TEMPS ET ACTIVITÉ SELON UNIX

Baptiste Mélès

La Découverte | « Réseaux »

2017/6 n° 206 | pages 125 à 153

ISSN 0751-7971
ISBN 9782707197740

Article disponible en ligne à l'adresse :

https://www.cairn.info/revue-reseaux-2017-6-page-125.htm

TEMPS ET ACTIVITÉ SELON UNIX

Baptiste MÉLÈS

DOI: 10.3917/res.206.0125

Les systèmes d'exploitation sont des programmes qui administrent les ressources matérielles de l'ordinateur – processeur, mémoires, entrées, sorties – tout en offrant à l'utilisateur une interface permettant d'interagir avec la machine. Derrière cette fonction technique se cachent des opérations ontologiques non triviales, que l'on entende le terme d'« ontologie » dans son sens informatique de structuration conceptuelle d'un domaine ou dans son sens philosophique de théorie de l'être. Nous entendrons en effet par ontologie l'ensemble de ce qui est reconnu dans un contexte donné comme existant et des structures qui en permettent la description et la manipulation.

Quelles sont ces opérations ontologiques non triviales ? Le système d'exploitation doit connaître à la fois l'ontologie du matériel informatique, composée d'un processeur (ainsi que, dans ce processeur, d'instructions, de registres, d'opérandes, d'adresses, etc.), d'une mémoire, d'entrées et de sorties diverses, et d'autre part une ou plusieurs ontologies de représentations humaines, composées par exemple de curseurs, de commandes et d'arguments, ou de souris, de fenêtres et d'icônes. Ces ontologies, qui sont autant de façons de concevoir le fonctionnement de l'ordinateur, s'expriment dans des langages distincts, typiquement le langage machine d'un côté et des langages d'actions et de constats textuels ou graphiques de l'autre. Le code source du système d'exploitation assure ainsi une tâche de traduction entre les ontologies hétérogènes du mécanicien et de l'ergonome.

Mais loin d'avoir lieu directement de langage à langage, cette traduction passe généralement par un langage tiers, idiomatique au système d'exploitation. L'ontologie qui s'y exprime, et que l'on peut voir comme celle d'une machine virtualisée (Krakowiak, 2014), ne calque ni celle du matériel ni celle de l'interface. Par exemple, les « processus » manipulés par le système d'exploitation n'existent pas au niveau du matériel et l'utilisateur ne manipule généralement pas cette notion, hormis dans un contexte d'administration système – c'est-à-dire précisément de redescente dans l'ontologie du système d'exploitation. De même, la notion de « fichier » qu'utilise le système d'exploitation ne correspond pas en toute généralité à ce que l'utilisateur entend

intuitivement par ce terme : plus d'une personne serait surprise d'apprendre qu'il y a des « fichiers » que l'on ne peut copier sur un disque dur[1]. L'ontologie « tierce » des systèmes d'exploitation s'exprime ainsi dans un système de concepts propre qui peut être décrit pour lui-même et qui, même s'il est, via son code source, plongé dans un autre, n'est pas déterminé par lui de façon univoque. Les créateurs de systèmes d'exploitation bénéficient et font usage d'une grande liberté conceptuelle.

De telles ontologies présentent un intérêt plus large que de simples choix techniques : il s'agit bien souvent de choix d'intelligibilité. Le code source d'un système d'exploitation représente ainsi pour le philosophe le cas de figure idéal d'un système de concepts incluant des notions aussi fondamentales que celles d'acte, d'objet, de temps et d'espace, toutes définies de manière entièrement explicite dans le langage d'une autre ontologie, elle-même définie sans ambiguïté. Le système des concepts d'un système d'exploitation peut dès lors être étudié à la façon dont les historiens structuraux de la philosophie, à la suite de Martial Gueroult, reconstruisent l'architecture d'un système philosophique.

Nous avons proposé ailleurs de voir la dualité entre processus et fichier sous Unix comme une contribution à l'étude métaphysique des relations entre l'acte et l'objet (Mélès, 2013). Dans la continuité de ce travail, nous traiterons ici du concept de temps propre au système d'exploitation Unix, en montrant d'une part comment il se distingue de celui de la machine et de l'autre comment il s'articule à la notion d'activité propre à Unix, celle de processus.

À cette fin, nous exposerons dans un premier temps la temporalité propre aux machines synchrones définies par le rapport sur l'EDVAC en 1945, avant de montrer la spécificité de la temporalité d'Unix en nous appuyant sur le code source de sa version 6.

LE TEMPS DES MACHINES SYNCHRONES DE VON NEUMANN

Pour pouvoir exposer la temporalité d'Unix, il nous faut d'abord exposer celle dont elle se distingue, et qui trouve son modèle dominant dans les « machines

1. C'est le cas des « fichiers spéciaux » sous Unix, qui incarnent typiquement des périphériques (Ritchie et Thompson, 1974, p. 367), ou des tubes (*pipes*), qui représentent la composition de processus successifs.

de von Neumann », définies en 1945 dans le « First Draft of a Report on the EDVAC » (von Neumann, 1945, section 2). Le rapport présente d'entrée de jeu les principaux ingrédients des machines dont il donne le modèle. Celles-ci doivent d'abord posséder une unité de contrôle arithmétique (CA) et de contrôle logique (CC), responsable de la partie algorithmique du travail. Cette unité effectue les calculs arithmétiques (addition, soustraction, multiplication, division, éventuellement racine carrée, logarithme, etc.) et s'oriente dans l'algorithme en prenant des décisions mécaniques. La machine doit ensuite posséder une mémoire (M), c'est-à-dire un espace de rangement pour les données initiales, intermédiaires et finales du calcul, les instructions du programme, les tables arithmétiques, etc. Il faut enfin y ajouter des entrées (I) comme le clavier et les cartes perforées, des sorties (O) comme un écran ou une imprimante, ainsi qu'une mémoire tampon pour les entrées et sorties (R). Il semble donc qu'il faille pour la machine de von Neumann ces cinq ingrédients et rien de plus.

Rien de plus ? Manifestement si, car avant de décrire l'usage concret de sa machine dans différents algorithmes élémentaires, notamment les quatre opérations fondamentales de l'arithmétique, le rapport consacre trois fois plus de pages à traiter le problème du *temps* dans les calculateurs automatiques : il décrit deux manières, dites respectivement asynchrone et synchrone, dont une machine peut réguler son rythme (*time itself*).

La manière « asynchrone » consiste à se rythmer « de manière autonome, selon les temps de réaction successifs de ses éléments ». Chacun de ses éléments fonctionnant à son rythme, « tous les stimuli doivent en dernière instance trouver leur origine dans l'entrée (*input*) » : dans le mode asynchrone, le temps de la machine est le temps propre de ses rouages, qui dépend lui-même de la temporalité des stimuli fournis en entrée. Il s'agit donc d'une temporalité essentiellement externe, pour ne pas dire subie : le temps est dicté par l'environnement et limité par la vitesse des rouages. C'est le temps du physicien expérimental, qui établit des correspondances extrinsèques entre la suite des phénomènes physiques observés d'une part et de l'autre une suite de phénomènes apparaissant sur l'instrument de mesure du temps. Dans ce cas de figure, le phénomène mesuré n'a cure du chronomètre : l'horloge cassée, le corps poursuit sa chute. Dans les machines asynchrones, telles que les présente le rapport sur l'EDVAC, le calcul suit un cours temporel analogue aux processus physiques, indépendant de tout éventuel instrument de mesure temporelle.

L'auteur du rapport reconnaît l'intérêt du mode asynchrone pour des machines « lentes », c'est-à-dire exécutant de 100 à 1000 instructions par seconde, et ce même si l'exécution de plusieurs opérations simultanées requiert de démultiplier d'autant l'espace occupé par certaines parties de la machine ; par exemple, pour exécuter deux additions simultanées, il faut dédoubler les organes nécessaires à l'addition pour éviter tout télescopage des deux processus. Pour des machines très rapides comme celles qui, à l'époque, utilisent des tubes à vide et dont la fréquence de calcul est de l'ordre du mégahertz (c'est-à-dire exécutant un million d'opérations par seconde), en revanche, le mode asynchrone s'impose moins nécessairement[2]. Le temps n'y étant pas une ressource rare, il est moins coûteux de s'y autoriser des moments de latence s'ils permettent en retour un gain en termes de prévisibilité.

> La seconde option, « synchrone », consiste en effet en ce que le rythme (*timing*) [des éléments] soit imprimé par une horloge fixée, impulsant certains stimuli nécessaires pour que l'appareil fonctionne à certains moments récurrents selon une périodicité définie. Cette horloge peut être un axe en rotation dans un appareil mécanique ou électromécanique ; et ce peut être un oscillateur électrique (éventuellement contrôlé par un cristal) dans un appareil purement électrique.
>
> (Section 4.1)

Dans le mode synchrone, le temps n'est plus dicté par le rythme des rouages. La machine a même tendance, du point de vue chronométrique, à perdre du temps si tel est le prix à payer pour garantir que toutes les opérations intermédiaires seront terminées au moment de passer d'un ensemble d'opérations « simultanées » au suivant.

Ayant présenté cette alternative entre un rythme décentralisé et un rythme centralisé, von Neumann opte pour la seconde option : « Si l'on peut se fier à la synchronisation de plusieurs suites distinctes d'opérations accomplies simultanément par l'appareil, le rythme imprimé par une horloge est évidemment préférable » (section 4.1). L'activité de la machine doit alors être rythmée par des impulsions périodiques : « Nous proposons d'utiliser les délais τ comme unités absolues de temps, sur lesquelles nous pouvons nous appuyer pour synchroniser les fonctions des différentes parties de l'appareil » (section 6.3). Cela suppose d'ajouter à la machine un sixième type d'organe, distinct des éléments

2. Par comparaison, la rapidité d'un ordinateur du commerce se mesure aujourd'hui en gigahertz (un milliard d'opérations par seconde).

proprement calculatoires envisagés jusqu'ici – l'unité arithmétique et logique, la mémoire, le tampon, les entrées et les sorties. Il s'agit de l'horloge.

> La meilleure façon de concevoir l'horloge centrale est de la voir comme un oscillateur électrique, qui émette à chaque période τ une brève impulsion standard d'une longueur τ' durant environ $(1/5)\tau - (1/2)\tau$. Les stimuli émis en propre par un élément sont en fait des impulsions de l'horloge, pour laquelle l'impulsion agit comme une porte. On a manifestement une grande liberté pour déterminer la période pendant laquelle la porte doit rester ouverte afin de pouvoir transmettre sans distorsion l'impulsion de l'horloge.
>
> (Section 6.3)

L'horloge porte donc un nom trompeur : même si son fonctionnement, vu de façon externe, repose en réalité sur la régularité d'un phénomène physique – typiquement l'oscillation émise par un cristal de quartz taillé de manière adéquate et traversé par un courant électrique –, elle ne donne pas l'heure comme si elle se contentait de la mesurer, mais elle la dicte à la machine. Du point de vue de la machine, ce n'est pas un organe qui *lit* l'heure, mais qui l'*écrit*.

Le choix des machines synchrones ou asynchrones implique bien davantage que les considérations d'efficacité technique avancées par von Neumann. Deux conceptions du rapport entre temps et activité s'y affrontent : l'une où l'événement se déroule indépendamment de l'instrument de mesure, l'autre où celui-ci exerce un rapport causal sur le déroulement de l'activité. Que l'on pense au coureur, qui utilise sa montre tantôt pour chronométrer ses performances, tantôt pour partir à la douche ; ou, dans un cadre dialogique, à l'incompréhension mutuelle entre l'orateur d'un colloque, concentré sur le temps que requiert intrinsèquement le déroulement de son propos, et l'organisateur consultant de plus en plus ostensiblement sa montre, instrument de mesure temporelle extrinsèque au discours. Pour désigner ces deux usages du temps, nous parlerons respectivement d'*horloge passive* et d'*horloge active*. Comme on le voit, ces deux conceptions du rapport entre temps et activité débordent largement le monde des ordinateurs, qui n'en est ici que l'occasion.

La voie des machines synchrones est celle qui est encore aujourd'hui majoritairement suivie dans l'industrie, entre autres pour des raisons qui tiennent aux réalisations physiques des différentes parties de la machine[3]. Il se trouve

3. Il existe des tentatives de machines asynchrones, inspirées par exemple des réflexions de John Backus (Backus, 1978). Très tôt, les machines ont également mélangé des parties synchrones

en effet que la vitesse de calcul du processeur dépasse largement celle des écritures et lectures en mémoire, elle-même généralement supérieure à celle des lectures et écritures dans les entrées et sorties. Or ces différents éléments ont besoin d'interagir selon des alternances strictement réglées : on doit pouvoir charger dans tel registre du processeur la valeur inscrite dans tel emplacement de la mémoire, ranger dans la mémoire la valeur contenue dans tel registre du processeur, consulter dans la mémoire la prochaine instruction à effectuer, écrire dans la sortie le résultat contenu dans le tampon, etc. Si l'on laissait libre cours à leurs temporalités propres, ces différentes activités se télescoperaient presque immédiatement. Le résultat, imprévisible d'un point de vue informatique, ne serait explicable que par le physicien.

Quitte à ce que le processeur « s'ennuie » très fréquemment, c'est-à-dire concède certains moments d'inactivité, le mode synchrone permet d'éviter la catastrophe. Nos ordinateurs sont ainsi généralement munis d'un oscillateur à cristal de quartz qui, décrémentant la valeur d'un compteur jusqu'à parvenir à une valeur nulle, envoie périodiquement des impulsions, dont chacune entraîne l'effectuation d'un ensemble d'opérations élémentaires « simultanées »[4], ou du moins suffisamment proches pour paraître telles selon le grain temporel adopté. Un processeur à trois gigahertz est ainsi capable d'effectuer trois milliards d'instructions par seconde, ce qui suppose simplement qu'à chaque impulsion la machine soit parvenue dans un état stable, comme dans le jeu d'enfants « un, deux, trois, soleil », où toutes les « souris » doivent être immobiles lorsque, périodiquement, le « chat » se retourne, mais peuvent se mouvoir librement aussitôt qu'il leur tourne le dos. Dans les machines synchrones comme, plus généralement, dans une large classe d'activités, le temps physique est ainsi découpé par une horloge en vue de rythmer une activité elle-même segmentée en opérations élémentaires.

---

et des parties asynchrones. Le PDP-11/40, par exemple, inclut une dose d'asynchronisme (*PDP11/40 Processor Handbook*, p. 1-1).

4. Plus précisément, *environ* une opération élémentaire est effectuée. Parfois, plusieurs opérations sont effectuées, si la machine permet le parallélisme au niveau du processeur. Parfois, à l'inverse, n'est exécutée qu'une portion d'opération élémentaire, lorsque le processeur est muni d'un jeu d'instructions complexes ; c'est typiquement le cas des processeurs CISC, par opposition aux processeurs de type RISC, dont les opérations tiennent généralement en un cycle d'horloge. Les choses sont encore plus complexes de nos jours, où CISC et RISC se mélangent.

Le niveau du processeur permet de montrer clairement le lien entre temporalité et activité, restreignant cependant ce dernier terme à la notion d'opération élémentaire. Que devient donc le temps lorsque l'activité change d'échelle ?

LE TEMPS DES PROCESSUS SELON UNIX

Nous étudierons la temporalité informatique de haut niveau en examinant le code source de la version 6 d'Unix (Unix 1975), un système certes « ancien », mais déjà porteur des principaux concepts qui structurent encore aujourd'hui une large famille de systèmes d'exploitation, dont Linux et la famille des systèmes BSD.

Cette version d'Unix, la première diffusée hors d'AT&T, joua pour de nombreuses raisons un rôle historique important. D'abord, elle décrivait un système d'exploitation conceptuellement élégant, mettant en application de nombreuses idées élaborées au début des années 1960 pour les systèmes d'exploitation multitâches et multi-utilisateurs CTSS et MULTICS (Ritchie, 1978 ; Walden et Van Vleck, 2011). Le premier concepteur d'Unix, Ken Thompson, contribuait en effet au projet MULTICS jusqu'à ce que son employeur, Bell, quitte le consortium en 1969 ; son équipe – Rudd Canaday, Douglas McIlroy, Joseph Ossanna et Dennis Ritchie – eut alors, à défaut de moyens, du moins les coudées franches pour mettre en œuvre ses idées (Ritchie, 1984).

Deuxième raison de son succès, cette version était écrite pour les PDP-11/40, PDP-11/45 et PDP-11/70 de DEC (1972-1975), ordinateurs polyvalents et bon marché, à ce titre largement répandus dans les universités américaines. De plus, la loi anti-trust interdisant à AT&T, en situation de monopole, de commercialiser Unix, la version 6 d'Unix se diffusa librement dans les universités et institutions de recherche (Lazard et Mounier-Kuhn, 2016, pp. 173-175). Cette rencontre entre les laboratoires de recherche Bell et la communauté universitaire constitua la matière d'un « passionnant conte sociologique » (Salus, 1994, p. 1 et *passim*).

Enfin, la version 6 d'Unix fut popularisée par le commentaire de John Lions, qui dès 1977 « comment[a] le code source d'UNIX sur un style habituellement réservé aux exégèses de Chaucer ou Shakespeare » (Tanenbaum, 2003, p. 713). Ce commentaire, qui longtemps circula sous le manteau, est aujourd'hui encore un instrument indispensable pour qui veut étudier de près la structure d'Unix (Lions, 1996).

Telles sont les raisons pour lesquelles nous avons choisi la version 6 pour étudier le temps selon Unix. Or rien ne permet mieux de saisir l'ontologie d'un système d'exploitation que la liste de ses « appels système », c'est-à-dire des types d'invocation possibles du mode noyau par le mode utilisateur. Comme l'affirme très justement Marc Rochkind, « les primitives [ou appels système] définissent Unix » (Rochkind, 1987, p. 1). Chacun des appels système est pour l'utilisateur une façon de tirer par la manche le noyau du système d'exploitation et de lui demander l'autorisation d'effectuer une action, sachant qu'il a tous les droits et qu'il décide souverainement de ce que l'utilisateur a ou non le droit de faire.

De quoi parlent donc les appels système ? Pour le savoir, consultons le fichier `sysent.c` du code source d'Unix. Ce fichier contient un tableau associant à chaque appel système la routine qui le définit dans des fichiers tels que `sys1.c` et `sys4.c`. Afin de guider la lecture, précisons que les textes insérés entre les suites de caractères « `/*` » et « `*/` » sont des commentaires, c'est-à-dire des indications destinées au lecteur humain qui ne sont pas exécutées par la machine. Le fichier possède une structure très simple : il se compose d'un tableau d'entiers qui, à la ligne 2910, reçoit le nom de `sysent` et qui contient à chaque ligne, comme l'indique le commentaire initial (l. 2907-2908), le nombre d'arguments attendus (par exemple `0`), l'adresse de la fonction à appeler (par exemple `&nullsys`, c'est-à-dire l'adresse enregistrée dans une variable nommée `nullsys`) et un commentaire indiquant le numéro et le nom de l'appel système (par exemple `indir` pour l'appel système `0`). Les appels système sont numérotés automatiquement à partir de `0`. Voyons donc ce qu'un coup d'œil superficiel sur ce fichier peut déjà nous apprendre sur la liste des appels système, partant sur l'ontologie d'Unix.

```
2904: /*
2905:  * This table is the switch used to transfer
2906:  * to the appropriate routine for processing a system call.
2907:  * Each row contains the number of arguments expected
2908:  * and a pointer to the routine.
2909:  */
2910: int     sysent[]
2911: {
2912:         0, &nullsys,            /*  0 = indir */
2913:         0, &rexit,              /*  1 = exit */
2914:         0, &fork,               /*  2 = fork */
2915:         2, &read,               /*  3 = read */
2916:         2, &write,              /*  4 = write */
2917:         2, &open,               /*  5 = open */
2918:         0, &close,              /*  6 = close */
2919:         0, &wait,               /*  7 = wait */
```

```
2920:         2, &creat,              /*  8 = creat */
2921:         2, &link,               /*  9 = link */
2922:         1, &unlink,             /* 10 = unlink */
2923:         2, &exec,               /* 11 = exec */
2924:         1, &chdir,              /* 12 = chdir */
2925:         0, &gtime,              /* 13 = time */
2926:         3, &mknod,              /* 14 = mknod */
2927:         2, &chmod,              /* 15 = chmod */
2928:         2, &chown,              /* 16 = chown */
2929:         1, &sbreak,             /* 17 = break */
2930:         2, &stat,               /* 18 = stat */
2931:         2, &seek,               /* 19 = seek */
2932:         0, &getpid,             /* 20 = getpid */
2933:         3, &smount,             /* 21 = mount */
2934:         1, &sumount,            /* 22 = umount */
2935:         0, &setuid,             /* 23 = setuid */
2936:         0, &getuid,             /* 24 = getuid */
2937:         0, &stime,              /* 25 = stime */
2938:         3, &ptrace,             /* 26 = ptrace */
2939:         0, &nosys,              /* 27 = x */
2940:         1, &fstat,              /* 28 = fstat */
2941:         0, &nosys,              /* 29 = x */
2942:         1, &nullsys,            /* 30 = smdate; inoperative */
2943:         1, &stty,               /* 31 = stty */
2944:         1, &gtty,               /* 32 = gtty */
2945:         0, &nosys,              /* 33 = x */
2946:         0, &nice,               /* 34 = nice */
2947:         0, &sslep,              /* 35 = sleep */
2948:         0, &sync,               /* 36 = sync */
2949:         1, &kill,               /* 37 = kill */
2950:         0, &getswit,            /* 38 = switch */
2951:         0, &nosys,              /* 39 = x */
2952:         0, &nosys,              /* 40 = x */
2953:         0, &dup,                /* 41 = dup */
2954:         0, &pipe,               /* 42 = pipe */
2955:         1, &times,              /* 43 = times */
2956:         4, &profil,             /* 44 = prof */
2957:         0, &nosys,              /* 45 = tiu */
2958:         0, &setgid,             /* 46 = setgid */
2959:         0, &getgid,             /* 47 = getgid */
2960:         2, &ssig,               /* 48 = sig */
2961:         0, &nosys,              /* 49 = x */
2962:         0, &nosys,              /* 50 = x */
[...]
2976: };
```

On reconnaît, dans cette liste, des appels système liés aux processus (`exec` et `fork` pour créer des processus, `setuid` et `getuid` pour gérer les droits des utilisateurs sur un processus, `nice` pour définir sa priorité d'exécution, `kill` pour lui envoyer un signal…) et d'autres au système de fichiers (`unlink` pour effacer un fichier, `chdir` pour changer de répertoire, `chmod` pour modifier les droits sur un fichier, `chown` pour changer le propriétaire d'un fichier…).

Jusqu'ici, tout semble confirmer les propos d'Andrew Tanenbaum, répartissant les appels système « en gros en deux grandes catégories : ceux qui se rapportent à des processus et ceux qui se rapportent au système de fichiers » (Tanenbaum et Woodhull, 1997, p. 15). Mais dans le système MINIX de Tanenbaum comme dans Unix, dont il dérive, l'expression « en gros » cache l'ensemble des appels système relatifs au temps tels que `stime` et `gtime`. Pour le système d'exploitation, le temps semble ainsi une notion aussi fondamentale que celles d'acte et d'objet[5].

Mais de quel temps s'agit-il ? Est-ce le même qu'au niveau du boîtier, le temps de l'horloge de la machine synchrone de von Neumann ?

TYPOLOGIE DES HORLOGES

Pour comprendre le temps propre au noyau d'Unix, il est nécessaire de bien distinguer trois horloges que peuvent contenir – et que contiennent généralement aujourd'hui – nos ordinateurs (Bovet et Cesati, 2001, ch. 5).

La première horloge, appelée TSC (*Time Stamps Counter*), est l'horloge active avec oscillateur à quartz que nous avons décrite plus haut. C'est l'horloge informatique qui permet la synchronisation de bas niveau des opérations. Une deuxième horloge, appelée RTC (*Real Time Clock*), est celle qui donne la date et l'heure ; elle possède une pile qui lui permet de continuer à fonctionner en cas de coupure de courant. C'est donc l'horloge passive qui compte le temps physique. La troisième horloge, qui va désormais nous intéresser, s'appelle PIT (*Programmable Interval Timer*). Cette horloge active régit la synchronisation de haut niveau, c'est-à-dire non plus à l'échelle des opérations élémentaires de la machine, mais au degré d'abstraction plus élevé du système d'exploitation.

Le PIT est programmé pour déclencher périodiquement une interruption, c'est-à-dire envoyer un signal spécifique au noyau du système d'exploitation. La période est fixée dans les fichiers d'en-tête du pilote de l'horloge,

5. Ce qui vaut pour Unix vaut encore pour ses héritiers, même si le nombre d'appels système s'y est considérablement élargi (en juin 2017, ils sont plus de 300 dans Linux 4.11.8). Linux possédait ainsi en 2000, dans sa version 2.2.14, des appels `time`, `ftime` et `gettimeofday` pour connaître la date et l'heure, `settimer` et `alarm` pour l'envoi de signaux périodiques, `adjtimex` pour la synchronisation d'horloges (Bovet et Cesati, 2001, ch. 5).

c'est-à-dire de la partie du code source décrivant comment le système doit interagir avec elle. La période ne résulte donc généralement pas, comme celle du TSC, d'une contrainte matérielle, mais d'une décision logicielle. On trouve ainsi dans le fichier `param.h` d'Unix, qui fixe les constantes du système, une section introduite par le commentaire

```
0128: /* tunable variables */
```

qui ne contient pas, comme on pourrait le croire, des « variables » au sens propre, c'est-à-dire des valeurs susceptibles de changer au cours de l'exécution du programme, mais plus précisément des constantes arbitraires[6], parmi lesquelles la valeur

```
0147: #define HZ      60          /* Ticks/second of the clock */
```

qui dicte la fréquence, sans doute choisie empiriquement, des signaux envoyés par l'horloge programmable[7]. Comme on peut le constater, cette échelle temporelle de 10^{-2} seconde est d'un tout autre ordre de grandeur que celle de 10^{-9} qui rythme l'activité du processeur : le temps du système d'exploitation est un temps long. Comme l'activité humaine, celle de l'ordinateur est structurée selon différents degrés de temporalité.

L'horloge est, au haut niveau du système d'exploitation comme au niveau élémentaire de la machine synchrone de von Neumann, l'un des très rares éléments indispensables au fonctionnement de l'ordinateur. Nous écrivions plus haut qu'un chronomètre cassé n'arrêtait pas la chute d'un corps ; nous allons voir qu'il en va tout autrement de la version 6 d'Unix en nous appuyant sur l'analyse de sa fonction principale, `main()`, définie dans le fichier `main.c`. Cette fonction très courte, que l'on trouvera ci-dessous, décrit en une centaine

---

6. Ce vocable impropre permet de distinguer psychologiquement ces constantes arbitraires d'autres constantes, appelées respectivement « constantes fondamentales » (lignes 100-107) et « signaux » (lignes 110-126), précédées en commentaire des avertissements apotropaïques « *fundamental constants: cannot be changed* » et « *signals: dont change* ».

7. Cette valeur a peu changé avec les années : si les concepteurs d'Unix optaient en 1975 pour une fréquence de 60 interruptions par seconde, Linux fixe aujourd'hui pour la plupart des machines une fréquence de 100 interruptions par seconde. L'ordre de grandeur n'a donc pour ainsi dire guère varié en quarante ans. Voir, dans le code source de Linux, les fichiers `param.h` des sous-répertoires de `include/` relatifs aux différentes architectures (Linux, 2000 ; Linux, 2017).

de lignes – lignes blanches et commentaires compris – le cœur du système d'exploitation.

```
1532: /*
1533:  * Initialization code.
1534:  * Called from m40.s or m45.s as
1535:  * soon as a stack and segmentation
1536:  * have been established.
1537:  * Functions:
1538:  *      clear and free user core
1539:  *      find which clock is configured
1540:  *      hand craft 0th process
1541:  *      call all initialization routines
1542:  *      fork - process 0 to schedule
1543:  *           - process 1 execute bootstrap
1544:  *
1545:  * panic: no clock -- neither clock responds
1546:  * loop at loc 6 in user mode -- /etc/init
1547:  *      cannot be executed.
1548:  */
1549:
1550: main()
1551: {
1552:         extern schar;
1553:         register i, *p;
1554:
1555:         /*
1556:          * zero and free all of core
1557:          */
1558:
1559:         updlock = 0;
1560:         i = *ka6 + USIZE;
1561:         UISD->r[0] = 077406;
1562:         for(;;) {
1563:                 UISA->r[0] = i;
1564:                 if(fuibyte(0) < 0)
1565:                         break;
1566:                 clearseg(i);
1567:                 maxmem++;
1568:                 mfree(coremap, 1, i);
1569:                 i++;
1570:         }
1571:         if(cputype == 70)
1572:         for(i=0; i<62; i=+2) {
1573:                 UBMAP->r[i] = i<<12;
1574:                 UBMAP->r[i+1] = 0;
1575:         }
1576:         printf("mem = %l\n", maxmem*5/16);
1577:         printf('RESTRICTED RIGHTS\n\n');
1578:         printf('Use, duplication or disclosure is subject to\n');
1579:         printf('restrictions stated in Contract with Western\n');
1580:         printf('Electric Company, Inc.\n');
1581:
1582:         maxmem = min(maxmem, MAXMEM);
1583:         mfree(swapmap, nswap, swplo);
1584:
1585:         /*
1586:          * set up system process
1587:          */
```

```
1588:
1589:         proc[0].p_addr = *ka6;
1590:         proc[0].p_size = USIZE;
1591:         proc[0].p_stat = SRUN;
1592:         proc[0].p_flag =| SLOAD|SSYS;
1593:         u.u_procp = &proc[0];
1594:
1595:         /*
1596:          * determine clock
1597:          */
1598:
1599:         UISA->r[7] = ka6[1]; /* io segment */
1600:         UISD->r[7] = 077406;
1601:         lks = CLOCK1;
1602:         if(fuiword(lks) == -1) {
1603:                 lks = CLOCK2;
1604:                 if(fuiword(lks) == -1)
1605:                         panic("no clock");
1606:         }
1607:         *lks = 0115;
1608:
1609:         /*
1610:          * set up 'known' i-nodes
1611:          */
1612:
1613:         cinit();
1614:         binit();
1615:         iinit();
1616:         rootdir = iget(rootdev, ROOTINO);
1617:         rootdir->i_flag =& ~ILOCK;
1618:         u.u_cdir = iget(rootdev, ROOTINO);
1619:         u.u_cdir->i_flag =& ~ILOCK;
1620:
1621:         /*
1622:          * make init process
1623:          * enter scheduling loop
1624:          * with system process
1625:          */
1626:
1627:         if(newproc()) {
1628:                 expand(USIZE+1);
1629:                 estabur(0, 1, 0, 0);
1630:                 copyout(icode, 0, sizeof icode);
1631:                 /*
1632:                  * Return goes to loc. 0 of user init
1633:                  * code just copied out.
1634:                  */
1635:                 return;
1636:         }
1637:         sched();
1638: }
```

LA CONSTITUTION D'UN ESPACE

La première tâche de la fonction principale d'Unix est d'initialiser et de libérer la mémoire, c'est-à-dire l'espace disponible pour accueillir les données[8]. Celle-ci est répartie sur deux lieux : une mémoire principale (*core*) et une mémoire d'échange (*swap*), portion de la mémoire de masse réservée pour servir d'extension à la précédente.

Le segment initial de la mémoire principale, étant occupé par le système d'exploitation en cours d'exécution, ne doit évidemment pas être initialisé sous peine de catastrophe. Il faudra également réserver une place de taille `USIZE`[9] pour le descripteur de propriétaire d'un processus particulier (ligne 1560) que nous retrouverons plus loin. Pour initialiser la mémoire principale (`coremap`), la fonction `main()` doit donc choisir comme point de départ, indexé par la variable `i`, le bloc qui se trouve à la fin de l'espace occupé par le système d'exploitation (`*ka6`), auquel on ajoute un nombre `USIZE` d'octets :

```
1560:          i = *ka6 + USIZE;
```

Ensuite, aussi longtemps qu'elle arrive à lire (avec la fonction `fuibyte()`, ligne 1564) le contenu du premier mot du bloc choisi – autrement dit, aussi longtemps que l'indice renvoie vers un emplacement de la mémoire existant –, elle efface le contenu du bloc concerné avec une fonction nommée `clearseg()` (ligne 1566), incrémente le compteur indiquant la taille maximale autorisée par processus (`maxmem++`, ligne 1567), enfin ajoute au cadastre des zones libres le bloc initialisé, qui est de taille 1, et passe au bloc suivant :

```
1568:                  mfree(coremap, 1, i);
1569:                  i++;
```

Après quelques lignes de code propres au PDP-11/70 (lignes 1571-1575), la fonction affiche l'espace mémoire disponible puis des informations légales (lignes 1576-1580) et limite l'espace mémoire maximal autorisé par processus au moyen d'une constante `MAXMEM` (définie dans le fichier `param.h`). En résumé, pour initialiser la mémoire, Unix commence par tester si l'emplacement visé existe, puis efface son contenu, enfin met à jour le registre des

8. Le commentaire de la ligne 1556, trompeur, pourrait laisser croire qu'il ne s'agit d'« initialiser et de libérer » que la mémoire principale (*core*) ; mais la mémoire d'échange (*swap*) est également concernée.

9. La constante `USIZE` est fixée dans le fichier `systm.h` à 16 blocs.

emplacements disponibles en mémoire. C'est seulement à la fin de ce parcours que le système d'exploitation connaît, et peut afficher, les dimensions de la mémoire principale. Nous ne saurions trop recommander au lecteur curieux d'utiliser un émulateur de PDP-11[10] : il verra s'afficher la taille de la mémoire, puis les mentions légales, enfin l'invite de commande (le caractère « # ») :

```
mem = 1042
RESTRICTED RIGHTS
Use, duplication or disclosure is subject to
restrictions stated in Contract with Western
Electric Company, Inc.
#
```

La partie libre de la mémoire principale (`coremap`) ayant ainsi été initialisée, la fonction `main()` ajoute d'un seul coup à la liste des zones mémoire disponibles l'intégralité de la mémoire d'échange (`swapmap`), qui contient le nombre `nswap` de blocs, dont le premier est indexé par la valeur `swplo` :

```
1583:          mfree(swapmap, nswap, swplo);
```

La première tâche de la fonction principale d'Unix est ainsi de réinitialiser l'intégralité de la mémoire – à l'exception de son segment initial, réservé à l'administration – et de mettre à jour le cadastre des emplacements disponibles.

L'ontologie du système d'exploitation est ainsi prioritairement munie d'un espace. Mais peut-on parler d'un réel changement par rapport à l'ontologie sous-jacente de la machine, dans laquelle, après tout, la mémoire constitue aussi un espace ? En réalité, les propriétés de cet espace se sont profondément transformées. Avant ces quelques lignes, l'espace mémoire est une *terra incognita* dont on ignore la taille et dont le contenu est imprévisible. Le système d'exploitation ne possède à son sujet que deux connaissances : le *point distingué* qu'est la position initiale indexée par

```
1560:          i = *ka6 + USIZE;
```

et une *topologie discrète, inidimensionnelle et finie*, garantie de ce que l'espace puisse être parcouru intégralement par la suite des incrémentations `i++`

10. Il existe par exemple un émulateur disponible à l'adresse http://pdp11.aiju.de/ (page consultée le 5 septembre 2017). Pour lancer la version 6 d'Unix, l'utilisateur doit appuyer sur le bouton run et taper `unix` puis Entrée.

jusqu'à ce que la poursuite de l'exploration échoue. Le système ne connaît donc de cet espace incontrôlé que des propriétés topologiques très générales. Après l'exécution de ces quelques lignes de code, à l'inverse, non seulement le contenu de cet espace est sous contrôle, puisqu'il a été expressément remis à zéro, mais nous en possédons en outre le cadastre complet, actuellement composé de deux grandes zones libres – celle de la mémoire principale et celle de la mémoire d'échange – dont on pourra ensuite allouer des portions de taille maximale `maxmem` à la pile de données de chaque processus. En d'autres termes, l'espace simplement topologique – encore indéfini et inexploré – de la machine a cédé la place à un espace métrique – cette fois mesuré et balisé – du système d'exploitation.

LE PREMIER MOTEUR

Ainsi muni d'un espace sous contrôle, le système passe à la deuxième tâche majeure : l'initialisation du processus 0, processus fondamental du système d'exploitation, ancêtre de tous les autres.

Nous autres lecteurs du code source savons que ce processus est, du point de vue de la machine, créé – en l'occurrence par l'exécution de la fonction `main()`. Mais du point de vue interne du système d'exploitation, il présente toutes les apparences d'un « premier moteur » aristotélicien (Aristote, 2014, ch. 5). Il est en effet l'acte qui met en branle tous les autres sans être lui-même mû par aucun, hypothèse dont ni Aristote ni Unix ne sauraient faire l'économie, inscrivant l'un comme l'autre les processus causaux en une arborescence tout en s'interdisant le recours à l'infini actuel d'une régression dans le temps. L'arbre doit donc posséder une racine.

Ce premier moteur possède toutes les caractéristiques d'un acte en général, c'est-à-dire en l'occurrence d'un processus : un descripteur de propriétaire et un descripteur de processus. Comme il était prévu plus haut dans la fonction `main()` (ligne 1560), le descripteur du propriétaire de ce processus s'inscrit en effet immédiatement à la suite de l'espace mémoire occupé par le noyau (`*ka6`) et se voit réserver un espace dont la taille est fixée par la constante `USIZE` :

```
1589:           proc[0].p_addr = *ka6;
1590:           proc[0].p_size = USIZE;
```

Le statut du processus 0 est fixé comme « prêt à être exécuté » (SRUN) et deux attributs (*flags*) lui sont accolés, l'un indiquant que le descripteur de propriétaire de ce processus se trouve dans la mémoire principale (SLOAD), l'autre que ce processus ne doit jamais être relégué dans la mémoire d'échange (SSYS) :

```
1591:          proc[0].p_stat = SRUN;
1592:          proc[0].p_flag =| SLOAD|SSYS;
```

Enfin, le descripteur du processus est associé au descripteur de son propriétaire :

```
1593:          u.u_procp = &proc[0];
```

Toutes les informations relatives au processus 0, et notamment ses privilèges, ont ainsi été renseignées. Si les actes sous Unix sont très exactement les processus, le lancement du système vient d'instituer le premier acte digne de ce nom. Il sera désormais impossible, dans le langage interne du système, de remonter plus haut dans la genèse.

LE TEMPS PROPRE D'UNIX

À ce stade, l'ontologie d'Unix comprend donc un espace muni d'une mesure et un acte qui, du point de vue du système, présentera tous les attributs d'un premier moteur. La troisième grande tâche à accomplir est de fixer la temporalité du système d'exploitation.

Quelques précisions sur le matériel s'imposent ici – hélas[11] ! Le PDP-11, ordinateur auquel était exclusivement destinée la version 6 d'Unix, pouvait posséder deux types d'horloges : soit une horloge à fréquence de ligne KW11-L, réglée sur le courant électrique et d'adresse physique 777546, soit l'horloge programmable à temps réel KW11-P, contrôlable à l'adresse physique 772540[12]. Unix doit déterminer au plus vite l'horloge qu'il utilisera.

---

11. Ce particularisme matériel a fort heureusement disparu des versions ultérieures d'Unix, comme de Minix et de Linux, dont le code vise à la plus grande généralité. Dès la version 7 d'Unix, le code était rendu plus général, condition nécessaire pour pouvoir porter le système sur Interdata 8/32 (Tanenbaum, 2003, p. 713).

12. Une horloge à fréquence de ligne reçoit ses impulsions du courant électrique à 110 ou 220 volts et provoque une interruption à chaque cycle de voltage, à 50 ou 60 hertz (Tanenbaum et Woodhull, 1997, p. 223 ; DEC 1972a, p. E-5 ; DEC 1972b ; DEC 1973, p. 2-1a).

Le fichier de la fonction principale `main.c` a précédemment identifié par des constantes `CLOCK1` et `CLOCK2` les adresses, à translation près[13], des deux horloges possibles :

```
1509: #define CLOCK1  0177546
1510: #define CLOCK2  0172540
```

Il faut savoir si au moins l'une des deux horloges est disponible, et laquelle. Le système commence ainsi par inscrire dans un pointeur nommé `lks` l'adresse de l'horloge `CLOCK1` :

```
1601:         lks = CLOCK1;
```

S'il échoue à lire la valeur stockée à cette adresse au moyen de la fonction `fuiword()`, il se rabat sur `CLOCK2` :

```
1602:         if(fuiword(lks) == -1) {
1603:                 lks = CLOCK2;
[...]
1606:         }
```

S'il échoue également à trouver cette seconde horloge, il déclenche une sortie « en panique » :

```
1604:                 if(fuiword(lks) == -1)
1605:                         panic("no clock");
```

Le cas de figure est rarissime : la fonction `panic()`, qui bloque l'intégralité du système, ne figure que très exceptionnellement dans le code source d'Unix, système conçu pour fonctionner dans les conditions les plus extrêmes. Autant dire que l'horloge est l'un des très rares organes absolument indispensables au fonctionnement du système. Si l'une des deux horloges est présente, quelle qu'elle soit, elle est remise à zéro par l'instruction

```
1607:         *lks = 0115;
```

qui fixe sa valeur au nombre octal 115[14]. À partir de cet instant précis, le système est muni d'une temporalité rythmée par des interruptions régulières.

---

13. Comme le relève John Lions, « les adresses de l'intervalle 0160000-0177777 sont transposées vers l'intervalle 0760000-0777777 » (Lions, 1996, p. 2-5c).

14. Dans le langage C, en 1975 comme aujourd'hui, un nombre précédé d'un 0 est un nombre octal (Lions, 1996, p. 3-5a ; Kernighan et Ritchie, 2004, p. 37). Le nombre 0115 équivaut donc

Les conditions générales de possibilité du système d'exploitation sont ainsi réunies : un espace métrique, un premier moteur, une temporalité structurée. Le système peut dès lors entrer dans son fonctionnement normal, comme on passe de la métaphysique à la physique. C'est sur cette base que la fonction principale d'Unix initialise le système de fichiers (lignes 1609-1619), crée le processus d'initialisation ou processus 1, père de tous les processus normaux (ligne 1627)[15], et lance la boucle infinie d'ordonnancement des processus :

```
1637:          sched();
```

Toutes ces opérations, cruciales pour le système d'exploitation et seules accessibles dans l'expérience de l'utilisateur, ne sont possibles que sur la base des trois transcendantaux – espace, premier moteur, temporalité – qui les précèdent.

OPÉRATIONS PÉRIODIQUES ET DEGRÉS DE PRIORITÉ

Reste à savoir précisément quel usage il est fait de la temporalité, car « le travail de l'horloge matérielle se réduit à engendrer des interruptions à des intervalles donnés. Toute autre tâche impliquant le temps doit être réalisée par le logiciel, le pilote d'horloge » (Tanenbaum et Woodhull, 1997, p. 224). C'est en effet le pilote d'horloge, programme chargé de faire interagir le noyau du système d'exploitation avec ce périphérique particulier, qui décrit l'ensemble des opérations déclenchées par chaque interruption.

Le pilote est décrit dans la fonction `clock()` du fichier `clock.c` [16]. Toujours intempestif, ce processus en interrompt nécessairement un autre – soit le processus 1, soit une instance d'exécution précédente du pilote d'horloge, soit n'importe quel autre processus ; le processus interrompu est, dans le code du pilote, désigné génériquement par le nom `ps`.

---

au nombre décimal 77, mais nous avons échoué à trouver dans les manuels du KW11-L et du KW11-P (DEC 1973 ; DEC 1972b) la raison d'être de cette valeur.

15. Comme l'observe John Lions, les lignes 1628-1635 ne sont curieusement jamais exécutées puisque la fonction `newproc()`, définie dans le fichier `slp.c` aux lignes 1826-1919, renvoie toujours 0 (Lions, 1996, p. 6.4a).

16. Dans Linux, les fichiers concernés sont principalement `kernel/time.c` et `kernel/sched.c`.

Le début de la fonction `clock()` est exécuté avec une priorité de 6. Aucun autre périphérique ne bénéficie d'un tel degré d'urgence – autrement dit, rien n'est plus urgent que le temps (Lions 1996, p. 11-1a). Le pilote peut donc presque instantanément réinitialiser l'horloge, quelle qu'elle soit, au moyen de la commande

```
3730:         /*
3731:          * restart clock
3732:          */
3733:
3734:         *lks = 0115;
```

dont nous avons déjà vu la première occurrence dans la fonction principale d'Unix (ligne 1607). Le système met ensuite à jour la console système des PDP-11/45 et PDP-11/70 (ligne 3740). Il exécute de rares fonctions absolument prioritaires, toutes liées à l'envoi de caractères sur le terminal[17]. Ces fonctions et leurs arguments sont enregistrés dans un tableau de structures du nom de `callout`, défini dans le fichier `systm.h`. Lorsqu'une fonction doit attendre 0 top, c'est qu'elle doit être exécutée instantanément ; dans tout autre cas, on décrémente à chaque top le nombre de tops d'attente qu'il lui reste. Le tableau `callout` doit donc contenir la liste des fonctions à exécuter, de leurs arguments et du nombre de tops d'attente qu'il reste à chacune :

```
0253: /* The callout structure is for a routine
0254:  * arranging to be called by the clock interrupt
0255:  * (clock.c) with a specified argument,
0256:  * within a specified amount of time.
0257:  * It is used, for example, to time tab delays
0258:  * on teletypes. */
0259:
0260: struct  callo
0261: {
0262:         int     c_time;         /* incremental time */
0263:         int     c_arg;          /* argument to routine */
0264:         int     (*c_func)();    /* routine */
0265: } callout[NCALL];
```

Une partie importante du pilote d'horloge est consacrée à la gestion des fonctions prioritaires. Nous en détaillerons le fragment suivant :

17. Cette structure ne sert en effet que pour la fonction `ttrstrt` (Lions, 1996, pp. 11-1b et 25-3bc).

```
3742:          /*
3743:           * callouts
3744:           * if none, just return
3745:           * else update first non-zero time
3746:           */
3747:
3748:          if(callout[0].c_func == 0)
3749:                  goto out;
3750:          p2 = &callout[0];
3751:          while(p2->c_time<=0 && p2->c_func!=0)
3752:                  p2++;
3753:          p2->c_time--;
3754:
3755:          /*
3756:           * if ps is high, just return
3757:           */
3758:
3759:          if((ps&0340) != 0)
3760:                  goto out;
3761:
3762:          /*
3763:           * callout
3764:           */
3765:
3766:          spl5();
3767:          if(callout[0].c_time <= 0) {
3768:                  p1 = &callout[0];
3769:                  while(p1->c_func != 0 && p1->c_time <= 0) {
3770:                          (*p1->c_func)(p1->c_arg);
3771:                          p1++;
3772:                  }
3773:                  p2 = &callout[0];
3774:                  while(p2->c_func = p1->c_func) {
3775:                          p2->c_time = p1->c_time;
3776:                          p2->c_arg = p1->c_arg;
3777:                          p1++;
3778:                          p2++;
3779:                  }
3780:          }
[...]
3787: out:
```

Pour exécuter les fonctions prioritaires – sous réserve que la liste n'en soit pas vide (lignes 3748-3749) –, le pilote d'horloge décrémente le nombre de tops d'attente de chaque fonction listée dans `callout` (lignes 3750-3753). Il peut alors passer à une opération légèrement moins urgente, exécutée en priorité 5 (ligne 3766) et seulement si le processus `ps` qu'a interrompu le processus courant du pilote d'horloge n'est pas prioritaire sur lui (lignes 3755-3760). Il s'agit de l'exécution des fonctions dont le nombre restant de tops d'attente vaut 0 (lignes 3767-3780). En résumé, le système considère à intervalles réguliers l'ensemble des tâches à effectuer afin de les mener à bien selon leur ordre de priorité, sachant que la priorité absolue revient précisément aux tâches temporelles.

Se trouvant toujours dans une priorité élevée de 5 ou 6, le pilote accomplit ensuite d'autres tâches relatives au temps. Il incrémente le décompte `u_utime` ou `u_stime` du temps occupé par le processus `ps`, selon qu'il appartienne au mode utilisateur (`UMODE`) ou au mode noyau :

```
3788:          if((ps&UMODE) == UMODE) {
3789:                  u.u_utime++;
[...]
3792:          } else
3793:                  u.u_stime++;
```

Enfin, sous réserve que le processus `ps` ne soit pas prioritaire sur la suite des opérations du processus `clock` courant (lignes 3798-3799), l'heure du système, comptée en secondes, est mise à jour : si le compteur de tops, une fois incrémenté (`lbolt++`), égale ou dépasse la valeur `HZ`, c'est-à-dire le nombre de tops par seconde, c'est qu'au moins une seconde s'est écoulée. On peut donc retirer à ce compteur la valeur `HZ` de la période et ajouter 1 au compteur des secondes écoulées depuis le 1er janvier 1970 à minuit :

```
3797:          if(++lbolt >= HZ) {
3798:                  if((ps&0340) != 0)
3799:                          return;
3800:                  lbolt =- HZ;
3801:                  if(++time[1] == 0)
3802:                          ++time[0];
```

Ainsi l'« horloge » passive, celle qui donne l'heure, est-elle mise à jour à partir de l'horloge active, et non l'inverse. Il existe un Grand Horloger dont l'une des principales fonctions est de mouvoir à chaque seconde les aiguilles des horloges.

Ces tâches temporelles primordiales ayant été réalisées, ce qui reste n'est guère urgent. Il n'est exécuté qu'en priorité 1 :

```
3803:                  spl1();
```

Il est possible, en particulier, que le temps écoulé depuis le pénultième top d'horloge ait dépassé les 16.7 ou 20 millisecondes – selon que leur fréquence soit de 60 ou de 50 hertz – et que, par conséquent, les parties faiblement prioritaires de plusieurs exécutions successives du pilote d'horloge se bousculent au portillon. Mais ce cas de figure est sans gravité aucune pour le système, qui a la garantie absolue que les parties prioritaires sont exécutées en temps et en heure. Les autres le seront en leur temps, dans l'ordre d'arrivée, quand

le système en aura le loisir, n'occasionnant au pire que de menus désagréments – une baisse de réactivité de la console, le caractère provisoirement obsolète des informations temporelles des processus inactifs. Il en va de même lorsque nous rentrons de vacances ou sortons d'une période de travail particulièrement chargée.

C'est ainsi à un rythme de sénateur que sont réveillés d'abord les processus mis en sommeil pour un temps déterminé (lignes 3804-3805) puis ceux qui ne demandent d'être exécutés que lorsque les deux derniers bits du compteur de secondes valent zéro, autrement dit toutes les quatre secondes (lignes 3806-3809) :

```
3804:                   if(time[1]==tout[1] && time[0]==tout[0])
3805:                           wakeup(tout);
3806:                   if((time[1]&03) == 0) {
3807:                           runrun++;
3808:                           wakeup(&lbolt);
3809:                   }
```

C'est avec la même nonchalance que sont mises à jour les statistiques temporelles de tous les processus (`ptime`, `p_cpu`, `p_pri`), considérés un par un :

```
3810:                   for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
3811:                   if (pp->p_stat) {
3812:                           if(pp->p_time != 127)
3813:                                   pp->p_time++;
3814:                           if((pp->p_cpu & 0377) > SCHMAG)
3815:                                   pp->p_cpu =- SCHMAG; else
3816:                                   pp->p_cpu = 0;
3817:                           if(pp->p_pri > PUSER)
3818:                                   setpri(pp);
3819:                   }
```

Les tâches qu'effectue à un rythme périodique le pilote d'horloge sont donc de deux ordres, clairement distingués par leur niveau de priorité. Certaines sont des tâches urgentes, exécutées de manière quasi instantanée, car avec une très haute priorité et en début de fonction, ce qui rend peu probable toute interruption par un autre processus : telles sont la réinitialisation de l'horloge, la mise à jour de la console, l'exécution des fonctions prioritaires (typiquement celles du terminal), la mise à jour des données temporelles du processus en cours d'exécution (ou, plus précisément, qui était en cours d'exécution avant son interruption par le pilote d'horloge) et la mise à jour de l'heure du système. Les autres tâches sont celles dont la périodicité correspond à des échéances moins strictes : si le système est suroccupé, on peut les procrastiner sans grand danger.

Le type de temporalité dicté par les tops d'horloge pourrait paraître le simple décalque des machines synchrones de von Neumann : ne s'agit-il pas ici aussi d'une activité déclenchée périodiquement par des interruptions d'horloge ? Ce qu'il importe d'observer, par-delà l'important changement d'échelle physique – de la nanoseconde à la centiseconde –, est surtout le changement de nature des actes rythmés par cette nouvelle temporalité.

Il ne s'agit plus d'opérations élémentaires de la machine – processeur, mémoires, entrées et sorties –, mais de gestion des processus, autrement dit d'actes plus riches. Les processus sont en effet plus étendus dans le temps : contrairement aux opérations élémentaires de la machine, qui occupent un temps fini[18], celle d'un processus peut occuper un temps arbitrairement long. Les processus sont également plus étendus dans l'espace : ils possèdent non seulement des propriétés nouvelles, enregistrées dans les structures de données que sont les descripteurs, mais se voient aussi associer une pile de données en mémoire, autrement dit une portion d'espace. Aussi le temps du pilote d'horloge est-il le temps qui rythme une autre activité, propre au niveau du système d'exploitation et plus épaisse que celle de l'ontologie mécanique sous-jacente.

Comment caractériser ce temps ? C'est un temps qui, contrairement à celui du PDP-11, peut être administré finement ; un temps que l'on peut contrôler, exactement comme le cadastre des zones libres de la mémoire permet de contrôler l'espace. On peut ainsi reporter des actions à plus tard, simuler la simultanéité macroscopique de plusieurs processus en les alternant (pseudo-parallélisme), exécuter les actes en tenant compte de leur degré d'urgence. Ce temps est donc toujours la mesure de l'activité, mais de l'activité de haut niveau qu'est le processus, qui, si on le regarde de près, contient beaucoup d'actes et d'objets de bas niveau (des opérandes dans les registres de processeur, des données en mémoire, etc.). On voit ainsi, techniquement mise en œuvre, l'idée qu'il puisse exister plusieurs temporalités imbriquées les unes dans les autres et qu'à ces différents niveaux de temporalité correspondent des degrés différents d'activité.

---

18. Voir le manuel du PDP-11/40 : « Dans le cas le plus général, le temps d'exécution d'une instruction est la somme du temps d'adressage de la source, du temps d'adressage de la destination et du temps d'exécution et d'extraction (*fetch*) » (DEC 1972a, p. C-1). Toute l'annexe C de ce manuel contient d'ailleurs les estimations très précises du temps d'exécution de chaque instruction du PDP-11/40.

CONCLUSION

L'informatique offre à la philosophie pure l'occasion de prolonger la réflexion ouverte par l'autre grande science formelle que sont les mathématiques. Devant la Société française de philosophie, Jean Cavaillès disait en effet en 1939.

> « ne [pas chercher] à définir les Mathématiques, mais, au moyen des Mathématiques, à savoir ce que cela veut dire que connaître, penser ; c'est au fond, très modestement repris, le problème que posait Kant. La connaissance mathématique est centrale pour savoir ce qu'est la connaissance »
>
> (Cavaillès, 1946, p. 34).

Si, comme nous le croyons, l'informatique est la science – nécessairement formelle – de l'inscription des processus rationnels dans une matière en général, l'activité informatique devient aussi centrale pour savoir ce qu'est l'activité que la connaissance mathématique pour savoir ce qu'est la connaissance.

Convaincu, après Hourya Benis Sinaceur, que « l'analyse d'un texte scientifique est un exercice moins répandu, mais non moins instructif que celle d'un texte littéraire » (Sinaceur, 1991, p. 21), nous espérons avoir tiré des textes ici analysés deux convictions. La première est qu'il peut exister plusieurs niveaux d'activité et que des actes de niveau supérieur – ici les processus – peuvent intégrer, tout en les cachant, non seulement des actes, mais également des objets de niveau inférieur. La seconde conviction est qu'à ces différents niveaux d'activité correspondent autant d'échelles de temporalité : signe que, en informatique comme en métaphysique, le temps est aussi essentiel à l'activité en général que l'espace à l'objectivité.

RÉFÉRENCES

ARISTOTE (2014), *Physique*, in Pellegrin P. (dir.), *Aristote. Œuvres complètes*, Paris, Flammarion.

BACKUS J. (1978), « Can Programming be Liberated from the von Neumann Style? A functional style and its algebra of programs », *Communications of the ACM*, vol. 21, n° 8, août, pp. 613-641.

BOVET D., CESATI M. (2001), *Le Noyau Linux*, trad. J. Cornavin et E. Chaput, Paris, O'Reilly.

CAVAILLÈS J. (1946), « La pensée mathématique », in *Bulletin de la Société française de Philosophie*, XL (1946), compte rendu de la séance du 4 février 1939.

DEC (1972a), *PDP11/40 Processor Handbook*, Digital Equipment Corporation.

DEC (1972b), *KW11-P Programmable Real-Time Clock Manual*, Digital Equipment Corporation.

DEC (1973), *KW11-L Line Time Clock Manual*, Digital Equipment Corporation.

KERNIGHAN B., RITCHIE D. (2004), *Le Langage C. Norme ANSI*, Paris, Dunod.

KRAKOWIAK S. (2014), « Les débuts d'une approche scientifique des systèmes d'exploitation », *Interstices*, https://interstices.info/jcms/int_70839/les-debuts-dune-approche-scientifique-des-systemes-dexploitation.

LAZARD E., MOUNIER-KUHN P. (2016), *Histoire illustrée de l'informatique*, Paris, EDP Sciences.

LINUX (2000), noyau 2.2.14, http://www.kernel.org.

LINUX (2017), noyau 4.11.8, http://www.kernel.org.

LIONS J. (1996), *Lions Commentary on UNIX 6th Edition with Source Code*, Charlottesville, Peer-to-Peer Communications.

MÉLÈS B. (2013), « Unix selon l'ordre des raisons : la philosophie de la pratique informatique », *Philosophia Scientiæ*, vol. 17, n° 3, pp. 181-198.

MINIX (1997), in Tanenbaum A. et Woodhull A., *Operating Systems. Design and Implementation*, Upper Saddle River, Prentice Hall, annexe A, pp. 521-903.

VON NEUMANN J. (1945), « First Draft of a Report on the EDVAC », Contract No. W–670–ORD–4926 Between the United States Army Ordnance Department and the University of Pennsylvania, Moore School of Electrical Engineering, University of Pennsylvania, 30 juin 1945, section 2.

RITCHIE D., THOMPSON K. (1974), « The UNIX Time-Sharing System », *Communications of the ACM*, 17/7, juillet, pp. 365-375.

RITCHIE D. (1978), « The Unix Time-sharing System : A Retrospective », *Bell Labs Technical Journal*, vol. 57, n° 6, juillet-août, pp. 1947-1969.

RITCHIE D. (1984), « The Evolution of the Unix Time-sharing System », *AT&T Bell Laboratories Technical Journal*, vol. 63, n° 6, Part 2, octobre, pp. 1577-1593.

ROCHKIND M. (1987), *Unix : programmation avancée*, Paris, Masson.

SALUS P. (1994), *A Quarter Century of UNIX*, Reading: Addison Wesley.

SINACEUR H. (1991), *Corps et modèles. Essai sur l'histoire de l'algèbre réelle*, Paris, Vrin.

TANENBAUM A. (2003), *Systèmes d'exploitation*, 2e édition, Paris, Pearson.

UNIX (1975), in (Lions 1996), pp. 1-90.

TANENBAUM A., WOODHULL A. (1997), *Operating Systems. Design and Implementation*, Upper Saddle River, Prentice Hall.

WALDEN D., VAN VLECK T. (2011), *The Compatible Time Sharing System (1961-1973), Fiftieth Anniversary, Commemorative Overview*, Washinghton, IEEE Computer Society.