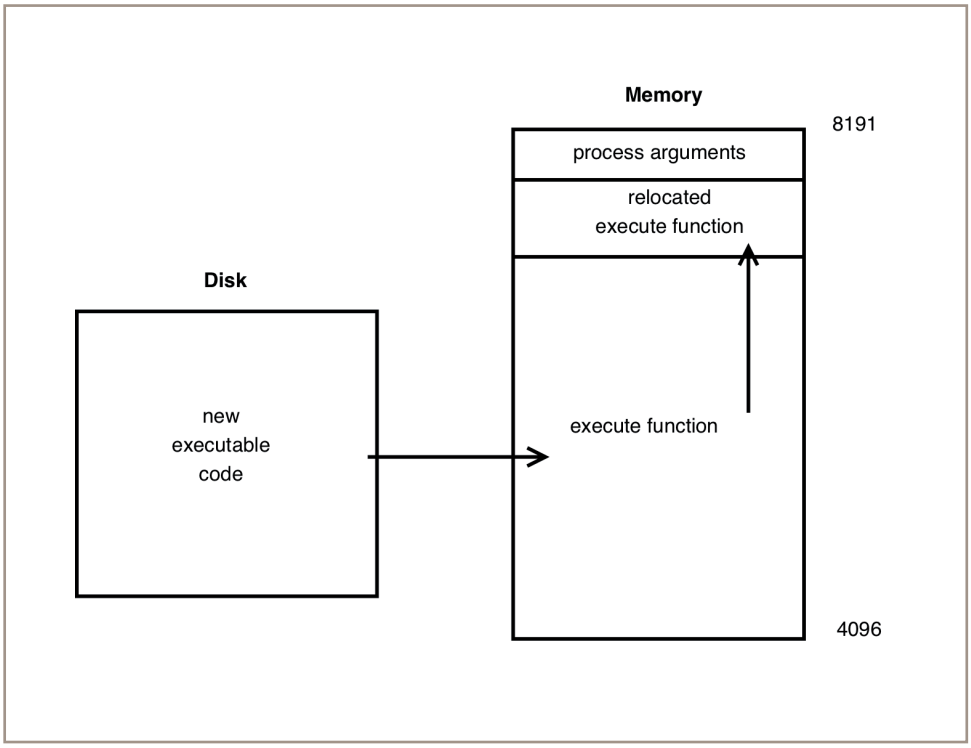

Figure 4 – Le mécanisme exec de l'interpréteur (*shell*)

Le contrôle de processus dans sa forme moderne a été conçu et implémenté en deux jours. C'est étonnant à quel point il s'adaptait au système existant ; en même temps, il est facile de voir comment les caractéristiques légèrement inhabituelles de la conception sont présentes précisément parce qu'elles représentaient des petits changements faciles à coder dans l'existant. La séparation entre les fonctions *fork* et *exec* en est un bon exemple. Le modèle le plus commun pour créer de nouveaux processus implique de spécifier un programme pour le processus à exécuter ; dans Unix, le processus créé par la commande *fork* continue de faire tourner le même programme que son parent jusqu'à ce qu'il appelle explicitement un *exec*.

La séparation des fonctions n'est certainement pas limitée à Unix, et était en fait présente dans le système de temps-partagé de Berkeley (Project Genie) qui était bien connu de Thompson. Néanmoins, il paraît raisonnable de penser qu'elle existe dans Unix principalement parce qu'il était facile d'implémenter *fork* sans changer grand-chose d'autre. Le système gérait à l'origine de multiples (c'est-à-dire deux) processus ; il y avait une table des processus, et les processus étaient échangés entre la mémoire principale et le disque. L'implémentation initiale de *swap* ne nécessitait que :

1. l'extension de la table des processus,

2. l'ajout d'un appel fork qui copiait le

processus en cours dans la partie de la mémoire dédiée à l'échange d'entrée/sortie déjà existantes, et faisait des ajustements dans la table des processus.

En fait, l'appel *fork* du PDP-7 nécessitait précisément 27 lignes de code assembleur. Bien sûr, d'autres changements étaient nécessaires dans le système d'exploitation et les programmes de l'utilisateur, et certains d'entre eux étaient plutôt intéressants et inattendus. Mais un mécanisme combiné de *fork/exec* aurait été considérablement plus compliqué, ne serait-ce parce que l'*exec* en tant que tel n'existait pas ; sa fonction était déjà réalisée par l'interpréteur, en utilisant des entrées/sorties explicites (*explicit I/O*). (Ritchie, 1980)

Les commandes *exit* et *wait*

Dans les systèmes Unix d'aujour d'hui, un processus qui a « fork()é » un nouveau processus peut faire un *wait()* pour attendre que le nouveau processus se termine. Ce dernier peut réaliser un *exit()* pour clore son exécution. Cette commande renvoie un statut « exit » pour signifier sa terminaison au processus original qui attend sur le *wait()*.

Comme pour *fork()* et *exec()*, les commandes *wait()* et *exit()* sont issues d'autres mécanismes. Mais dans ce cas, les mécanismes précédents n'étaient pas moins, mais plus sophistiqués.

Les primitives qui sont devenues *exit* et *wait* étaient considérablement plus générales que le mécanisme actuel. Un couple de primitives envoyait des messages d'un mot entre des processus identifiés :

Smes(pid, message)

(pid, message) = rmes ()

Le processus cible de *smes* n'avait pas besoin d'avoir une relation de filiation avec le receveur, même si le système ne donnait pas de mécanisme pour communiquer les identifiants des processus, à part le *fork()* qui renvoyait au père l'ID du fils et réciproquement. Les messages n'étaient pas mis en file d'attente ; le processus émetteur était retardé jusqu'à ce que le récepteur lise le message.

Le service de messages était utilisé ainsi : l'interpréteur *(shell)* parent, après avoir créé un processus pour exécuter une commande, envoyait un message au nouveau processus par *smes()* ; quand la commande se terminait (en supposant qu'elle ne tentait pas de lire un quelconque message) l'appel *smes()* bloqué de l'interpréteur renvoyait une erreur signalant que le processus cible n'existait pas. Ainsi le *smes()* de l'interpréteur devenait dans les faits un équivalent de la commande *wait()*.

Un protocole différent, qui exploitait la plupart des généralités offertes par les messages, était utilisé entre le programme d'initialisation et les interpréteurs pour chaque terminal. Le processus d'initialisation, dont il était convenu que l'ID était 1, créait un interpréteur pour chacun des terminaux, puis faisait un *rmes()* ; chaque interpréteur, après avoir lu la fin de son fichier d'entrée, utilisait la fonction *smes()* pour envoyer le traditionnel message « j'ai terminé » au processus d'initialisation, qui recréait un nouveau processus interpréteur pour ce terminal.

> Je ne me souviens pas d'autre utilisation des messages. Ceci explique pourquoi le service a été remplacé par les appels [*exit* et] *wait()* de l'actuel système Unix, qui est moins général, mais plus directement approprié au but recherché. Un bug évident dans le mécanisme explique aussi possiblement ce remplacement : si un processus de commande essayait d'utiliser des messages pour communiquer avec d'autres processus, cela pouvait perturber la synchronisation de l'interpréteur. L'interpréteur pouvait dépendre d'un message qui n'avait jamais été reçu ; si une commande exécutait *rmes(),* elle pouvait recevoir un faux message de l'interpréteur, et obliger l'interpréteur à lire une ligne d'entrée supplémentaire, comme si la commande était terminée. Si le besoin de messages généraux s'était manifesté, le bug aurait été réparé. (Ritchie, 1980)

Les outils de l'Unix PDP-7

Unix PDP-7 était non seulement un système d'exploitation polyvalent dont le noyau était compacté dans une mémoire de 4 000 mots, mais ses développeurs ont aussi créé un nombre important d'outils dont les descendants sont toujours utilisés aujourd'hui.

Dans son texte « Draft of the Unix Time-sharing System »[11] écrit en 1971 quand les deux systèmes Unix pour PDP-7 et PDP-11 existaient, Ritchie documente ces commandes et outils : *b*, *cat*, *chdir*, *chmod*, *cp*, *db*, *ed*, *ln*, *ls*, *mkdir*, *mv*, *nm*, *pr*, *rm*, *roff*, *sh*, *tm* et *un*. Seuls un petit nombre d'entre eux ne serait pas familier à l'utilisateur d'un Unix d'aujourd'hui :

- *b*, le compilateur B ;
- *db*, un débogueur pour des images mémoire (des images disque des processus qui ont planté) ;
- *roff*, un outil de traitement des documents, et précurseur de *nroff* ;
- *tm*, un outil pour imprimer des informations relatives au temps, en provenance du noyau (*kernel*), par exemple le temps dans le mode « utilisateur », le temps dans le mode noyau, le temps passé à traiter les interruptions ;
- *un*, qui liste les symboles non identifiés dans un exécutable.

Plusieurs outils d'Unix PDP-7 méritent d'être passés en revue de manière plus complète.

L'interpréteur (*shell*)

Unix PDP-7 a été le premier système d'exploitation à fournir un interpréteur de commande qui était modifiable et remplaçable par l'utilisateur. Des systèmes antérieurs comme CTSS et Multics avaient des interpréteurs en ligne de commande, mais ils étaient partie intégrante du système et ne pouvaient pas être changés ou modifiés.

Unix PDP-7 a été aussi le premier système à fournir à ses outils des abstrac-

11 Ritchie D.M., «Draft: The UNIX Time-sharing System», texte de 1971 publié sur le site The Unix Heritage Society [URL : http://www.tuhs.org/Archive/Distributions/Research/McIlroy_v0/UnixEditionZero-Threshold_OCR.pdf].

tions d'unité standard d'entrée (*standard input*) et d'unité standard de sortie (*standard output*). En substance, tout processus démarrait avec un fichier déjà ouvert pour lire des entrées, et un autre fichier ouvert pour écrire les sorties. Par défaut, ces fichiers étaient connectés au terminal de l'utilisateur.

L'interpréteur de l'Unix PDP-7 fournissait à l'utilisateur des mécanismes pour associer ces fichiers ouverts à des fichiers existants (ou nouveaux), et aussi aux fichiers spéciaux. Ritchie indique :

> La notation très pratique pour les redirections I/O, à l'aide des symboles « > » et « < », n'existait pas aux tout débuts du système Unix PDP-7, mais elle est apparue très tôt. Comme beaucoup d'autres choses dans Unix, cela a été inspiré d'une idée venant de Multics. Multics avait un mécanisme de redirection des I/O assez général (Project MAC, 1969) incarnant (représentant) des flux (*stream*) d'I/O identifiés qui peuvent être redirigés dynamiquement vers différents dispositifs, fichiers, et même à travers des modules spéciaux de traitement des flux. Dans la version de Multics qui nous était familière, il existait même une commande qui passait (orientait) la sortie suivante, normalement destinée au terminal, vers un fichier, et une autre commande pour rattacher la sortie au terminal. Si, sur Unix, on tape la commande :
>
> *ls > xx*
>
> pour avoir la liste des noms de fichiers dans [le fichier] xx, sur Multics la même opération était réalisée par :
>
> *iocall attach user output file xx list*
>
> *iocall attach user output syn user i/o*
>
> Même si cette séquence très maladroite était souvent utilisée à l'époque de Multics, et aurait été très simple à intégrer dans l'interpréteur de Multics, ça ne nous est pas venu à l'idée, ni à d'autres, à ce moment-là. Je suppose que l'ampleur du projet Multics en donne la raison : ceux qui ont implémenté le système I/O étaient aux Bell Labs à Murray Hill, alors que l'interpréteur a été réalisé au MIT. Nous n'envisagions pas de faire des changements dans l'interpréteur (c'était leur programme) ; réciproquement, les détenteurs de l'interpréteur n'avaient peut-être pas connaissance de l'utilité de *iocall,* sans parler de sa maladresse. (Le manuel Multics de 1969 (Project MAC, 1969) fait référence à *iocall* comme une commande *author-maintained,* c'est-à-dire non standard). Le système I/O d'Unix et son interpréteur étant sous contrôle exclusif de Thompson, cela a été l'affaire d'une heure ou deux pour l'implémenter, quand la bonne idée a fait surface. (Ritchie, 1980)

La notion de tube (*pipe*) est un autre concept important d'Unix qui n'arrivera pas dans le système avant la 3e édition de l'Unix PDP-11 en 1973.

Le traitement de texte avec *roff*

Mis à part la recherche sur les systèmes d'exploitation, le système Unix a été utilisé très tôt en tant que système de traitement de document. L'évolution de *roff*, l'outil de traitement de document,

est intéressante. Il a vu le jour sous le nom de *runoff*, l'outil de J. Saltzer, écrit en langage assembleur pour le système d'exploitation CTSS (Saltzer, 1965). Celui-ci a été réécrit par Doug McIlroy en BCPL pour le système d'exploitation Multics. Ensuite, il a été réécrit en assembleur PDP-7 pour donner *roff* (avec des fonctionnalités réduites) pour l'Unix PDP-7. Puis, pour justifier l'achat d'un mini-ordinateur PDP-11, les chercheurs sur Unix ont réécrit *roff* en langage assembleur du PDP-11 pour assurer les besoins en traitement de documents du département des brevets d'AT&T. Les utilisateurs courants de Linux considèrent peut-être que l'*open source* est un concept relativement nouveau, mais le partage et le développement de code source a été un mécanisme important depuis les débuts de l'informatique.

Le compilateur B

Comme nous l'avons vu plus haut, Thompson avait écrit un assembleur pour le mini-ordinateur PDP-7 pour faire du système Unix un système autonome. Le noyau et tous les outils originaux de l'Unix PDP-7 ont été écrits en assembleur PDP-7. Mais les développeurs Unix ont éprouvé le besoin d'utiliser des langages de haut niveau. Thompson indique que :

> Depuis le début [nous voulions écrire le système dans un langage de haut niveau]… C'était une influence de Multics. Vu la complexité qu'il y a à conserver l'existant, nous savions pertinemment qu'on ne pouvait pas conserver un élément [*a fortiori* en assembleur], l'écrire et le faire fonctionner, [alors] qu'il va forcément évoluer… PL/I [le langage de haut-niveau utilisé pour Multics] était de trop haut niveau pour nous, ainsi que la version plus simple de PL/I dans Multics (un truc appelé EPL). Après le montage [d'Unix PDP-7], ou en même temps que sa sortie, BCPL était en train d'émerger et c'était un ticket gagnant pour nous deux [*i.e.* Thompson et Ritchie]. Nous avons été tous les deux captivés par le langage et nous avons beaucoup travaillé avec[12].

BCPL était un langage pour systèmes « orientés-mot » développé par Martin Richards à l'Université de Cambridge (Richards, 1969), qui était conçu pour être assez portable sur d'autres machines et systèmes. Comme BCPL était cependant encore un trop « gros » langage pour le PDP-7, Thompson a repris les structures essentielles de BCPL et a écrit un interpréteur d'un langage intermédiaire implémentant ces structures pour le PDP-7. Il a ensuite écrit un compilateur à destination de ce langage intermédiaire, et a créé le langage B.

> C'était le même langage que BCPL, mais il avait l'air complètement différent. En termes de syntaxe c'était une réécriture, mais la sémantique était exactement la même que BCPL. Et en fait, la syntaxe était telle que, si vous

12 Entretien de M. Mahoney avec Ken Thompson, 1989 [URL : https://www.princeton.edu/~hos/mike/transcripts/thompson.htm].

> ne regardiez pas de trop près, vous auriez dit que c'était du C. Parce que c'était du C, mais sans les types[13].

Très peu de choses de cette phase d'Unix ont survécu, seulement quelques programmes PDP-7 simples, écrits en B. Le langage B, et son compilateur, allaient évoluer en de nombreuses étapes vers le langage C (Ritchie, 1993). Ritchie et Thompson réécriront le noyau d'Unix en langage C de haut niveau en 1974 (Ritchie & Thompson, 1974), atteignant leur objectif d'écrire le système en langage de haut niveau.

Restaurer le système Unix PDP-7

Pendant de nombreuses années, le système Unix PDP-7 est resté essentiellement une créature mythologique. Même si son existence était documentée[14] (Salus, 1994), aucun code source n'avait subsisté à part pour la commande *dsw*, posté par Dennis Ritchie sur le groupe de nouvelles net.unix-wizards de Usenet en 1984.

En tant que fondateur de l'Unix Heritage Society[15], j'ai passé beaucoup de temps à récupérer des vieux systèmes Unix et les remettre en fonctionnement. Les restaurations de la première version de l'Unix PDP-11 et du premier compilateur C (Toomey, 2009) ont été des succès notables. Mais le système Unix PDP-7 restait insaisissable.

En 2016, un membre de longue date de l'Unix Heritage Society a révélé qu'il était en possession d'un tirage papier du code source d'Unix PDP-7, qu'il avait copié dans les années 1980 lorsqu'il travaillait aux Bell Labs d'AT&T. Nous l'avons encouragé à le scanner et à le partager avec la Society.

Le code source de n'importe quel système informatique est inutile en soi, à moins qu'il n'y ait un environnement pour le convertir en exécutables pour la machine et un environnement qui puisse faire tourner ces exécutables. En 2016, le PDP-7 était un souvenir lointain et, même si une telle machine était disponible, il n'y avait pas d'assembleur pour convertir le code source en format exécutable.

Heureusement, Bob Supnik et une cohorte d'excellents développeurs avait conçu SimH (Supnik & Walden, 2015), un simulateur de nombreux ordinateurs anciens incluant le PDP-7 et ses périphériques. Lorsque le code source de l'Unix PDP-7 a été rendu disponible, une équipe de trois personnes (Phil Budne, moi-même et Robert Swierczek) a commencé à convertir le code source en un système fonctionnel.

La première tâche consistait à écrire un nouvel assembleur PDP-7, ce qui a été

---

13 *ibid.*

14 « Draft : The UNIX Time-Sharing System », D.M. Ritchie, 1971. *Cf.* note 11.

15 Site web de l'association [URL : http://www.tuhs.org].

initié par moi-même, et complété ensuite par Phil Budne. Nous avions le code source pour l'assembleur Unix PDP-7, mais nous étions dans une situation de type « l'œuf et la poule » : la source de l'assembleur ne pouvait s'assembler lui-même.

Avec notre assembleur, nous avons pu assembler le code source en code machine pour le PDP-7 mais nous ne pouvions pas encore exécuter ces instructions. Nous ne pouvions pas utiliser SimH, parce qu'il simulait un système entier ; pour exécuter un code Unix, nous avions besoin d'un système complet : machine, noyau, un système de fichier opérationnel et des outils. Un bug dans une seule de ces parties empêcherait le système de fonctionner correctement.

À la place, nous avons choisi d'implémenter un simulateur du mode *User* : un mode qui exécute la plupart des instructions machine du PDP-7, et convertit les appels système de l'Unix PDP-7 en appels système à l'OS hôte sous-jacent. Wine (Amstadt & Johnson, 1994) est un exemple de simulateur de mode *User* similaire. Utilisant ce simulateur, nous pouvions tester notre assembleur et le code source des outils originaux de l'Unix PDP-7 sans se soucier du noyau Unix ou du système de fichiers.

Avec un assembleur et des outils en lesquels nous pouvions avoir confiance, nous nous sommes intéressés au noyau de l'Unix PDP-7 et à la construction d'un système de fichiers. Phil Budne s'est chargé de faire fonctionner le noyau, et j'ai écrit un outil pour construire un système de fichiers approprié. Les deux missions étaient ardues. Les développeurs d'Unix avaient laissé très peu de commentaires dans leur code assembleur, et les instructions PDP-7 étaient étrangères à tous les membres de l'équipe. Nous avons passé beaucoup de temps à déduire l'objectif de sections de code assembleur, et à ajouter des commentaires et annotations à celui-ci. En ce qui me concerne, le manque total de documentation sur le système de fichiers en a compliqué le développement. En utilisant les informations que nous avions à portée de main (le code source du noyau, la documentation du système Unix PDP-11 et la possibilité de faire exécuter pas à pas les instructions sur SimH), j'ai pu construire au final un système de fichiers Unix PDP-7 opérationnel.

Phil Budne a eu des difficultés analogues avec le noyau de l'Unix PDP-7. Il n'y avait pas de documentation sur l'écran Graphics-2, qui avait été construit au sein d'AT&T[16]. Non seulement Phil devait déduire ses modes opératoires, mais il devait aussi ajouter du code à SimH pour simuler le terminal.

Ken Thompson avait pris un mois en 1969 pour écrire le noyau, l'assembleur, l'interpréteur et le débugger. Sur ces quatre outils essentiels, le code source de l'interpréteur avait disparu. Utilisant

16 *Cf.* la page dédiée sur le site des Bell Labs [URL : http://cm.bell-labs.co/who/dmr/spacetravel.html].

les informations fournies par Ritchie dans son article « The Evolution of the Unix Time-Sharing System » (Ritchie & Thompson, 1974) et les bribes du code source PDP-7 existant, Phil Budne a reconstruit héroïquement un interpréteur fonctionnel pour le système Unix PDP-7.

Lorsque toutes ces tâches réalisées ont été mis bout à bout, l'équipe a pu annoncer que le système temps-partagé Unix PDP-7 avait été complètement remis en fonctionnement ; le code source original et les outils utilisés pour la reconstruction sont disponibles en téléchargement sur Github[17]. Cela inclut plusieurs outils pour lesquels le code source avait disparu : *cp*, *ln*, *ls* et *mv*.

En arrière-plan, Robert Swierczek avait travaillé à la reconstruction du compilateur B. Son code source avait également été perdu, mais l'interpréteur pour le langage intermédiaire avait survécu. Robert a utilisé le code source du plus ancien compilateur C, retiré le code concernant les types, et réorienté pour produire le code de l'interpréteur. Chef-d'œuvre d'amorçage (*bootstrapping*) inversé, Robert a écrit le compilateur B pour qu'il soit valable pour du code B ou C, et pour qu'il puisse se recompiler lui-même. Pour construire le compilateur, il est d'abord compilé en utilisant un compilateur C moderne. Ce compilateur qui n'est pas encore un compilateur PDP-7, est ensuite utilisé pour compiler le code source du compilateur B (une nouvelle fois), générant la version pour le langage intermédiaire qui peut être exécutée en utilisant l'interpréteur B de l'Unix PDP-7.

Conclusion

Le développement d'Unix a été en quelque sorte une réaction à l'encontre de la mentalité « *bigger is better* » de Multics. Thompson avait l'intuition que « *c'était une bonne idée de sortir de Multics. C'était trop gros, trop cher, trop sophistiqué. Il était clair que c'était un exercice de construction d'usine à gaz.* »[18]

Mais Unix n'a pas été un simple rejet du concept de Multics : en effet, beaucoup d'idées de Multics ont été simplifiées et ajoutées à Unix. Ritchie souligne :

> Nous étions un peu oppressés par la mentalité « grand système ». Ken voulait faire quelque chose de simple. Unix n'est pas qu'une réaction contre Multics, c'était une combinaison de ces choses. Multics n'était plus là pour nous, mais nous appréciions le sentiment d'informatique interactive qu'il procurait. Ken avait des idées à creuser sur comment faire un système. Le matériel disponible ainsi que nos penchants

17 *Cf.* la page dédiée sur la plateforme de partage de code Github [URL : https://github.com/DoctorWkt/pdp7-unix].

18 Entretien de M. Mahoney avec Ken Thompson, 1989 [URL : https://www.princeton.edu/~hos/mike/transcripts/thompson.htm].

> nous ont incité à construire des petites choses élégantes plutôt que des choses grandioses.

Dans le temps, le système d'exploitation Unix allait épouser une philosophie de conception qui a été résumée par Doug Mcllroy :

> C'est la philosophie Unix : écrire des programmes pour faire une chose et la faire bien. Écrire des programmes pour travailler ensemble, collaborer. Écrire des programmes pour manipuler des flux de texte, parce que c'est une interface universelle (Raymond, 2003).

Avec la version PDP-7 d'Unix, Thompson, Ritchie et d'autres expérimentaient encore les concepts et structures qui allaient se développer en fin de compte et se cristalliser dans cette philosophie. Ils étaient aux prises avec la structure du système de fichiers, les mécanismes fondamentaux des processus Unix, les caractéristiques de l'interpréteur et la possibilité d'abstraire les opérations sur les périphériques en des opérations sur des fichiers génériques. Les deux derniers éléments qui allaient donner cette philosophie de « boîte à outils » à Unix, les tubes et le noyau portable, allaient arriver dans les quatre années suivantes.

En utilisant une plateforme matérielle dépassée, le PDP-7, et avec seulement 8 192 mots de mémoire, Thompson et Ritchie ont pu construire un système d'exploitation multitâches, multi-utilisateurs avec un système de fichiers multi-répertoires. L'Unix PDP-7 n'a pas seulement distillé et simplifié beaucoup de concepts d'autres systèmes (Multics, Project Genie), il a aussi introduit des innovations comme les liens physiques, les périphériques représentés comme des fichiers spéciaux et les redirections généralisées d'entrée/sortie. Beaucoup des caractéristiques, outils et appels système introduit dans l'Unix PDP-7 ont cours aujourd'hui encore dans les descendants comme BSD[19], et dans des réécritures très propres comme Linux. L'héritage de ce tout petit système d'exploitation vit encore, cinquante ans après sa création.

19 Berkeley Software Distribution – une famille de systèmes d'exploitation dérivés d'Unix, produits originellement à l'Université de Berkeley.

Bibliographie

Amstadt B. & Johnson M.K. (1994). « Wine ». *Linux Journal,* Issue 4, p. 3.

Brooks F.P. (1975). *The mythical man-month: essays on software engineering.* Reading, Mass. : Addison-Wesley.

Corbato F.J., Saltzer, J.H. & Clingen, C.T. (1972). « Multics: the first seven years ». *American Federation of Information Processing Societies: AFIPS Conference Proceedings,* 1972 Spring, Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, pp. 571-583.

Nyman L. & Laakso M. (2016). « Notes on the history of fork and join ». *IEEE Annals of the History of Computing* 38(3), 84-87.

Project MAC (1969). *The Multiplexed Information and Computing Service: Programmers' Manual.* Cambridge MA. : Mass. Inst. of Technology.

Raymond E.S. (2003). *The art of Unix programming.* Reading : Addison-Wesley Professional.

Richards M. (1969). « BCPL: A Tool for Compiler Writing and System Programming ». *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, AFIPS '69*, New York : ACM, pp. 557-566

Ritchie D.M. (1980). « The Evolution of the Unix Time-Sharing System ». *Proceedings of a Symposium on Language Design and Programming Methodology,* London, Springer-Verlag, pp. 25-36.

Ritchie D.M. (1993). « The Development of the C Language » in The Second ACMSIGPLAN Conference on History of Programming Languages, HOPL-II, New York, ACM, pp. 201-208.

Ritchie D.M. & Thompson K. (1974). « The UNIX Time-sharing System ». *Communications of the ACM* 17(7), pp. 365-375.

Saltzer J.H. (1965). *Manuscript Typing and Editing,* 2nd edition, Cambridge, Mass. : MIT Press, p. AH.9.01.

Salus P.H. (1994). *A Quarter Century of UNIX.* New York : ACM Press/Addison-Wesley Publishing Co.

Supnik B. & Walden D. (2015). « The Story of SimH ». *IEEE Annals of the History of Computing* 37(3), pp. 78-80.

Toomey W. (2009). « The Restoration of Early Unix Artifacts » in 2009 USENIX Annual Technical Conference (USENIX ATC 09), San Diego, USENIX Association.