xv6

un système d'exploitation simple, proche d'Unix, pour l'enseignement

Russ Cox
Frans Kaashoek
Robert Morris

xv6-book@pdos.csail.mit.edu

Traduction française de la version du 4 septembre 2018 par
Timothée Zerbib
(encadré par Pierre David)

Disponible sur https://pdagog.gitlab.io/xv6-livret/

Table des matières

Préface et remerciements

Préface de la version originale

Ceci est un projet de texte destiné à un cours sur les systèmes d'exploitation. Il explique les principaux concepts des systèmes par l'étude d'un noyau exemple, nommé xv6. Xv6 est une ré-implémentation d'Unix Version 6 (v6) de Dennis Ritchie et Ken Thompson. Xv6 suit grossièrement la structure et le style de v6, mais est implémenté en C ANSI pour un multi-processeurs à base de x86.

Le texte devrait être lu avec le code source de xv6. Cette approche est inspirée par le livre « Commentary on UNIX 6th Edition » de John Lions (première édition, 14 juin 2000, paru chez Peer to Peer Communications, ISBN: 1-57398-013-7). Voir https://pdos.csail.mit.edu/6.828 pour des références à des ressources en ligne pour v6 et xv6, y compris plusieurs travaux pratiques sur xv6 à réaliser à la maison.

Nous avons utilisé ce texte dans 6.828, le cours de systèmes d'exploitation du MIT. Nous remercions les enseignants et les étudiants de 6.828 qui ont tous directement ou indirectement contribué à xv6. En particulier, nous souhaitons remercier Austin Clements et Nickolai Zeldovich. Enfin, nous souhaitons remercier les personnes qui nous a envoyé par e-mail des bugs dans le texte ou des suggestions de preuves : Abutalib Aghayev, Sebastian Boehm, Anton Burtsev, Raphael Carvalho, Rasit Eskicioglu, Color Fuzzy, Giuseppe, Tao Guo, Robert Hilderman, Wolfgang Keller, Austin Liew, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, miguelgvieira, Mark Morrissey, Harry Pan, Askar Safin, Salman Shah, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda et Zou Chang Wei.

Si vous remarquez des erreurs ou si vous avez des suggestions d'amélioration, veuillez envoyer un courriel à Frans Kaashoek et Robert Morris (kaashoek,rtm@csail.mit.edu).

Préface de la traduction française

Lorsque Timothée Zerbib m'a demandé un sujet de stage de licence sur les systèmes d'exploitation, en avril 2019, j'ai pensé à certains commentaires d'étudiants de master qui trouvaient difficile la lecture du livret xv6, notamment en raison de la langue. J'ai donc proposé à Timothée un stage visant à traduire ce livret. Ainsi est née cette traduction, que nous proposons sous licence libre à tous les étudiants francophones souhaitant approfondir les mécanismes de fonctionnement d'un système d'exploitation réel.

Pour réaliser cette traduction, nous avons fait certains choix par rapport à la version originale. Le premier est l'utilisation de LaTeX avec lequel nous avons plus de familiarité plutôt que `troff`. Le deuxième est le choix d'une numérotation des chapitres et des sections pour faciliter le repérage. Par ailleurs, la version originale contient une sorte d'index dans le texte, que nous avons choisi d'ignorer. Enfin, traduire des termes techniques pose toujours question :

nous avons choisi, lorsqu'il n'y avait pas de terme couramment admis en français, de conserver le terme d'origine en anglais. Les nombreuses notes du traducteur (abrégées par NdT) mentionnent ces choix de termes.

Cette traduction est disponible sur https://pdagog.gitlab.io/xv6-livret/. Nous encourageons les lecteurs à contribuer à l'étoffer, ou à signaler les erreurs ou les imprécisions via https://gitlab.com/pdagog/xv6-livret.

Chapitre 0

Interface du système d'exploitation

Le travail d'un système d'exploitation consiste à partager un ordinateur entre plusieurs programmes et à fournir un ensemble de services plus utile que celui fourni par le matériel seul. Le système d'exploitation gère et abstrait la couche matérielle de sorte que, par exemple, un logiciel de traitement de texte n'a pas besoin de se préoccuper du type de disque dur utilisé. Il partage également le matériel entre plusieurs programmes afin qu'ils s'exécutent (ou semblent s'exécuter) en même temps. Enfin, les systèmes d'exploitation fournissent aux programmes des moyens contrôlés d'interagir ensemble, afin qu'ils puissent partager des données ou travailler de concert.

Un système d'exploitation fournit des services aux programmes utilisateurs via une interface. Concevoir une bonne interface s'avère être difficile. D'une part, nous aimerions que l'interface soit simple et minimale, car cela facilite la rédaction d'une implémentation correcte. D'autre part, nous pourrions être tentés d'offrir de nombreuses fonctionnalités sophistiquées aux applications. Une bonne façon de résoudre ce dilemme est de concevoir des interfaces qui reposent sur un nombre restreint de mécanismes qui peuvent être combinés pour en offrir de plus généraux.

Ce livret utilise un système d'exploitation comme exemple concret pour illustrer les concepts des systèmes d'exploitation. Ce système, nommé xv6, fournit les interfaces de base introduites par le système Unix de Ken Thompson et Dennis Ritchie, tout en imitant son design interne. Unix fournit une interface minimale dont les mécanismes se combinent bien, offrant un surprenant degré de généralité. Cette interface a eu tellement de succès que les systèmes d'exploitation modernes — BSD, Linux, Mac OS X, Solaris, et même, dans une moindre mesure, Microsoft Windows — ont une interface de ce type. Comprendre xv6 est un bon moyen de comprendre n'importe lequel de ces systèmes et bien d'autres.

Comme le montre la figure 1, xv6 prend la forme traditionnelle d'un « noyau », un programme particulier qui fournit des services aux autres programmes en cours d'exécution. Chaque programme en cours d'exécution, appelé « processus », possède son espace mémoire contenant des instructions, des données et une pile. Les instructions représentent les calculs à effectuer. Les données sont les variables sur lesquelles les calculs vont être effectués. La pile organise les appels de procédures du programme.

Lorsqu'un processus doit appeler un service du noyau, il demande un appel de procédure à l'interface du système d'exploitation. Une telle procédure est appelée un « appel système [1] ». L'appel système arrive au noyau ; le noyau effectue le service et retourne. Ainsi, un processus alterne son exécution entre « l'espace utilisateur » et « l'espace noyau ».

1. NdT : Nous utiliserons les termes « appel système » et « primitive système » (ou simplement primitive) indépendamment.

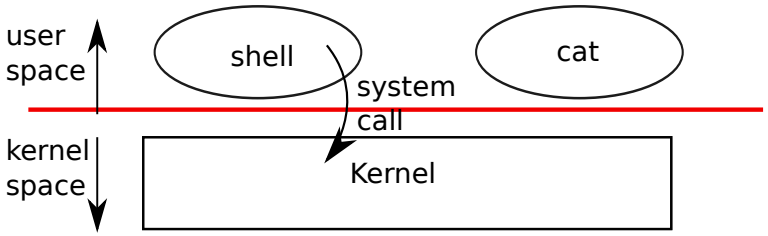

FIGURE 1 – Un noyau et deux processus utilisateurs.

Le noyau utilise les mécanismes matériels de protection du processeur afin de s'assurer que chaque processus s'exécutant dans l'espace utilisateur ne peut accéder qu'à son propre espace mémoire. Le noyau s'exécute avec les privilèges matériels requis pour implémenter ces protections tandis que les programmes utilisateurs s'exécutent sans ces privilèges. Quand un programme utilisateur fait un appel système, le matériel élève les privilèges et exécute une fonction pré-définie dans le noyau.

La liste des appels système qu'un noyau fournit correspond à l'interface que voient les programmes utilisateurs. Le noyau xv6 fournit une partie des services et appels système qu'offrent traditionnellement les noyaux Unix. La figure 2 liste tous les appels système de xv6.

La suite de ce chapitre décrit brièvement les services fournis par xv6 — processus, mémoire, descripteurs de fichier, tubes et système de fichiers — et les illustre avec des extraits de code et des discussions sur la façon dont le *shell*, qui est l'interface utilisateur principale dans les systèmes de type Unix traditionnels, les utilise. L'utililisation des appels système par le shell illustre le soin avec lequel ils ont été conçus.

Le shell est un programme ordinaire qui lit des commandes entrées par l'utilisateur et les exécute. Le fait que le shell soit un programme utilisateur et non une partie du noyau illustre la puissance de l'interface des appels système : le shell est un programme qui n'a rien de particulier. Cela signifie également que le shell est facilement remplaçable ; par conséquent, les systèmes Unix modernes possèdent une large variété de shells, chacun ayant sa propre interface utilisateur et ses propres fonctionnalités. Le shell de xv6 est une implémentation simplifiée de ce qui fait le cœur du shell de Bourne d'Unix [2]. Son implémentation peut être trouvée à la ligne 8550.

0.1 Processus et mémoire

Un processus xv6 est constitué d'un espace mémoire utilisateur (instructions, données et pile), et d'un état par processus conservé par le noyau [3]. Xv6 peut partager le temps (*time sharing*) entre les processus : il commute de manière transparente les processus en attente d'exécution sur les processeurs disponibles. Lorsqu'un processus n'est pas en cours d'exécution, xv6 sauvegarde ses registres processeurs afin de les restaurer lors de sa prochaine exécution. Le noyau associe à chaque processus un identifiant, ou `pid`.

Un processus peut créer un autre processus en utilisant l'appel système `fork`. `Fork` crée un nouveau processus, appelé « processus enfant », avec exactement le même contenu mémoire que le processus original, appelé « processus parent ». `Fork` retourne à la fois dans le parent et

2. NdT : Le shell de Bourne est le shell par défaut depuis la version 7 d'Unix et a servi de base pour la norme POSIX. Son fichier exécutable se trouve à l'emplacement `/bin/sh`.

3. NdT : Pour chaque processus, le noyau sauvegarde ses attributs (PID, PPID, UID, GID...) ainsi que les registres processeurs permettant de restaurer son contexte (cf. Chapitre **??**).

Appel système	Description
`fork()`	Crée un processus
`exit()`	Termine le processus courant
`wait()`	Attend qu'un processus enfant se termine
`kill(pid)`	Termine le processus `pid`
`getpid()`	Retourne le pid du processus courant
`sleep(n)`	Met le processus en attente pendant `n` tops d'horloge
`exec(filename, *argv)`	Charge un fichier et l'exécute
`sbrk(n)`	Augmente l'espace mémoire du processus de `n` octets
`open(filename, flags)`	Ouvre un fichier ; `flags` indique le mode (lecure/écriture)
`read(fd, buf, n)`	Stocke `n` octets depuis un fichier ouvert dans `buf`
`write(fd, buf, n)`	Écrit `n` octets dans un fichier ouvert
`close(fd)`	Libère le descripteur de fichier `fd`
`dup(fd)`	Duplique le descripteur de fichier `fd`
`pipe(p)`	Crée un tube et retourne les descripteurs de fichier dans `p`
`chdir(dirname)`	Change le répertoire courant
`mkdir(dirname)`	Crée un nouveau répertoire
`mknod(name, major, minor)`	Crée un fichier spécial de type périphérique
`fstat(fd)`	Renvoie les informations sur un fichier ouvert
`link(f1, f2)`	Crée un nouveau nom (`f2`) pour le fichier `f1`
`unlink(filename)`	Supprime un fichier

FIGURE 2 – Liste des appels système de xv6.

dans l'enfant. Dans le parent, `fork` renvoie le pid du processus enfant ; dans l'enfant, il renvoie zéro. Par exemple, considérons le fragment de programme suivant :

```
int pid = fork();
if(pid > 0){
  printf("parent: child=%d\n", pid);
  pid = wait();
  printf("child %d is done\n", pid);
} else if(pid == 0){
  printf("child: exiting\n");
  exit();
} else {
  printf("fork error\n");
}
```

L'appel système `exit` provoque l'arrêt du processus appelant et la libération des ressources comme la mémoire et les fichiers ouverts. L'appel système `wait` renvoie le pid d'un enfant du processus courant terminé ; si aucun des enfants de l'appelant n'est terminé, `wait` attend qu'un d'eux se termine. Dans l'exemple, les sorties :

```
parent: child: 1234
child: exiting
```

peuvent être affichées dans cet ordre ou son inverse, dépendant de si c'est le parent ou l'enfant qui appelle le premier sa fonction `printf`. Après que l'enfant ait terminé, le `wait` du parent retourne, provoquant l'affichage par le `printf` du parent :

```
parent: child 1234 is done
```

Bien qu'initialement, l'enfant possède le même contenu mémoire que le parent, le parent et l'enfant s'exécutent avec des espaces mémoires et des registres différents : changer une variable dans l'un n'affecte pas l'autre. Par exemple, lorsque la valeur de retour de `wait` est stockée dans la variable `pid` dans le processus parent, cela ne change pas la valeur de la variable `pid` dans l'enfant. La valeur de `pid` dans l'enfant sera toujours zéro.

L'appel système `exec` remplace l'espace mémoire du processus appelant par un nouvel espace mémoire chargé depuis l'image d'un fichier stocké dans le système de fichiers. Ce fichier doit avoir un format particulier, qui spécifie quelle partie du fichier comporte les instructions, quelle partie comporte les données, à quelle instruction commencer, etc. Xv6 utilise le format ELF, que le chapitre 2 détaille davantage. Lorsque `exec` termine avec succès, il ne retourne pas au programme appelant ; à la place, les instructions chargées depuis le fichier commencent leur exécution à partir du point d'entrée déclaré dans l'en-tête ELF. `Exec` prend deux arguments : le nom du fichier contenant l'exécutable et un tableau d'arguments sous forme de chaînes de caractères. Par exemple :

```
char *argv[3];

argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec␣error\n");
```

Ce fragment remplace le programme appelant par une instance du programme `/bin/echo` s'exécutant avec la liste d'arguments `echo hello`. La plupart des programmes ignorent le premier argument, qui est par convention le nom du programme.

Le shell de xv6 utilise les appels ci-dessus pour exécuter des programmes pour le compte des utilisateurs. La structure principale du shell est simple (cf. `main` [8701]). La boucle principale lit une ligne entrée par l'utilisateur avec `getcmd`. Puis, elle appelle `fork`, qui crée une copie du processus du shell. Le parent appelle `wait`, tandis que l'enfant exécute la commande. Par exemple, si l'utilisateur avait tapé `"echo hello"`, `runcmd` aurait été appelé avec `"echo hello"` pour argument. `runcmd` [8606] exécute la commande réelle. Pour `"echo hello"` il appellerait `exec` [8626]. Si `exec` réussit, alors l'enfant exécutera les instructions de `echo` au lieu de `runcmd`. À un moment, `echo` va appeler `exit`, ce qui provoquera la terminaison de la fonction `wait` du parent dans la fonction `main` [8701]. Vous pourriez vous demander pourquoi `fork` et `exec` ne sont pas combinés en un appel unique ; nous allons voir plus loin qu'il est plus astucieux de séparer les appels pour la création d'un processus et le chargement d'un programme.

Xv6 alloue la majorité de son espace mémoire utilisateur de façon implicite : `fork` alloue la mémoire requise pour permettre la copie de la mémoire du processus parent, et `exec` alloue suffisamment de mémoire pour contenir le fichier exécutable. Un processus qui nécessite plus de mémoire durant son exécution (peut-être à cause d'un `malloc`) peut appeler `sbrk(n)` pour augmenter de n octets la partie de la mémoire consacrée aux données ; `sbrk` renvoie l'adresse de la mémoire nouvellement allouée.

Xv6 ne fournit pas de notion d'utilisateur ni de protection entre utilisateurs ; en termes Unix, tous les processus xv6 s'exécutent en tant que superutilisateur (*root*).

0.2 I/O et Descripteurs de fichier

Un « descripteur de fichier » est un petit entier représentant un objet géré par le noyau duquel un processus peut lire ou sur lequel il peut écrire. Un processus peut obtenir un descripteur de fichier en ouvrant un fichier, un répertoire ou un périphérique, ou bien en créant un tube, ou en dupliquant un autre descripteur de fichier. Pour simplifier, nous allons souvent désigner l'objet auquel se réfère un descripteur de fichier par le mot fichier ; l'interface de descripteur de fichier abstrait les différences entre fichiers, tubes et périphériques, les faisant tous paraître comme des flux d'octets.

En interne, le noyau xv6 utilise le descripteur de fichier comme un index dans une table propre à chaque processus, de sorte que chacun ait un espace privé de descripteurs de fichier commençant à zéro. Par convention, un processus lit depuis le descripteur de fichier 0 (entrée standard), écrit ses sorties sur le descripteur de fichier 1 (sortie standard) et écrit ses messages d'erreur sur le descripteur de fichier 2 (sortie d'erreur). Comme nous le verrons, le shell exploite cette convention pour implémenter les redirections des entrées/sorties et les pipelines. Le shell s'assure de toujours posséder au moins trois descripteurs de fichier ouvert [8707], qui sont par défaut les descripteurs de fichier de la console.

Les appels système `read` et `write` lisent et écrivent des octets sur les fichiers ouverts désignés par les descripteurs de fichier. L'appel `read(fd, buf, n)` lit au plus `n` octets depuis le descripteur `fd`, les copie dans `buf` et renvoie le nombre d'octets lus. Chaque descripteur de fichier qui fait référence à un fichier lui associe un *offset* [4]. `Read` lit des données à partir de l'offset associé au fichier puis augmente cet offset du nombre d'octets lus : un `read` ultérieur va retourner les octets suivant ceux retournés par le premier `read`. Lorsqu'il n'y a plus d'octet à lire, `read` renvoie zéro pour signaler la fin du fichier.

L'appel `write(fd, buf, n)` écrit `n` octets depuis `buf` sur le descripteur de fichier `fd` et renvoie le nombre d'octets écrits. Ce nombre n'est inférieur à `n` que lorsqu'il se produit une erreur. Tout comme `read`, `write` écrit des données à l'offset courant du fichier puis augmente cet offset du nombre d'octets écrits : chaque `write` reprend là où le précédent s'était arrêté.

Le fragment de programme suivant (formant le cœur de `cat`) copie des données depuis l'entrée standard sur la sortie standard. Si une erreur survient, il écrit un message sur la sortie d'erreur.

```
char buf[512];
int n;

for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read␣error\n");
    exit();
  }
  if(write(1, buf, n) != n){
    fprintf(2, "write␣error\n");
    exit();
  }
}
```

Ce qu'il faut retenir de ce morceau de code est que `cat` ne sait pas s'il est en train de lire un fichier, la console ou un tube. De même, `cat` ne sait pas s'il écrit sur la console, un fichier ou n'importe quoi d'autre. L'utilisation de descripteurs de fichier et la convention que le descripteur de fichier 0 correspond à l'entrée et le descripteur 1 à la sortie conduisent à une implémentation simple de `cat`.

L'appel système `close` libère un descripteur de fichier, le rendant réutilisable par un futur appel à `open`, `pipe` ou `dup` (voir ci-après). Un descripteur de fichier nouvellement alloué correspond toujours au plus petit descripteur non utilisé par le processus courant.

Les descripteurs de fichier couplés à `fork` simplifient l'implémentation des redirections d'entrées/sorties. `Fork` copie la table des descripteurs de fichier du parent dans sa mémoire, de

4. NdT : Le terme *offset* correspond à « position relative » (par rapport à une référence). Une traduction pourrait être « décalage » mais nous avons préféré conserver le terme anglais, faute de terme communément admis. Dans ce contexte, un *offset* est donc le nombre d'octets depuis le début du fichier.

sorte que l'enfant démarre avec exactement les mêmes fichiers ouverts que le parent. L'appel système `exec` remplace le processus appelant en mémoire, mais conserve sa table des fichiers ouverts. Ce comportement permet au shell d'implémenter les redirections d'entrées/sorties en appelant `fork`, en réouvrant le descripteur de fichier choisi puis en exécutant le nouveau programme. Voici une version simplifiée du code exécuté par un shell pour la commande `cat < input.txt`

```
char *argv[2];

argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
  close(0);
  open("input.txt", O_RDONLY);
  exec("cat", argv);
}
```

Après la fermeture par l'enfant du descripteur 0, il est garanti qu'`open` utilise ce descripteur pour le fichier `input.txt` nouvellement ouvert : 0 sera le plus petit descripteur de fichier disponible. `Cat` s'exécutera alors avec son descripteur de fichier 0 (entrée standard) faisant référence à `input.txt`.

Le code pour les redirections d'entrées/sorties dans le shell de xv6 fonctionne exactement de cette façon [8630]. Rappelez-vous qu'à ce moment dans le code, le shell a déjà forké le shell enfant et que `runcmd` appellera `exec` pour charger le nouveau programme. La raison de la séparation de `fork` et `exec` devrait maintenant être claire, car s'ils sont séparés, le shell peut créer un enfant avec `fork`, utiliser `open`, `close` et `dup` dans l'enfant pour changer les descripteurs de fichier de l'entrée et de la sortie standard, avant de lancer `exec`. Aucun changement dans le programme appelé par `exec` (`cat` dans notre exemple) n'est requis. Si `fork` et `exec` étaient combinés en un seul appel système, un autre schéma (probablement plus complexe) aurait été nécessaire pour pemettre au shell de rediriger l'entrée et la sortie standard, ou alors le programme lui-même devrait comprendre comment rediriger l'entrée/sortie.

Bien que `fork` copie la table des descripteurs de fichier, les offsets sont partagés entre parent et enfant. Considérons l'exemple :

```
if(fork() == 0) {
  write(1, "hello␣", 6);
  exit();
} else {
  wait();
  write(1, "world\n", 6);
}
```

À la fin de ce morceau de code, le fichier attaché au descripteur 1 contiendra `hello world`. Le `write` du parent (qui, grâce à `wait`, ne s'exécute qu'après la terminaison de l'enfant) reprend là où le `write` de l'enfant s'était arrêté. Ce comportement permet de produire une sortie séquentielle à partir de séquences de commandes shell, comme `(echo hello; echo world) > output.txt`.

L'appel système `dup` duplique un descripteur de fichier existant, et renvoie un nouveau descripteur faisant référence au même objet d'entrée/sortie sous-jacent. Les deux descripteurs de fichier partagent le même offset, comme le font les descripteurs de fichier dupliqués par `fork`. Voici une autre façon d'écrire `hello world` dans un fichier :

```
fd = dup(1);
write(1, "hello␣", 6);
write(fd, "world\n", 6);
```

Deux descripteurs de fichier partagent leur offset s'ils dérivent du même descripteur par une séquence d'appels à `fork` et `dup`. Hormis cela, les descripteurs de fichier ne partagent pas leur offset, même s'ils résultent d'appels à `open` sur le même fichier. `Dup` permet aux shells d'implémenter des commandes comme : `ls fichier_existant fichier_inexistant > tmp1 2>&1`. Le « `2>&1` » indique au shell de transmettre à la commande un descripteur de fichier 2 qui est une copie (via `dup`) du descripteur 1. Le nom du fichier existant et le message d'erreur pour le fichier non existant vont tous deux être écrits dans le fichier `tmp1`. Le shell de xv6 ne supporte pas les redirections d'entrées/sorties pour le descripteur correspondant à la sortie d'erreur, mais vous savez maintenant comment l'implémenter.

Les descripteurs de fichier constituent une puissante abstraction car ils masquent les détails de ce à quoi ils sont connectés : un processus écrivant sur le descripteur de fichier 1 peut écrire sur un fichier, un périphérique tel que la console ou un tube.

0.3 Tubes

Un *tube* est un petit espace noyau se présentant aux processus comme une paire de descripteurs de fichier, un pour la lecture et l'autre pour l'écriture. Écrire des données à un bout du tube rend ces données accessibles à la lecture à l'autre bout du tube. Les tubes fournissent un moyen aux processus de communiquer.

L'exemple de code suivant exécute le programme `wc` avec l'entrée standard connectée au côté lecture du tube.

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

pipe(p);
if(fork() == 0) {
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {
  close(p[0]);
  write(p[1], "hello␣world\n", 12);
  close(p[1]);
}
```

Le programme appelle `pipe`, qui crée un nouveau tube et sauvegarde les descripteurs de fichier pour lecture et écriture dans le tableau `p`. Après `fork`, le parent comme l'enfant auront leurs descripteurs de fichier faisant référence au tube. Le processus enfant duplique l'extrémité lecture sur son descripteur 0, ferme les descripteurs de fichier de `p` et exécute `wc`. Lorsque `wc` lit depuis son entrée standard, il lit depuis le tube. Le processus parent ferme l'extrémité lecture du tube, écrit sur le tube puis ferme l'extrémité écriture du tube.

Si aucune donnée n'est disponible, un appel à `read` sur un tube résulte en l'attente qu'une donnée soit écrite ou que tous les descripteurs de fichier faisant référence à l'extrémité écriture du tube soient fermés ; dans ce dernier cas, `read` renvoie 0, exactement comme s'il avait atteint la fin d'un fichier de données. Le fait que `read` soit bloquant {jusqu'à ce qu'il soit impossible que de nouvelles données arrivent / tant que de nouvelles données peuvent arriver} est une

des raisons pour lesquelles il est important que le processus enfant ferme l'extrémité écriture du tube avant d'exécuter `wc` : si l'un des descripteurs de fichier de `wc` faisait référence à l'extrémité écriture du tube, `wc` ne percevrait jamais la fin du fichier.

Le shell de xv6 implémente les pipelines tels que `grep fork sh.c | wc -l` de manière similaire au code ci-dessus 8650. Le processus enfant crée un tube pour connecter la partie gauche du pipeline à sa partie droite. Puis il appelle `fork` et `runcmd` pour la partie gauche du pipeline et `fork` et `runcmd` pour la partie droite, et attend que les deux se terminent. La partie droite du pipeline peut très bien être une commande qui contient elle-même un tube (par exemple `a | b | c`), qui lui-même fork deux enfants (un pour `b` et l'autre pour `c`). Ainsi, le shell peut créer un arbre de processus. Les feuilles de cet arbre sont les commandes et les noeuds internes les processus qui attendent que les enfants gauche et droit retournent. En principe, on pourrait faire en sorte que les nœuds intérieurs exécutent l'extrémité gauche du pipeline, mais le réaliser correctement risquerait de compliquer la mise en œuvre.

Les tubes peuvent ne pas sembler plus puissants que les fichiers temporaires : le pipeline `hello world | wc` peut être implémenté sans tubes de la manière suivante `echo hello world > /tmp/xyz; wc < /tmp/xyz` Les tubes ont au minimum quatre avantages sur les fichiers temporaires dans cette situation. D'abord, les tubes se suppriment automatiquement ; avec une redirection passant par des fichiers, le shell devrait faire attention lors de la suppression de `/tmp/xyz` une fois ses tâches terminées. Ensuite, les tubes peuvent échanger des flux de données arbitrairement grands alors que la redirection par fichier nécessite une place suffisante sur le disque pour stocker toutes les données. De plus, les tubes permettent une exécution parallèle des différentes étapes du pipeline, alors que l'approche par fichier requiert la terminaison du premier programme avant de démarrer le second. Enfin, si vous implémentez une communication entre processus, le bloquage des tubes en lecture/écriture est plus efficace que la sémantique non bloquante des fichiers.

0.4 Système de fichiers

Le système de fichiers de xv6 fournit des fichiers de données, représentés par une suite linéaire d'octets, et des répertoires, qui contiennent des références nommées à des fichiers de données et à d'autres répertoires. Les répertoires forment un arbre commençant à un répertoire particulier appelé la « racine ». Un « chemin » tel que `/a/b/c` fait référence au fichier ou répertoire nommé `c` situé à l'intérieur du répertoire nommé `b`, lui-même situé à l'intérieur du répertoire nommé `a`, lui-même situé dans le répertoire racine `/`. Les chemins qui ne commencent pas par `/` sont évalués relativement au « répertoire courant » du processus appelant, qui peut être modifié à l'aide de l'appel système `chdir`. Les deux fragments de code suivants ouvrent le même fichier (en supposant que tous les répertoires impliqués existent) :

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);


open("/a/b/c", O_RDONLY);
```

La première partie du code change le répertoire courant du processus pour `/a/b` ; la seconde partie ne fait pas référence au répertoire courant du processus, pas plus qu'elle ne le modifie.

Il existe plusieurs appels système qui créent un nouveau fichier ou répertoire : `mkdir` qui crée un nouveau répertoire, `open` avec l'option `O_CREATE` qui crée un nouveau fichier de données et `mknod` qui crée un nouveau fichier de type périphérique. L'exemple suivant les illustre tous les trois :

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

`Mknod` crée un fichier dans le système de fichiers, mais ce fichier est vide. Les métadonnées de ce fichier le définissent comme un fichier spécial de type périphérique et conservent ses numéros de périphérique majeur et mineur [5] (représentés par les deux arguments de `mknod`) qui identifient de manière unique un périphérique du noyau. Lorsqu'un processus ouvrira ce fichier, le noyau transférera les appels système `read` et `write` à l'implémentation du noyau du périphérique au lieu de le faire parvenir au système de fichiers.

`Fstat` récupère les informations à propos de l'objet auquel se réfère un descripteur de fichier. Cette fonction remplit une structure de données `struct stat` définie dans `stat.h` par :

```
#define T_DIR  1    // Répertoire
#define T_FILE 2    // Fichier
#define T_DEV  3    // Périphérique

struct stat {
  short type;  // Type de fichier
  int dev;     // Numéro de périphérique du disque contenant le système de fichiers
  uint ino;    // Numéro inode
  short nlink; // Nombre de références vers ce fichier
  uint size;   // Taille du fichier en octets
};
```

Il faut distinguer le nom des fichiers des fichiers eux-mêmes ; le même fichier sous-jacent, appelé « inode [6] », peut avoir plusieurs noms, appelés « liens ». L'appel système `link` crée un autre nom dans le système de fichiers, faisant référence au même inode qu'un fichier préexistant. Ce morceau de code crée un nouveau fichier nommé à la fois `a` et `b` :

```
open("a", O_CREATE|O_WRONLY);
link("a", "b");
```

Lire ou écrire sur a revient à lire ou écrire sur b. Chaque inode est référencé par un unique « numéro d'inode ». Après l'exécution du morceau de code ci-dessus, il est possible de déterminer que `a` et `b` font référence au même fichier en utilisant `fstat` : tous deux vont avoir le même numéro d'inode (`ino`), et le compteur de références du fichier (`nlink`) aura pour valeur 2.

L'appel système `unlink` supprime un nom du système de fichiers. Le numéro d'inode et l'espace mémoire contenant les données du fichier ne seront libérés que lorsque le compteur de référence vaudra 0 et qu'aucun descripteur de fichier n'y fera plus référence. Ainsi, ajouter `unlink("a")` au bloc de code précédent laisse accessible l'inode et le contenu du fichier via `b`. De plus,

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR);
unlink("/tmp/xyz");
```

est une manière idiomatique de créer un inode temporaire qui sera supprimé lorsque le processus fermera le descripteur de fichier `fd` ou lorsqu'il se terminera.

Les commandes shell pour utiliser les opérations du système de fichiers sont implémentées en tant que programmes utilisateurs comme `mkdir`, `ln`, `rm`, etc. Une telle conception permet à

5. NdT : Le numéro majeur correspond au numéro du pilote. Le numéro mineur correspond au numéro du périphérique géré par ce pilote.

6. NdT : Le mot inode vient de la contraction des mots index et node, c'est pourquoi il sera par la suite considéré comme étant masculin.

n'importe qui d'étendre le shell avec de nouvelles commandes utilisateurs en ajoutant simplement un nouveau programme en mode non privilégié. Avec le recul, cette façon de faire semble évidente, mais d'autres systèmes conçus à la même époque qu'Unix incorporaient souvent de telles commandes dans le shell (et ont incorporé le shell dans le noyau).

`Cd` est une exception puisqu'il est lié au shell [8716]. `Cd` doit changer le répertoire de travail courant du shell lui-même. Si `cd` était exécuté comme une commande ordinaire, le shell aurait forké un processus fils, ce fils aurait exécuté `cd` et `cd` aurait changé le répertoire de travail courant du **fils**. Le répertoire de travail courant du parent (c.à.d le répertoire de travail courant du shell) n'aurait pas changé.

0.5 Dans la réalité

L'association par Unix des descripteurs de fichiers « standards », des tuyaux ainsi que de la syntaxe pratique du shell pour exécuter des opérations sur ces derniers a été une avancée majeure dans la rédaction de programmes réutilisables dans un cadre général. L'idée a suscité toute une culture d'« outils logiciel », en grande partie responsable de la puissance et de la popularité d'Unix, et le shell était le premier soi-disant « langage de script ». L'interface des appels système Unix persiste aujourd'hui dans des systèmes tels que BSD, Linux et Mac OS X.

L'interface des appels système Unix a été standardisée à travers le standard POSIX (*Portable Operating System Interface*). Xv6 n'est **pas** conforme à POSIX. Il n'implémente pas certains appels système (dont certains très basiques comme `lseek`), il n'implémente des appels système que partiellement, etc. Nos principaux objectifs pour xv6 sont la clarté et la simplicité en fournissant une interface des appels système semblable à celle d'Unix. De nombreuses personnes ont étendu xv6 en ajoutant quelques appels système basiques et une bibliothèque C simple de sorte qu'il puisse exécuter des programmes Unix basiques. Les noyaux modernes fournissent toutefois de nombreux appels système supplémentaires, et beaucoup plus de types de services qu'xv6. Par exemple, ils supportent la mise en réseau, un système de fenêtrage, des *threads*[7] au niveau utilisateur, des pilotes pour de nombreux périphériques... Les noyaux modernes évoluent continuellement et rapidement, et offrent de nombreuses fonctionnalités en plus de POSIX.

Les systèmes d'exploitation modernes dérivant d'Unix n'ont pour la plupart pas suivi le modèle Unix de représentation des périphériques en tant que fichiers spéciaux, comme la `console` dont nous avons discuté plus tôt. Les auteurs d'Unix sont allés plus loin en créant Plan 9[8], qui applique le concept du « tout fichier » aux systèmes modernes, représentant les réseaux, les graphiques et les autres ressources comme des fichiers ou des arborescences de fichiers.

L'abstraction du système de fichiers a été une puissante idée. Cependant, il existe d'autres modèles pour les interfaces des systèmes d'exploitation. Multics, un prédecesseur d'Unix, abstrait le stockage des fichiers de façon à le faire ressembler à de la mémoire, produisant une interface très différente. La complexité du design de Multics a eu une influence directe sur les concepteurs d'Unix, qui ont tenté de créer quelque chose de plus simple.

Ce livret étudie comment xv6 implémente son interface de type Unix, mais ces idées et concepts s'appliquent bien au delà d'Unix. Tous les systèmes d'exploitation doivent multiplexer les processus sur le matériel {sous-jacent/impliqué}, isoler les processus les uns des autres et fournir des mécanismes permettant une communication entre processus. Après avoir étudié xv6, vous

7. NdT : Il n'existe pas vraiment de traduction française du mot *thread*. Bien que certains lui préfèrent « fil d'exécution », nous conserverons dans ce document le mot d'origine.

8. NdT : Plan 9 est un système d'exploitation créé pour régler certains problèmes d'Unix jugés trop profonds pour être corrigés. Plan 9 hérite de notions Unix et en améliore certaines comme l'adage du « tout fichier ».

devriez être capable d'examiner d'autres systèmes d'exploitation plus complexes et d'identifier les concepts de xv6 également présents dans ces systèmes.

Chapitre 1

Organisation du système d'exploitation

Une fonctionnalité obligatoire pour un système d'exploitation est de pouvoir supporter plusieurs activités simultanées. Par exemple, en utilisant l'interface des appels système décrite dans le chapitre 0, un processus peut créer des nouveaux processus avec `fork`. Le système d'exploitation doit partager les ressources de l'ordinateur dans le temps entre ces différents processus. Par exemple, même s'il y a plus de processus qu'il n'y a de processeurs, le système d'exploitation doit s'assurer que chacun des processus progressent. Le système d'exploitation doit aussi s'arranger pour créer une *isolation* entre les processus. À savoir, si un processus a un bogue et plante [1], celà ne devrait pas affecter les processus qui ne dépendent pas du processus qui s'est arrêté. L'isolation complète est cependant trop stricte, puisqu'il devrait être possible pour les processus d'interagir ; les pipelines en sont un exemple. Ainsi, un système d'exploitation doit remplir trois exigences : le multiplexage, l'isolation et l'interaction.

Ce chapitre fournit une vue d'ensemble de comment l'organisation des systèmes d'exploitation permet de remplir ces trois exigences. Il s'avère qu'il existe plusieurs façons de le faire, mais ce texte se concentre sur les conceptions traditionnelles centrées autour d'un *noyau monolithique*, qui est utilisé dans beaucoup de systèmes d'exploitation Unix. Ce chapitre introduit la conception de xv6 en suivant la création de son premier processus, lorsque xv6 commence à s'exécuter. En faisant ainsi, ce texte fournit un aperçu de l'implémentation de toutes les abstractions majeures que fournit xv6, comment elles interagissent et comment les trois exigences que sont le multiplexage, l'isolation et l'interaction sont satisfaites. Xv6 évite au maximum de faire du premier processus un cas particulier. Au lieu de cela, il réutilise du code qu'il fournit pour des opérations standards. Les chapitres subséquents vont présenter chaque abstraction plus en détail.

Xv6 tourne sur les processeurs Intel 80386 ou versions x86 supérieures, sur plateforme PC, et beaucoup de ses fonctionnalités bas-niveau (par exemple, son implémentation des processus) sont spécifiques à x86. Ce livre suppose que le lecteur ait fait un peu de programmation au niveau machine et va introduire les idées propres à x86 au fur et à mesure. L'annexe A résume brièvement la plateforme PC.

1.1 Abstraire les ressources physiques

La première question que l'on pourrait se poser face à un système d'exploitation est sa nécessité. Le fait est que l'on pourrait implémenter les appels système de la figure 2 (page 11)

1. NdT : Bien que familier, nous utiliserons le terme « planter » pour traduire l'idée originale qu'un programme *fail*, c'est-à-dire cesse de fonctionner de façon inopinée.

comme une bibliothèque, à laquelle se lient les applications. Avec cette technique, chaque application pourrait avoir sa propre bibliothèque adaptée à ses besoins. Les applications pourraient interagir directement avec les ressources matérielles et utiliser ces ressources au mieux pour l'application (par exemple, pour atteindre des performances élevées ou prédictibles). Certains systèmes embarqués ou temps réel sont organisés de cette façon.

L'inconvénient de cette approche « bibliothèque » est que, si plusieurs applications sont en cours d'exécution, elles doivent se comporter correctement. Par exemple, chaque application doit périodiquement se retirer du processeur pour qu'une autre application puisse s'exécuter. Un tel schéma *coopératif* pourrait convenir si toutes les applications se faisaient confiance et n'avaient pas de bogue. Il est plus courant que les applications ne se fassent pas confiance et aient des bogues, c'est pourquoi on désire souvent une isolation plus stricte que celle fournie par un schéma coopératif.

Pour parvenir à une isolation stricte, il est utile d'interdire aux applications l'accès direct aux ressources matérielles sensibles, et à la place, d'abstraire les ressources en services. Par exemple, les applications n'interagissent avec un système de fichiers qu'au travers des appels système `open`, `read`, `write` et `close`, au lieu de lire et d'écrire des secteurs bruts de disque. Cela fournit aux applications la commodité des chemins d'accès et permet au système d'exploitation (en tant qu'implémenteur de l'interface) de gérer le disque.

De même, Unix commute de manière transparente les processeurs matériels entre les processus, sauvegardant et restaurant l'état des registres si nécessaire, de sorte que les applications n'aient pas à se préoccuper du partage du temps. Cela permet implicitement au système d'exploitation de partager les processeurs même si des applications sont dans une boucle infinie.

Comme autre exemple, les processus Unix utilisent `exec` pour construire leur image mémoire, au lieu d'interagir directement avec la mémoire physique. Ceci permet au système d'exploitation de décider où placer un processus en mémoire ; si la quantité de mémoire est faible, le système d'exploitation pourrait même stocker des données d'un processus sur le disque. `Exec` fournit aussi aux utilisateurs la commodité d'un système de fichiers pour stocker les images des programmes exécutables.

Au sein d'Unix, plusieurs formes d'interactions se produisent via les descripteurs de fichier. Non seulement ils abstraient de nombreux détails (comme par exemple l'emplacement des données d'un tube ou d'un fichier), mais ils sont aussi définis de sorte que les interactions soient simplifiées. Par exemple, si une application plante dans un pipeline, le noyau génère une fin de fichier pour le prochain processus du pipeline.

Comme vous pouvez le constater, l'interface des appels système de la figure 2 est soigneusement conçue pour fournir à la fois un confort de programmation et une isolation stricte. L'interface Unix n'est pas la seule méthode pour abstraire les ressources, mais elle a prouvé en être une très efficace.

1.2 Mode utilisateur, mode noyau et appels système

L'isolation stricte nécessite une frontière nette entre applications et système d'exploitation. Si une application fait une erreur, il ne faut pas que le système d'exploitation ou une autre application plante. À la place, le système d'exploitation devrait être capable de nettoyer l'application ayant échoué et de continuer à exécuter les autres applications. Pour atteindre une isolation stricte, le système d'exploitation doit faire en sorte que les applications ne puissent pas modifier (ni même lire) les structures de données du système d'exploitation et les instructions, et que les applications n'aient pas accès à l'espace mémoire d'autres processus.

Les processeurs fournissent un support matériel pour l'isolation stricte. Par exemple, le processeur x86, comme bien d'autres processeurs, possède deux modes dans lesquels il peut exécuter les instructions : le *mode noyau* et le *mode utilisateur*. Dans le mode noyau, le processeur a le droit d'exécuter des *instructions privilégiées*. Par exemple, lire et écrire sur le disque (ou tout autre périphérique d'entrée/sortie) met en œuvre des instructions privilégiées. Si une application en mode utilisateur tente d'exécuter une instruction privilégiée, alors le processeur n'exécute pas cette instruction mais passe en mode noyau pour que le programme en mode noyau puisse nettoyer l'application, puisqu'elle a fait quelque chose qu'elle n'aurait pas dû faire. La figure 1 du chapitre 0 (page 10) illustre cette organisation. Une application ne peut exécuter que des instructions en mode utilisateur (par exemple additionner des nombres...). Il est dit qu'elle s'exécute dans l'*espace utilisateur*, alors que le programme en mode noyau peut aussi exécuter des instructions privilégiées. Il est dit qu'il s'exécute dans l'*espace noyau*. Le programme s'exécutant dans l'espace noyau est appelé le *noyau*.

Une application qui veut lire ou écrire un fichier sur le disque doit passer par le noyau pour faire cela, puisqu'elle même ne peut pas exécuter des instructions d'entrées/sorties. Les processeurs fournissent une instruction spéciale qui bascule le processeur du mode utilisateur vers le mode noyau et entre dans le noyau au point d'entrée défini par le noyau (le processeur x86 fournit pour ce faire l'instruction `int`). Une fois le processeur passé en mode noyau, le noyau peut valider les arguments de l'appel système, décider si l'application est autorisée à effectuer l'opération demandée pour ensuite la rejeter ou l'exécuter. Il est important que le noyau définisse le point d'entrée lors de la transition vers le mode noyau ; si l'application pouvait décider elle-même, une application malicieuse pourrait entrer dans le noyau après la validation des arguments, par exemple.

1.3 Organisation du noyau

Une question clef de la conception est de décider quelle partie du système d'exploitation doit s'exécuter en mode noyau. Une possibilité serait de faire résider l'intégralité du système d'exploitation dans le noyau, de sorte que l'implémentation de tous les appels système s'exécute en mode noyau. Cette organisation s'appelle *noyau monolithique*.

Avec cette organisation, l'intégralité du système d'exploitation s'exécute avec tous les privilèges matériels. Cette organisation est pratique car le concepteur du système n'a pas besoin de déterminer quelle partie du système d'exploitation n'a pas besoin de tous les privilèges matériels. En outre, la coopération entre les différentes parties du système d'exploitation est facilitée. Par exemple, un système d'exploitation peut avoir un *buffer-cache* [2] qui peut être partagé à la fois par le système de fichiers et par le système de mémoire virtuelle.

Un inconvénient de cette organisation monolithique est que les interfaces entre les différentes parties du système d'exploitation sont souvent complexes (comme nous le verrons dans la suite de ce texte), et c'est pourquoi un développeur de système d'exploitation peut facilement faire des erreurs. Dans un noyau monolitique, une erreur est fatale, car une erreur en mode noyau va souvent provoquer l'arrêt du noyau. Si le noyau plante, l'ordinateur cesse de fonctionner, ainsi que toutes les applications. L'ordinateur doit redémarer pour fonctionner à nouveau.

Pour réduire le risque d'erreur dans le noyau, les concepteurs de systèmes d'exploitation peuvent minimiser la partie du système qui s'exécute en mode noyau, et exécuter la plus grande part du système en mode utilisateur. Une telle organisation du noyau est appelé un *micro-noyau*.

2. NdT : Si *buffer* se traduit habituellement par « tampon » et *cache* par « cache », nous avons préféré conserver ici le terme original *buffer-cache* qui correspond à un composant interne que l'on retrouve dans la plupart des systèmes Unix.

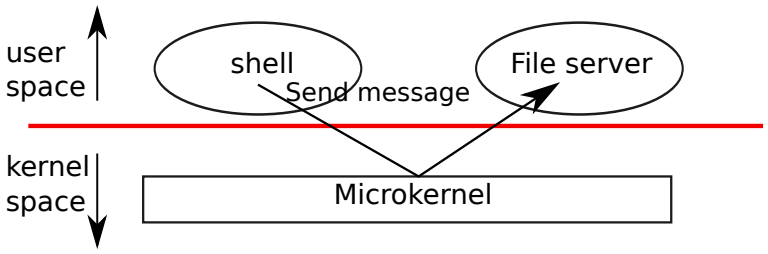

FIGURE 1.1 – Un micro-noyau avec un serveur système de fichier.

La figure 1.1 illustre le fonctionnement d'un micro-noyau : le système de fichiers s'exécute comme un processus au niveau utilisateur. Les services du système d'exploitation s'exécutant comme des processus sont appelés serveurs. Pour permettre aux applications d'interagir avec le système de fichiers, le noyau fournit un mécanisme de communication pour envoyer des messages depuis un processus en mode utilisateur vers un autre. Par exemple, si une application comme le shell veut lire ou écrire un fichier, elle envoie un message au serveur de fichiers et attend une réponse.

Dans un micro-noyau, l'interface du noyau est composée d'un petit nombre de fonctions de bas niveau permettant de démarrer une application, envoyer des messages, accéder aux périphériques, etc. Cette organisation permet au noyau de rester relativement simple, puisque la plupart des fonctions du système d'exploitation résident dans des serveurs au niveau utilisateur.

Xv6 est implémenté comme un noyau monolithique, comme la majorité des systèmes d'exploitation Unix. Ainsi, dans xv6, l'interface du noyau correspond à celle du système d'exploitation et le noyau implémente l'intégralité du système d'exploitation. Comme xv6 ne fournit pas beaucoup de services, son noyau est plus petit que certains micro-noyaux.

1.4 Aperçu des processus

L'unité d'isolation de xv6 (comme pour d'autres systèmes d'exploitation Unix) est un *processus*. L'abstraction processus empêche un processus de corrompre ou d'espionner l'espace mémoire d'un autre processus, le processeur, les descripteurs de fichier... Il empêche aussi les processus de corrompre le noyau lui-même, de sorte qu'un processus ne puisse outrepasser le mécanisme d'isolation du noyau. Le noyau doit implémenter l'abstraction processus avec attention car une application boguée ou maligne peut tromper le noyau ou le matériel en faisant quelque chose de mal (par exemple en contournant l'isolation forcée). Les mécanismes utilisés par le noyau pour implémenter les processus incluent l'indicateur du mode utilisateur/noyau, l'espace d'adressage et le découpage temporel des threads.

Pour aider à renforcer l'isolation, l'abstraction processus fournit à un programme l'illusion qu'il possède sa propre machine. Un processus fournit à un programme ce qui semble être une mémoire privée, ou un *espace d'adressage* dans lequel d'autres processus ne peuvent pas lire ou écrire. Un processus fournit également au programme ce qui paraît être son processeur personnel pour exécuter ses instructions.

Xv6 utilise des tables de pages (implémentées par le matériel) pour donner à chaque processus son propre espace d'adressage. La table des pages de x86 traduit [3] une *adresse virtuelle* (l'adresse manipulée par une instruction x86) en une *adresse physique* (l'adresse que la puce processeur

3. NdT : Le terme original est *map*.

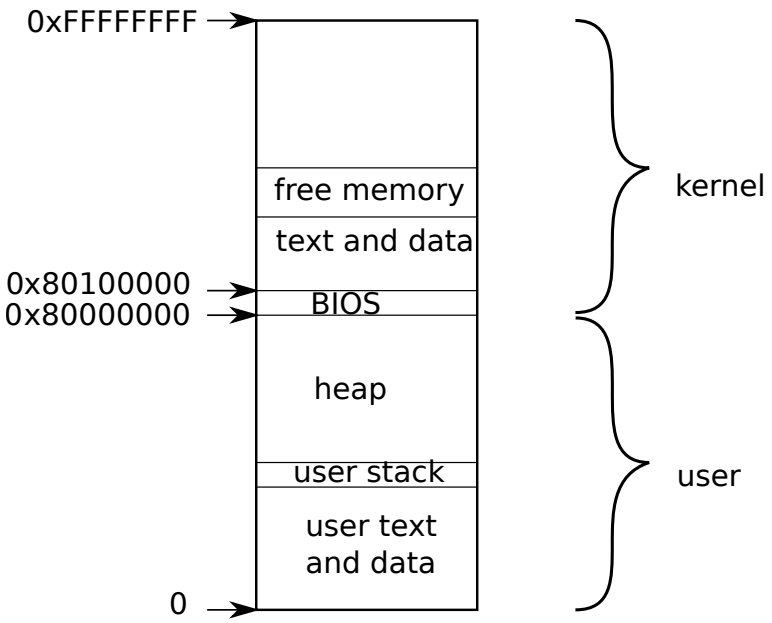

FIGURE 1.2 – Disposition d'un espace d'adressage virtuel.

envoie à la mémoire principale).

Xv6 maintient une table des pages séparée pour chaque processus qui définit l'espace d'adressage de ce processus. Comme illustré par la figure 1.2, un espace d'adressage comprend l'*espace mémoire utilisateur* qui commence à l'adresse virtuelle zéro. Les instructions viennent en premier, suivies des variables globales, puis la pile et finalement l'espace du *tas* (pour `malloc`) que le processus peut étendre au besoin.

Chaque espace d'adressage de processus inclut les instructions du noyau et les données ainsi que l'espace mémoire du programme utilisateur. Lorsqu'un processus réalise un appel système, la primitive s'exécute dans la partie réservée au noyau de l'espace d'adressage du processus. Cette disposition permet au code de l'appel système dans le noyau de référencer directement l'espace mémoire utilisateur. Pour laisser beaucoup de place pour l'espace mémoire utilisateur, l'espace d'adressage de xv6 place le noyau à des adresses élevées, commençant à l'adresse `0x80100000`.

Le noyau de xv6 conserve plusieurs informations d'état pour chaque processus, qu'il rassemble dans une `struct proc` [2337]. Les informations d'état du noyau les plus importantes pour un processus sont sa table des pages, sa pile noyau et son état d'exécution. Nous utiliserons la notation `p->xxx` pour faire référence aux éléments de la structure `proc`.

Chaque processus possède une file d'exécution (ou *thread*) qui exécute les instructions du processus. Un thread peut être suspendu et repris ultérieurement. Pour commuter de façon transparente entre les processus, le noyau suspend le thread en cours d'exécution et reprend le thread d'un autre processus. La majeure partie de l'état d'un thread (variables locales, adresses de retour des appels de fonctions) est stockée dans la pile du thread. Chaque processus possède deux piles : une pile utilisateur et une pile noyau (`p->kstack`). Lorsqu'un processus exécute des instructions utilisateurs, seule sa pile utilisateur est utilisée, sa pile noyau est vide. Lorsque le processus entre dans le noyau (pour un appel système ou une interruption), le code noyau s'exécute avec la pile noyau ; tant qu'un processus est dans le noyau, sa pile utilisateur contient toujours les données sauvegardées, mais est inutilisée. Le thread d'un processus alterne entre

l'utilisation active de ses piles utilisateur et noyau. La pile noyau est séparée (et protégée du code utilisateur) afin que le noyau puisse s'exécuter même si un processus a corrompu sa pile utilisateur.

Lorsqu'un processus fait un appel système, le processeur bascule vers la pile noyau, élève le niveau de privilège matériel et commence à exécuter les instructions du noyau qui implémentent l'appel système. Lorsque l'appel système est terminé, le noyau revient à l'espace utilisateur : le matériel abaisse son niveau de privilège, bascule vers la pile utilisateur et recommence à exécuter les instructions utilisateurs qui suivent directement l'instruction d'appel système. Un thread de processus peut se « bloquer » dans le noyau pour attendre une entrée/sortie puis reprendre où il s'était arrêté lorsque l'entrée/sortie est terminée.

`p->state` indique si le processus est alloué, prêt à s'exécuter, en cours d'exécution, en attente d'entrée/sortie ou en cours de terminaison.

`p->pgdir` contient la table des pages du processus, au format attendu par le matériel x86. Lors de l'exécution d'un processus, xv6 indique l'adresse dans `p->pgdir` à la MMU[4]. La table des pages d'un processus sert également à mémoriser les adresses des pages physiques allouées à l'espace mémoire du processus.

1.5 Code : le premier espace d'adressage

Pour rendre l'organisation de xv6 plus concrète, nous allons voir comment le noyau crée le premier espace d'adressage (pour lui-même), comment il crée et démarre le premier processus et comment ce processus effectue le premier appel système. En retraçant ces opérations, nous verrons en détail comment xv6 fournit une forte isolation pour les processus. La première étape pour fournir une forte isolation est de configurer le noyau afin qu'il s'exécute dans son propre espace d'adressage.

Lorsqu'un PC s'allume, il s'initialise puis charge un *chargeur d'amorçage* depuis le disque vers la mémoire avant de l'exécuter. L'annexe B détaille ces opérations. Le chargeur d'amorçage charge le noyau de xv6 depuis le disque et l'exécute en commençant à `entry` [1044]. La MMU n'est pas activée lorsque le noyau démarre : les adresses virtuelles correspondent directement aux adresses physiques.

Le chargeur d'amorçage charge le noyau dans la mémoire à l'adresse physique `0x100000`. La raison pour laquelle il ne charge pas le noyau à l'adresse `0x80100000`, où le noyau s'attend à trouver ses instructions et données, est qu'il peut ne pas y avoir de mémoire physique à des adresses si hautes sur de petites machines. La raison pour laquelle il place le noyau à l'adresse `0x100000` plutôt que `0x0` est que la plage d'adresses `0xa0000:0x100000` contient des périphériques d'entrées/sorties.

Pour permettre au reste du noyau de s'exécuter, `entry` crée une table des pages qui fait correspondre les adresses virtuelles à partir de l'adresse `0x80000000` (appelée `KERNBASE` [0207]) aux adresses physiques depuis l'adresse `0x0` (voir 1.3). Mettre en place deux plages d'adresses virtuelles qui relient la même plage de mémoire physique est une utilisation commune des tables de pages, et nous verrons d'autres exemples comme celui-ci.

La table des pages de `entry` est définie dans `main.c` [1306]. Nous étudierons les détails des tables de pages dans le chapitre 2, mais pour faire court, l'entrée 0 relie l'adresse virtuelle `0:0x400000` à l'adresse physique `0:0x400000`. Cette traduction d'adresses est nécessaire aussi longtemps que `entry` s'exécute à des adresses basses, mais sera supprimée à la fin.

4. NdT : Le terme original est *paging hardware*. Plutôt que traduire ce terme, nous préférons utiliser MMU (pour *Memory Management Unit*), qui désigne le composant matériel qui effectue entre autres la pagination.

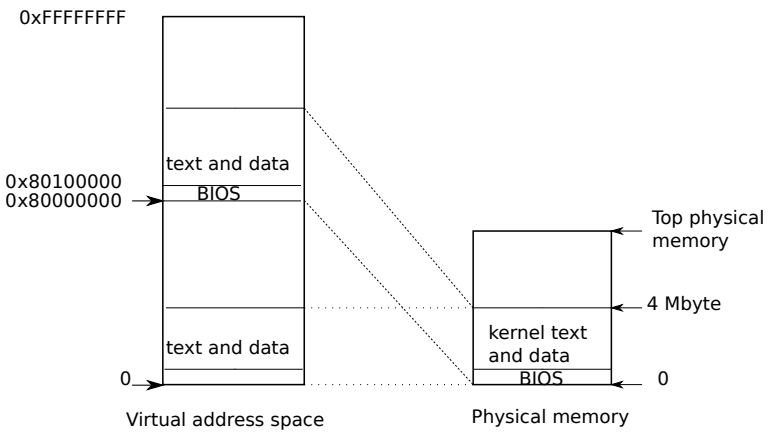

FIGURE 1.3 – Agencement d'un espace d'adressage virtuel de processus.

L'entrée 512 lie les adresses virtuelles `KERNBASE:KERNBASE+0x400000` aux adresses physiques `0:0x400000`. Cette entrée sera utilisée par le noyau après la fin de `entry` ; elle convertit les adresses virtuelles hautes auxquelles le noyau s'attend à trouver ses instructions et données, aux adresses physiques basses où le chargeur d'amorçage les a chargées. Cette traduction restreint les instructions et données noyaux à 4 Mo.

`Entry` charge ensuite l'adresse physique de `entrypgdir` dans le registre de contrôle `%cr3`. La valeur dans `%cr3` doit être une adresse physique. Cela n'aurait pas de sens de mettre l'adresse virtuelle de `entrypgdir` dans `%cr3` puisque la MMU ne sait pas encore comment traduire les adresses virtuelles ; elle ne possède pas de table des pages pour le moment. Le symbole `entrypgdir` fait référence à une adresse haute en mémoire, c'est pourquoi la macro `V2P_WO` [0213] soustrait `KERNBASE` afin de trouver l'adresse physique. Pour activer la MMU, xv6 met l'indicateur `CR0_PG` dans le registre de contrôle `%cr0`.

Le processeur est toujours en train d'exécuter des instructions à des adresses basses après l'activation de la MMU, ce qui n'est possible que parce que `entrypgdir` traduit aussi les adresses basses. Si xv6 avait omis l'entrée 0 de `entrypgdir`, l'ordinateur aurait crashé en essayant d'exécuter l'instruction suivant celle qui active la pagination.

Maintenant, `entry` doit passer au code C du noyau et l'exécuter à des adresse hautes. Tout d'abord, il fait pointer le pointeur de pile `%esp` sur l'emplacement mémoire à utiliser comme pile [1058]. Tous les symboles, incluant `stack`, ont des adresses hautes de sorte que la pile sera toujours valide, même après la fin de la traduction des adresses basses. Finalement, `entry` passe à `main`, qui est aussi une adresse haute. Un saut indirect est requis, car sinon l'assembleur aurait généré un saut direct relatif à PC, ce qui aurait exécuté la version mémoire basse de `main`. `Main` ne doit pas se terminer car il n'y a pas d'adresse de retour (PC) sur la pile. Le noyau s'exécute maintenant à des adresses hautes dans la fonction `main` [1217].

1.6 Code : Créer le premier processus

Nous allons maintenant regarder comment le noyau crée les processus utilisateurs et comment il s'assure qu'ils soient fortement isolés.

Après que `main` [1217] ait initialisé plusieurs périphériques et sous-systèmes, il crée le premier processus en appelant `userinit` [2520]. La première action de `userinit` est d'appeler `allocproc`. Le rôle de `allocproc` [2473] est d'allouer un emplacement (une `struct proc`)

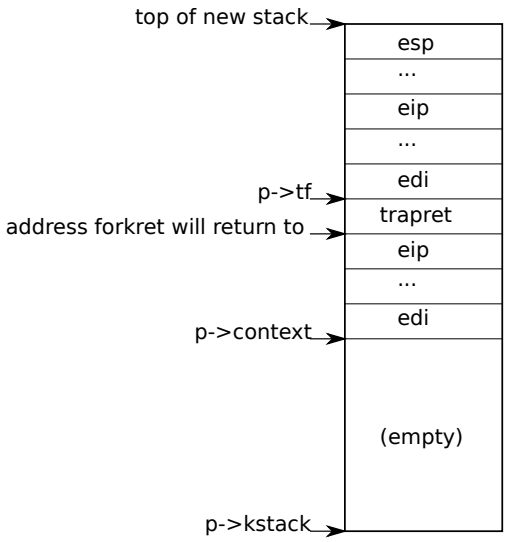

FIGURE 1.4 – Une nouvelle pile noyau.

dans la table des processus et d'initialiser les parties de l'état du processus requises pour permettre à son thread noyau de s'exécuter. `Allocproc` est appelé pour chaque nouveau processus alors que `userinit` n'est appelé que pour le tout premier. `Allocproc` balaye la table `proc` pour y trouver un emplacement avec l'état `UNUSED` [2480-2482]. Lorsqu'il trouve un emplacement inutilisé, `allocproc` met l'état à `EMBRYO` afin de le marquer comme utilisé et donne au processus un `pid` unique [2469-2489]. Puis, il essaye d'allouer une pile noyau pour le thread noyau du processus. Si cette allocation mémoire échoue, `allocproc` remet l'état à `UNUSED` puis renvoie zéro pour indiquer l'échec.

Maintenant, `allocproc` doit initialiser la nouvelle pile noyau. La fonction `allocproc` est écrite de façon à pouvoir être utilisée aussi bien par `fork` que lors de la création du premier processus. `Allocproc` initialise le processus avec une pile noyau spécialement préparée et des registres noyau qui provoquent le « retour » à l'espace utilisateur lors de sa première exécution. La pile noyau sera agencée comme décrit dans la figure 1.4. `Allocproc` fait une partie de ce travail en définissant des valeurs de retour du compteur ordinal qui provoqueront l'exécution, par le thread noyau du nouveau processus, d'abord de `forkret` puis de `trapret` [2507-2512]. Le thread du noyau va commencer son exécution avec le contenu de ses registres copié depuis `p->context`. Ainsi, mettre l'adresse de `forkret` dans `p->context->eip` fera que le thread noyau commencera par l'exécution de `forkret` [2853]. Cette fonction va retourner à l'adresse, quelle qu'elle soit, se trouvant au bas de la pile. Le code de changement de contexte [3059] positionne le pointeur de pile juste après `p->context`. `Allocproc` place `p->context` sur la pile et ajoute un pointeur sur `trapret` juste au-dessus : c'est là que `forkret` retournera ; ainsi, lorsque `forkret` se terminera, `trapret` restaurera les registres utilisateurs à partir des valeurs stockées en haut de la pile puis donnera la main au processus [3324]. Ce fonctionnement est identique pour un `fork` ordinaire et pour la création du premier processus, bien que dans ce dernier cas, le processus commencera en s'exécutant à l'adresse zéro de l'espace utilisateur plutôt qu'au retour de `fork`.

Comme nous le verrons dans le chapitre 3, le contrôle est transféré du programme utilisateur vers le noyau via un mécanisme d'interruptions utilisé par les appels système, les interruptions et les exceptions. À chaque fois que le contrôle est transféré dans le noyau alors qu'un processus est en cours d'exécution, le code d'entrée de déroutement de xv6 ainsi que le ma-

tériel[5] sauvegardent les registres utilisateurs sur la pile noyau du processus. `Userinit` écrit les valeurs sur le dessus de la nouvelle pile exactement comme si le processus était entré dans le noyau via une interruption [2533-2539], pour que le code habituel de retour du noyau au processus utilisateur fonctionne. Ces valeurs sont dans une `struct trapframe` qui contient les registres utilisateurs. Dès lors, la nouvelle pile noyau du processus est entièrement préparée comme illustré sur la figure 1.4.

Le premier processus va exécuter un petit programme (`initcode.S`; [8400]). Le processus a besoin de mémoire physique pour placer ce programme, qui doit y être copié, et le processus doit avoir une table des pages pour lier les adresses de l'espace utilisateur à cette mémoire.

`Userinit` appelle `setupkvm` [1818] pour créer une table des pages pour le processus avec, pour commencer, uniquement la traduction des adresses pour la mémoire utilisée par le noyau. Nous étudierons cette fonction en détail dans le chapitre 2, mais dans les grandes lignes, `setupkvm` et `userinit` créent un espace d'adressage comme illustré sur la figure 1.2.

Le contenu initial de l'espace mémoire utilisateur du premier processus est la forme compilée de `initcode.S`; pendant la compilation du noyau, l'éditeur de liens ajoute le code binaire dans le noyau et définit deux symboles spéciaux, `_binary_initcode_start` et `_binary_initcode_size` indiquant l'emplacement et la taille du binaire. `Userinit` le copie dans la nouvelle mémoire du processus en appelant `inituvm` qui alloue une page de mémoire physique, lie l'adresse virtuelle zéro à cette page et y copie le binaire [1886].

Ensuite, `userinit` initialise le contexte de trappe[6] dans la pile [0602] avec l'état initial du mode utilisateur : le registre `%cs` contient un sélecteur de segment pour le segment `SEG_UCODE` s'exécutant avec le niveau de privilège `DPL_USER` (c'est-à-dire en mode utilisateur plutôt qu'en mode noyau), et de manière analogue, `%ds`, `%es` et `%ss` utilisent `SEG_UDATA` avec le niveau de privilège `DPL_USER`. Le bit `FL_IF` du registre `%eflags` est mis à 1 pour permettre la prise en compte des interruptions matérielles[7] ; nous réexaminerons cela dans le chapitre 3.

Le pointeur de pile `%esp` est initialisé à la plus grande adresse virtuelle valide du processus, `p->sz`. Le pointeur d'instruction est mis au point d'entrée de initcode, soit l'adresse 0.

La fonction `userinit` initialise `p->name` à `initcode` principalement pour le débogage. Changer la valeur de `p->cwd` modifie le répertoire de travail courant du processus ; nous étudierons `namei` en détail dans le chapitre 6.

Une fois le processus initialisé, `userinit` le marque comme étant disponible pour l'ordonnancement en changeant la valeur de `p->state` à `RUNNABLE`.

1.7 Code : Exécuter le premier processus

Maintenant que l'état du premier processus est préparé, il est temps de l'exécuter. Après que `main` ait appelé `userinit`, `mpmain` appelle `scheduler` pour commencer à exécuter des processus [1257]. La fonction `scheduler` [2758] recherche un processus dont `p->state` vaut

5. NdT : Le matériel se charge de sauvegarder ce qui ne peut pas l'être par le système comme le compteur ordinal. Le processeur sauvegarde ainsi dans l'ordre les registres `%ss`, `%esp`, `%eflags`, `%cs`, `%eip` et `%err`.

6. NdT : Nous utiliserons « contexte de trappe » comme traduction de l'expression originale *trap frame*.

7. NdT : Le dixième bit du registre `%eflags` contient le niveau de privilège pour les entrées/sorties (IOPL). Ce niveau n'importe pas quant à la possibilité d'exécuter des instructions privilégiées qui n'ont pas de rapport avec les E/S, définie par le niveau de privilège du segment code (voir section 3.2, chapitre 3). Cependant, l'IOPL doit être supérieur ou égal au niveau de privilège courant (CPL), sans quoi il n'est pas pris en compte. Ainsi, dans xv6, l'IOPL vaut toujours 0, soit le niveau de privilège maximum ; en mode utilisateur, le CPL sera toujours supérieur à l'IOPL qui sera alors ignoré, les instructions d'E/S seront proscrites ; en mode noyau, le CPL comme l'IOPL vaudront 0 et ainsi les instructions d'E/S seront autorisées. Ceci permet de ne pas avoir à modifier systématiquement l'IOPL en plus du CPL.

RUNNABLE, et il n'y en a qu'un : initproc. Elle initialise la variable proc (propre à chaque processeur) avec le processus qu'elle a trouvé et appelle switchuvm pour dire au matériel d'utiliser la table des pages du processus cible [1879]. Changer la table des pages pendant l'exécution dans le noyau fonctionne car setupkvm fait en sorte que les tables de pages de tous les processus font une traduction identique des adresses du code et des données du noyau. Switchuvm initialise également un segment TSS (*Task State Segment*) SEG_TSS qui commande au matériel d'exécuter les appels système et les interruptions avec la pile noyau du processus. Nous réexaminerons le segment TSS dans le chapitre 3.

La fonction scheduler modifie maintenant p->state à RUNNING et appelle swtch [3059] afin d'effectuer un changement de contexte vers le thread noyau du processus cible. Swtch commence par sauvegarder les registres courants. Le contexte courant n'est pas celui d'un processus, mais plutôt un contexte particulier pour l'ordonnancement, propre à chaque processeur. Ainsi scheduler indique à swtch de sauvegarder les registres actuels dans un emplacement propre au processeur (cpu->scheduler) plutôt que dans le contexte de thread noyau de n'importe quel processus. Swtch charge ensuite les registres sauvegardés du thread noyau cible (p->context) dans les registres du processeur, incluant les pointeurs de pile et d'instruction. Nous étudierons swtch plus en détail dans le chapitre 5. L'instruction finale ret [3078] dépile %eip de la pile du processus courant, terminant la commutation de processus. Maintenant, le processeur s'exécute sur la pile noyau du processus p.

Allocproc avait précédemment mis la valeur forkret dans p->context->eip (dans initproc) de sorte que le ret commence l'exécution de forkret. Au premier appel (c'est celui-ci), forkret [2853] exécute des fonctions d'initialisation qui ne peuvent être exécutées dans main car elles nécessitent le contexte d'un processus normal avec sa propre pile noyau. Ensuite, forkret se termine. Allocproc s'est arrangé pour que le mot au sommet de la pile, après que p->context soit dépilé, soit trapret, ainsi maintenant trapret commence à s'exécuter avec %esp initialisé à p->tf. Trapret [0602] utilise les instructions de dépilement pour restaurer les registres depuis le contexte de trappe [0602] tout comme swtch le faisait avec le contexte noyau : l'instruction popl restaure les registres %gs, %fs, %es et %ds. L'instruction addl passe outre les deux champs trapno et errcode. Enfin, l'instruction iret dépile les registres %cs, %eip, %flags, %esp et %ss. Le contenu du contexte de trappe a été transféré vers le processeur, qui continue l'exécution depuis la valeur de %eip spécifiée dans le contexte de trappe. Pour initproc, cela correspond à l'adresse virtuelle zéro, la première instruction de initcode.S.

À ce stade, %eip contient la valeur zéro et %esp 4096. Il s'agit d'adresses virtuelles dans l'espace d'adressage du processus. La MMU les traduit en adresses physiques. Allocuvm a initialisé la table des pages du processus de sorte que l'adresse virtuelle zéro fasse référence à la mémoire physique allouée pour ce processus, et mis un indicateur (PTE_U) pour signifier à la MMU que le code utilisateur est autorisé à accéder à cet espace. Le fait que userinit [2533] initialise les bit de poids faible de %cs pour exécuter le code utilisateur avec un niveau de privilège courant (CPL) égal à 3 signifie que le code utilisateur ne peut utiliser que des pages avec l'indicateur PTE_U, et ne peut pas modifier les registres matériels sensibles comme %cr3. Ainsi, le processus est contraint d'utiliser uniquement son propre espace mémoire.

1.8 Code : Le premier appel système : exec

Maintenant que nous avons vu comment le noyau fournit une isolation forte pour les processus, regardons comment un processus utilisateur entre à nouveau dans le noyau pour demander des services qu'il ne peut accomplir par lui-même.

La première action de `initcode.S` est de demander l'appel système `exec`. Comme vu dans le chapitre 0, `exec` remplace les registres et la mémoire du processus courant par un nouveau programme, mais conserve les descripteurs de fichier, l'identité du processus et son parent.

`Initcode` [8409] commence par empiler trois valeurs – `$argv`, `$init` et `$0` – et met la valeur `SYS_exec` dans le registre `%eax` avant d'exécuter l'instruction `int T_SYSCALL` : il demande au noyau d'exécuter l'appel système `exec`. Si tout se passe bien, `exec` ne retourne jamais : il commence à exécuter le programme nommé par `$init`, qui est un pointeur sur la chaîne de caractères `init` se terminant par "\0" [8422-8424]. L'autre argument est le tableau des arguments en ligne de commande `argv` ; sa fin est marquée par un zéro. Si `exec` échoue et retourne, `initcode` boucle en demandant l'appel système `exit`, qui ne devrait sûrement pas retourner [8416-8420].

Ce code construit manuellement le premier appel système, qui ressemble aux appels système ordinaires que nous verrons dans le chapitre 3. Comme précédemment, cette configuration évite de faire du premier processus (dans ce cas, de son premier appel système) un cas particulier et réutilise plutôt du code que xv6 doit fournir pour des opérations standards.

Le chapitre 2 couvrira en détail l'implémentation de `exec` mais, pour résumer, il remplace `initcode` avec le binaire de `/init`, chargé depuis le système de fichiers. Maintenant, `initcode` [8400] est terminé et le processus exécutera `/init` à sa place.

`Init` [8510] crée si nécessaire un nouveau fichier console de type « périphérique » puis l'ouvre pour donner les descripteurs de fichier 0, 1 et 2. Puis il boucle, démarrant un shell, s'occupant des processus zombies orphelins jusqu'à ce que le shell se termine, puis recommence. Le système est en place.

1.9 Le monde réel

Dans le monde réel, on peut trouver à la fois des noyaux monolithiques et des micro-noyaux. Beaucoup de noyaux Unix sont monolithiques. Par exemple, Linux est un noyau monolithique bien que certaines fonctions du système s'exécutent comme des serveurs au niveau utilisateur (par exemple le système de fenêtrage). Les noyaux comme L4, Minix ou QNX sont organisés comme des micro-noyaux avec des serveurs, et ont été largement déployés dans des systèmes embarqués.

La plupart des systèmes d'exploitation ont adopté le concept de processus, et la plupart des processus ressemblent à ceux de xv6. Un système d'exploitation réel trouverait des structures de processus libres `proc` en temps constant dans une liste contenant exclusivement les processus libres au lieu d'une recherche en temps linéaire comme celle de `allocproc` ; xv6 utilise une recherche linéaire (le premier parmi plusieurs) pour des raisons de simplicité.

1.10 Exercices

1. Mettez un point d'arrêt à `swtch`. Demandez l'exécution d'une instruction assembleur (en utilisant `stepi` dans `gdb`) jusqu'au retour à `forkret`, puis utilisez `finish` pour poursuivre vers `trapret`, et enfin `stepi` jusqu'à arriver à l'adresse virtuelle zéro de `initcode` [8].
2. `KERNBASE` limite la taille mémoire utilisable par un processus, ce qui pourrait être embêtant sur une machine de 4 Go de RAM. Augmenter `KERNBASE` permettrait-il d'utiliser

8. NdT : Je n'ai personnellement pas réussi à obtenir les résultats escomptés en suivant ces étapes.

plus de mémoire ?

Chapitre 2

Tables de pages

Les tables de pages sont le mécanisme par lequel les systèmes d'exploitation contrôlent la signification d'une adresse mémoire. Ils permettent à xv6 de multiplexer les espaces d'adressage de différents processus dans une même mémoire physique, et de protéger les mémoires des différents processus. Le niveau d'indirection fourni par les tables de pages permet de nombreuses astuces. Xv6 utilise principalement les tables de pages pour multiplexer les espaces d'adressage et protéger la mémoire. Il utilise aussi quelques astuces simples permises par les tables de pages : lier la même mémoire (le noyau) dans différents espaces d'adressage, lier la même mémoire plus d'une fois dans un espace d'adressage (chaque page utilisateur est ainsi liée à la mémoire physique du noyau) et conserver une pile utilisateur sur une page non liée. Le reste de ce chapitre explique les tables de pages fournies par le x86 et comment xv6 les utilise. Si on compare avec les systèmes d'exploitation réels, la conception de xv6 est restreinte mais illustre les idées clefs.

2.1 La MMU

Pour mémoire, les instructions x86 (à la fois en mode utilisateur ou noyau) manipulent des adresses virtuelles. La RAM, ou mémoire physique, est indexée par des adresses physiques. La MMU du x86 relie ces deux sortes d'adresses en traduisant chaque adresse virtuelle en une adresse physique.

Une table des pages x86 est donc tout logiquement un tableau de 2^{20} (1 048 576) entrées, ou PTE (*Page Table Entries*). Chaque PTE contient un numéro de page physique sur 20 bits ou PPN (*Physical Page Number*) ainsi que quelques indicateurs. La MMU traduit une adresse virtuelle en utilisant ses 20 bits de poids fort comme index dans la table des pages afin de trouver une entrée et en remplaçant les 20 bits de poids fort de l'adresse avec le PPN issu de la PTE. La MMU recopie les 12 bits de poids faible, les gardant inchangés durant la traduction de l'adresse virtuelle vers l'adresse physique. Ainsi, la table des pages donne au système d'exploitation le contrôle sur la traduction des adresses virtuelles vers des adresses physiques, à la granularité de zones alignées et de taille 4 096 (2^{12}) octets. Un telle zone est appelée *page*.

Comme le montre la figure 2.1, la traduction à proprement parler se fait en deux étapes. Une table des pages est stockée en mémoire physique comme un arbre à deux niveaux. La racine de cet arbre est le *répertoire des pages* de 4 096 octets, qui contient 1 024 références, similaires aux PTE, à des *pages de table des pages*. Chaque page de table des pages est un tableau de 1 024 PTE de 32 bits. La MMU utilise les 10 bits de poids fort d'une adresse virtuelle pour sélectionner une entrée du répertoire de pages. Si cette entrée est présente, la MMU utilise les 10 bits suivants de l'adresse virtuelle pour sélectionner une PTE depuis la page de tables de pages référencée

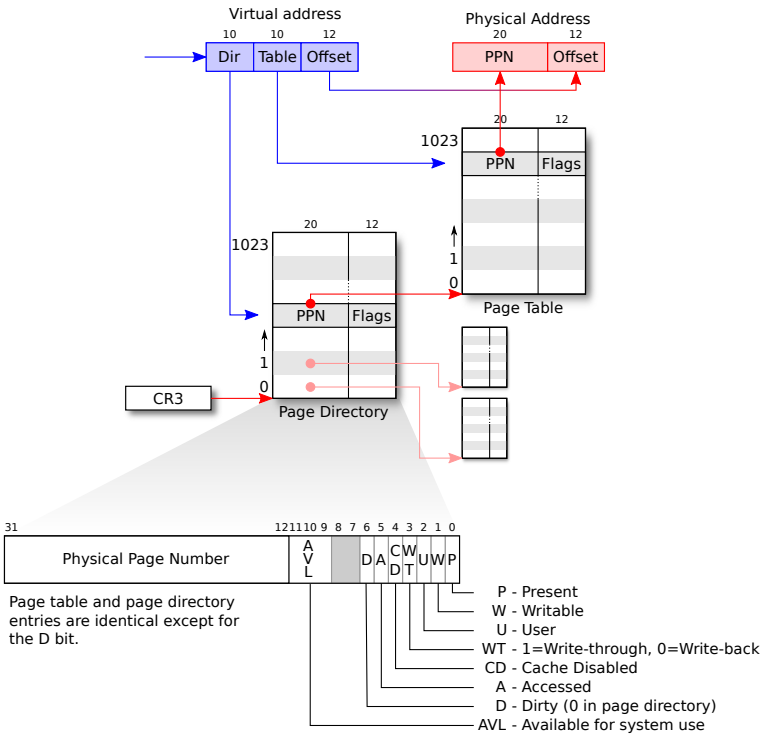

FIGURE 2.1 – MMU du x86.

par l'entrée du répertoire de pages. La MMU signale un défaut de page si l'entrée du répertoire de pages ou la PTE n'est pas présente, Cette structure à deux niveaux permet à une table des pages d'omettre des pages de table des pages entières dans le cas fréquent pour lequel de grandes plages d'adresses virtuelles n'ont pas de traduction.

Chaque PTE contient des bits (indicateurs) qui indiquent à la MMU la façon dont l'adresse virtuelle associée est autorisée à être utilisée. `PTE_P` indique si la PTE est présente : s'il n'est pas activé, une référence à cette page provoquera une erreur (c'est-à-dire qu'une telle référence n'est pas autorisée). `PTE_W` contrôle si les instructions sont autorisées à effectuer des écritures sur la page ; s'il est non défini, seules les lectures de données ou d'instructions sont autorisées. `PTE_U` contrôle si les programmes en mode utilisateur sont autorisés à utiliser cette page ; si cet indicateur est désactivé, seul le noyau est autorisé à utiliser la page. La figure 2.1 montre comment tout cela fonctionne. Les indicateurs et toutes les autres structures en relation avec la MMU sont définis dans `mmu.h` [0700].

Quelques précisions sur les termes. La mémoire physique fait référence aux cellules de stockage de la DRAM. Un octet de mémoire physique est situé à une adresse, appelée adresse physique. Les instructions n'utilisent que des adresses virtuelles, que la MMU traduit en adresses physiques puis envoie au composant matériel DRAM pour lire ou écrire dans les cellules de stockage. À ce stade de l'explication, il n'existe que des adresses virtuelles, la mémoire virtuelle, quant à elle, n'existe pas.

2.2 Espace d'adressage d'un processus

La table des pages créée par `entry` possède suffisamment d'entrées pour commencer l'exécution du code C du noyau. Cependant, `main` installe immédiatement une nouvelle table des pages dans `kvmalloc` [1840] car le noyau utilise une description plus élaborée de l'espace

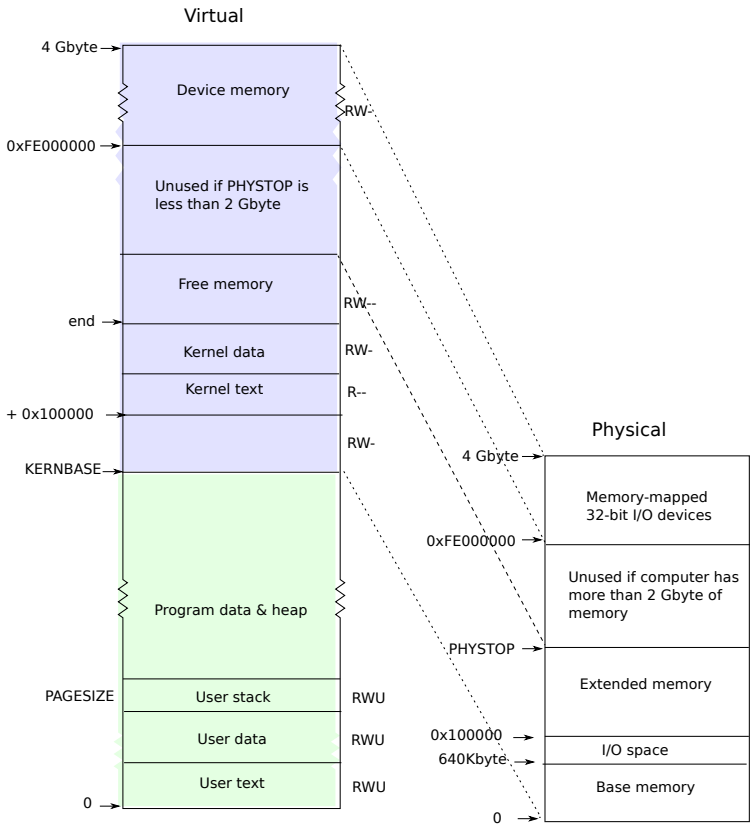

FIGURE 2.2 – Disposition de l'espace d'adressage virtuel et physique d'un processus. Notez que si une machine possède plus de 2 Go de mémoire physique, xv6 ne peut utiliser que l'espace compris entre `KERNBASE` et `0xFE00000`.

d'adressage des processus.

Chaque processus a une table des pages séparée et xv6 indique à la MMU de changer de table des pages lorsqu'il change de processus. Comme le montre la figure 2.2, un espace mémoire utilisateur commence à l'adresse virtuelle zéro et peut s'étendre jusqu'à `KERNBASE`, permettant à un processus d'adresser jusqu'à 2 gigaoctets de mémoire. Le fichier `memlayout.h` [0200] déclare les constantes pour l'agencement de la mémoire de xv6, et les macros pour convertir des adresses virtuelles en adresses physiques.

Lorsqu'un processus demande à xv6 plus de mémoire, xv6 commence par trouver une page physique libre pour fournir le stockage, puis ajoute les PTE à la table des pages du processus qui pointe sur cette nouvelle page physique. Xv6 active les indicateurs `PTE_U`, `PTE_W` et `PTE_P` pour ces PTE. La plupart des processus n'utilisent pas l'intégralité de l'espace d'adressage ; xv6 laisse l'indicateur `PTE_P` désactivé dans les PTE inutilisées. Les tables de pages de différents processus traduisent des adresses utilisateurs vers des pages différentes de la mémoire physique, de sorte que chaque processus possède sa propre mémoire utilisateur privée.

Xv6 inclut toutes les traductions nécessaires pour que le noyau puisse s'exécuter dans les tables de pages de chaque processus ; ces traductions apparaissent toutes après `KERNBASE`. Xv6 associe les adresses virtuelles `KERNBASE:KERNBASE+PHYSTOP` aux adresses physiques `0:PHYSTOP`. Une des raisons de cette traduction est de faire en sorte que le noyau puisse utiliser ses propres instructions et données. Une autre raison est que le noyau doit être parfois capable d'écrire dans une page donnée de la mémoire physique, par exemple lors de la création de pages de table des pages ; pour ce faire, il est pratique que chaque page physique apparaisse

à une adresse virtuelle prédictible. Un inconvénient de cet agencement est que xv6 ne peut pas utiliser plus de 2 Go de mémoire physique, car la partie noyau de l'espace d'adressage est de 2 Go. Ainsi, xv6 requiert que `PHYSTOP` soit plus petit que 2 Go, même si l'ordinateur possède plus de 2 Go de mémoire physique.

Certains périphériques qui utilisent l'adressage des entrées/sorties en mémoire apparaissent à des adresses physiques commençant à `0xFE000000`, c'est pourquoi les tables de pages de xv6 incluent une traduction directe pour eux. Ainsi, `PHYSTOP` doit être plus petit que 2 Go - 32 Mo (pour la mémoire des périphériques).

Xv6 n'active pas l'indicateur `PTE_U` dans les PTE au-delà de `KERNBASE`, donc seul le noyau peut les utiliser.

Avoir une traduction d'adresses à la fois pour la mémoire utilisateur et pour tout le noyau par la table des pages de tous les processus est commode lors du basculement entre code utilisateur et code noyau pendant les appels système et les interruptions : de tels basculements ne nécessitent pas de changement de table des pages. Dans la plupart des cas, le noyau ne possède pas sa propre table des pages ; il emprunte presque toujours la table des pages d'un processus.

Pour résumer, xv6 s'assure que chaque processus ne peut utiliser que sa propre mémoire. Chaque processus voit cette mémoire comme ayant des adresses virtuelles contiguës commençant à zéro, tandis que la mémoire physique du processus peut ne pas être contiguë. Xv6 implémente le premier paradigme en activant l'indicateur `PTE_U` uniquement sur les PTE d'adresses virtuelles faisant référence à la mémoire du processus lui-même. Il implémente le second en utilisant la capacité des tables de pages à traduire des adresses virtuelles successives en n'importe quelle page physique allouée au processus.

2.3 Code : créer un espace d'adressage

`Main` appelle `kvmalloc` [1840] pour créer et basculer vers une table des pages contenant des traductions pour les adresses supérieures à `KERNBASE` indispensables pour l'exécution du noyau. La plus grande partie de ce travail est faite par `setupkvm` [1818]. Cette fonction commence par allouer une page de mémoire pour contenir le répertoire des pages. Puis, elle appelle `mappages` pour installer les traductions dont le noyau a besoin, décrites dans le tableau `kmap` [1809]. La traduction inclut les données et instructions du noyau, la mémoire physique jusqu'à `PHYSTOP` et des plages mémoires qui sont en réalité des périphériques d'entrées/sorties. `Setupkvm` n'installe aucune traduction pour la mémoire utilisateur ; cela viendra plus tard.

`Mappages` [1760] installe des traductions dans une table des pages pour une plage d'adresses virtuelles vers la plage d'adresses physiques associée. Il fait ceci séparément pour chaque adresse virtuelle dans la plage, à des intervalles d'une page. Pour chaque adresse virtuelle, `mappages` appelle `walkpgdir` pour déterminer l'adresse de la PTE correspondante. `Mappages` initialise ensuite cette PTE pour qu'elle contienne le bon numéro de page physique, les permissions désirées (`PTE_W` et/ou `PTE_U`) ainsi que `PTE_P` pour marquer la PTE comme valide [1772].

La fonction `walkpgdir` [1735] imite les actions de la MMU puisqu'elle recherche dans les PTE une adresse virtuelle (voir figure 2.1). `Walkpgdir` utilise les 10 bits de poids fort de l'adresse virtuelle pour trouver l'entrée du répertoire de pages [1740]. Si cette entrée n'est pas présente, alors la table de pages demandée n'a pas encore été allouée ; si l'argument `alloc` vaut « vrai », `walkpgdir` alloue cette table et met son adresse physique dans le répertoire des pages. Enfin, `walkpgdir` utilise les 10 bits suivants de l'adresse virtuelle pour trouver l'adresse de la PTE dans la table des pages [1753].

2.4 Allocation de mémoire physique

Le noyau doit allouer et libérer la mémoire physique à l'exécution, que ce soit pour les tables de pages, la pile noyau et les tampons [1] de tubes.

Xv6 utilise la mémoire physique située entre la fin du noyau et `PHYSTOP` pour les allocations au cours de l'exécution. Il alloue et libère des pages entières de 4 096 octets à la fois. Il conserve une trace des pages libres en faisant passer une liste chaînée au travers des pages elles-mêmes. L'allocation consiste en la suppression d'une page de la liste chaînée ; la libération consiste en l'ajout de la page libre à la liste.

Il existe cependant un problème d'initialisation : toute la mémoire physique doit être traduite pour permettre à l'allocateur d'initialiser la liste des pages libres, mais créer une table des pages avec ces traductions implique d'allouer une page de tables de pages. Xv6 résout ce problème en utilisant durant l'initialisation un allocateur de page séparé qui alloue la mémoire juste après la fin du segment des données du noyau. Cet allocateur ne supporte pas la libération et est limité par une traduction de 4 Mo dans `entrypgdir`, mais cela reste suffisant pour allouer la première table des pages du noyau.

2.5 Code : Allocateur de mémoire physique

La structure de données de l'allocateur est une liste des pages physiques de la mémoire disponibles pour l'allocation. Chaque élément de cette liste est une `struct run` [3115]. Où l'allocateur trouve-t-il la place pour stocker cette structure de données ? Il place chaque structure `run` des pages disponibles au sein de la page disponible elle-même, puisque rien d'autre n'y est stocké. La liste des pages libres est protégée par un *spin lock* [2] [3119-3123]. La liste et le verrou sont emballés dans une structure pour clarifier le fait que le verrou protège les champs de la structure. Pour le moment, mieux vaut ignorer le verrou ainsi que les appels à `acquire` et `release` ; le chapitre 4 les détaillera.

La fonction `main` appelle `kinit1` et `kinit2` pour initialiser l'allocateur [3131]. La raison qui justifie deux fonctions est que durant la plus grande partie de `main`, il n'est pas possible d'utiliser de verrous ou de la mémoire au delà de 4 Mo. L'appel à `kinit1` met en place une allocation sans verrou dans les 4 premiers mégaoctets, tandis que l'appel à `kinit2` permet les verrous et l'allocation de davantage de mémoire. `Main` devrait déterminer la quantité de mémoire physique disponible, mais cela s'avère difficile sur une architecture x86. À la place, xv6 suppose que la machine possède 224 Mo (`PHYSTOP`) de mémoire physique, et utilise toute la mémoire située entre la fin du noyau et `PHYSTOP` comme réservoir initial de mémoire libre. `Kinit1` et `kinit2` appellent `freerange` pour ajouter de la mémoire à la liste des pages vides en appelant pour chaque page `kfree`. Une PTE ne peut faire référence qu'à une adresse physique alignée sur une limite de 4 096 octets (qui est un multiple de 4 096), aussi `freerange` utilise `PGROUNDUP` pour s'assurer qu'il libère seulement des adresses physiques alignées. L'allocateur commence sans mémoire ; ces appels à `kfree` lui en donnent à gérer.

L'allocateur fait référence à des pages physiques par leur adresse virtuelle et non par leur adresse physique ; c'est pourquoi `kinit` utilise `P2V(PHYSTOP)` pour traduire `PHYSTOP` (une adresse physique) en une adresse virtuelle. L'allocateur traite parfois les adresses comme des entiers afin d'effectuer des calculs arithmétiques (par exemple pour traverser toutes les pages

1. NdT : Le terme original *buffer* correspond à un espace mémoire temporaire. Nous utiliserons dans la suite le terme « tampon », bien que le terme original soit communément admis.
2. NdT : Le terme *spin lock* correspond à un « verrouillage avec attente active », mais nous avons préféré conserver l'expression anglaise, plus courte.

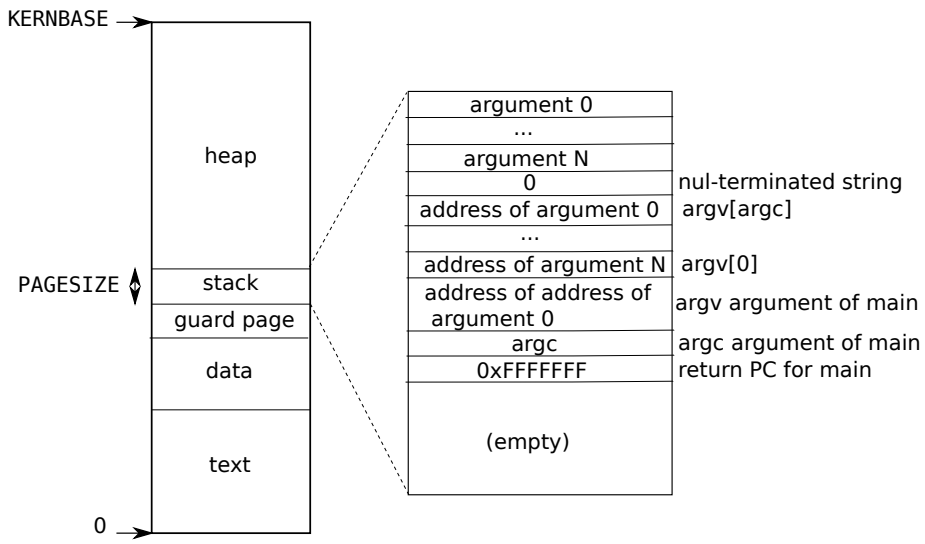

FIGURE 2.3 – Agencement en mémoire d'un processus utilisateur avec sa pile initiale.

dans `kinit`) et il les utilise parfois comme des pointeurs pour lire et écrire dans la mémoire (par exemple lorsqu'il manipule la structure `run` stockée dans chaque page) ; cette double utilisation est la raison principale pour laquelle le code de l'allocateur est truffé de conversions de types C (*cast*). L'autre raison est que la libération et l'allocation modifient implicitement le type de la mémoire.

La fonction `kfree` [3164] commence par mettre tous les octets de la mémoire libérée à 1. Ainsi, un code qui utiliserait cette mémoire après libération (qui utiliserait donc des « *dangling references* » [3]) lira des données erronées plutôt que d'anciennes données valides ; avec un peu de chance, le code se plantera plus rapidement. Ensuite, `kfree` convertit `v` en un pointeur sur `struct run`, sauvegarde l'ancien début de la liste des pages libres dans `r->next` et définit `r` comme nouveau début de la liste. `Kalloc` supprime puis renvoie le premier élément de la liste des pages libres.

2.6 Partie utilisateur d'un espace d'adressage

La figure 2.3 montre l'agencement de la mémoire utilisateur d'un processus en cours d'exécution avec xv6. Chaque processus utilisateur commence à l'adresse 0. La partie inférieure de l'espace d'adressage contient le texte [4] du programme utilisateur, ses données et sa pile. Le tas est situé au-dessus de la pile afin qu'il puisse s'étendre lorsque le processus fait un appel à `sbrk`. Il est à noter que les sections texte, données et pile sont disposées de manière contiguë dans l'espace d'adressage du processus, mais xv6 est libre d'utiliser des pages physiques non contiguës pour ces sections. Par exemple, lorsque xv6 étend le tas d'un processus, il peut utiliser n'importe quelle page physique libre pour la nouvelle page virtuelle, puis indiquer à la MMU la traduction de la page virtuelle vers la page physique allouée. Cette flexibilité est un avantage majeur de l'utilisation de la MMU.

La pile n'est constituée que d'une simple page, et figure sur le schéma avec son contenu initial

3. NdT : Le terme *dangling reference* désigne une référence vers une partie de la mémoire qui a été désallouée depuis la mise en place de cette référence. Une traduction pourrait être « référence pendouillante », mais nous avons préféré conserver l'expression originale.

4. NdT : Le terme *text* correspond habituellement au code binaire exécutable.

tel que créé par exec. Tout en haut de la pile, on retrouve les chaînes de caractères contenant les arguments entrés en ligne de commande, ainsi qu'un pointeur sur ces derniers. Juste en dessous se trouvent des valeurs permettant au programme de commencer à partir de main comme si la fonction main(argc, argv) venait d'être appelée. Pour se prémunir d'un débordement de la pile en dehors de sa page, xv6 place une page de garde juste en dessous. La page de garde n'est pas traduite, donc si la pile déborde de sa page, la MMU va générer une exception car elle ne peut pas traduire l'adresse fautive. Un vrai système d'exploitation pourrait allouer plus d'espace pour la pile afin qu'elle puisse croître au-delà d'une page.

2.7 Code : sbrk

Sbrk est l'appel système qui permet à un processus de réduire ou d'accroître son espace mémoire. Cet appel système est implémenté par la fonction growproc [2558]. Si n est positif, growproc alloue une ou plusieurs pages et les associe au sommet de l'espace d'adressage du processus. Si n est négatif, growproc retire la traduction d'une ou plusieurs pages de l'espace d'adressage du processus et libère les pages physiques correspondantes. Pour effectuer ces changements, xv6 modifie la table des pages du processus. Celle-ci est stockée en mémoire, ainsi le noyau peut la mettre à jour en utilisant des affectations ordinaires, ce que font allocuvm et deallocuvm. La MMU du x86 met les entrées de la table des pages dans un cache matériel nommé « TLB » (*Translation Look-aside Buffer*), et lorsque xv6 change de table des pages, il doit invalider les entrées de ce cache. S'il ne le fait pas, le TLB pourrait ultérieurement utiliser une ancienne traduction pointant sur une page physique qui aurait été entretemps allouée à un autre processus et, par conséquent, un processus pourrait être capable d'écrire dans la mémoire d'un autre processus. Xv6 invalide les entrées de cache obsolètes en rechargeant %cr3, le registre contenant l'adresse de la page des tables courante.

2.8 Code : exec

Exec est l'appel système qui crée la partie utilisateur d'un espace d'adressage. Il initialise cette partie depuis un fichier stocké dans le système de fichiers. Exec [6610] ouvre le fichier binaire nommé path en utilisant namei [6623], qui sera expliquée dans le chapitre 6. Puis, il lit l'en-tête ELF. Les applications de xv6 sont dans le « format ELF », format largement répandu et décrit dans elf.h. Un binaire ELF consiste en un en-tête ELF, struct elfhdr [0905], suivi d'une suite d'en-têtes de sections du programme, struct proghdr [0924]. Chaque proghdr décrit une section de l'application qui doit être chargée en mémoire ; avec xv6, les programmes ne possèdent qu'un seul en-tête de section, mais d'autres systèmes pourraient avoir des sections séparées pour les instructions et les données.

La première étape est une rapide vérification que le fichier soit bien un binaire au format ELF. Un tel fichier commence par le nombre de quatre octets 0x7F, 'E', 'L', 'F', ou ELF_MAGIC [0902] appelé « nombre magique ». Si l'en-tête ELF contient le bon nombre magique, exec considère que le binaire est bien formé.

Exec alloue une nouvelle table des pages sans traduction des adresses utilisateurs avec setupkvm [6651], et charge chaque segment en mémoire avec loaduvm [6655]. Allocuvm vérifie que l'adresse virtuelle demandée est inférieure à KERNBASE. Loaduvm [1903] utilise walkpgdir pour trouver l'adresse physique de la mémoire fraichement allouée dans laquelle placer chaque page du segment ELF, et readi pour lire depuis le fichier.

L'en-tête de la section du programme /init, le premier programme créé avec exec ressemble à cela :

```
# objdump -p _init

_init:     file format elf32-i386

Program Header:
    LOAD off    0x00000054 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x000008c0 memsz 0x000008cc flags rwx
```

Le champ `filesz` de l'en-tête de la section du programme peut être plus petit que `memsz`, indiquant que l'espace entre doit être comblé par des zéros (pour les variables globales en C) plutôt que lues depuis le fichier. Pour `/init`, `filesz` vaut 2 240 octets et `memsz` vaut 2 252 octets, ainsi `allocuvm` alloue suffisamment d'espace mémoire physique pour contenir 2 252 octets, mais ne lit que 2 240 octets depuis le fichier `/init`.

Maintenant, `exec` alloue et initialise la pile utilisateur. Il alloue une page unique pour la pile. `Exec` copie une par une les chaînes de caractères données en arguments au sommet de la pile, sauvegardant les pointeurs sur ces dernières dans `ustack`. Il place un pointeur nul à la fin de ce qui sera la liste `argv` passée à `main`. Les trois premières entrées dans `ustack` sont la fausse adresse de retour, ainsi que `argc` et le pointeur `argv`.

`Exec` place une page inaccessible juste en dessous de la page de pile, de sorte que les programmes qui essayent d'utiliser plus d'une page provoquent une erreur. Cette page inaccessible permet aussi à `exec` de gérer des arguments trop grands ; dans une telle situation, la fonction `copyout` [2118] qu'utilise `exec` pour copier les arguments sur la pile va remarquer que la page de destination n'est pas accessible et va renvoyer -1.

Pendant la préparation de la nouvelle image mémoire, si `exec` détecte une erreur telle qu'une section de programme invalide, il se rend à l'étiquette `bad`, libère la nouvelle image et renvoie -1. `Exec` doit attendre qu'il soit certain que l'appel système réussisse avant de libérer l'ancienne image : si l'ancienne image était perdue, l'appel système ne pourrait pas renvoyer -1. Les seuls cas d'erreur dans `exec` se produisent lors de la création d'une image. Une fois l'image complète, `exec` peut installer cette nouvelle image [6701] et libérer l'ancienne [6702]. Enfin, `exec` renvoie 0.

`Exec` charge des octets depuis un fichier ELF vers la mémoire à des adresses spécifiées dans le fichier ELF. Les utilisateurs et les processus peuvent placer les adresses de leur choix dans un fichier ELF. `Exec` est donc risqué car les adresses dans le fichier ELF peuvent faire référence, accidentellement ou non, au noyau. Les conséquences pour un noyau imprudent peuvent aller du *crash* à la corruption des mécanismes d'isolation du noyau (c'est-à-dire une faille de sécurité). Xv6 effectue un certain nombre de vérifications pour éviter ces risques. Pour comprendre l'importance de ces vérifications, considérons ce qui se passerait si xv6 ne vérifiait pas `if(ph.vaddr + ph.memsz < ph.vaddr)`. Il s'agit d'une vérification pour savoir si la somme dépasse un entier de 32 bits. Le danger est qu'un utilisateur pourrait construire un binaire ELF avec un `ph.vaddr` qui pointerait dans le noyau, et `memsz` suffisamment grand pour que la somme dépasse 0x1000. Tant que la somme est petite, elle passerait la vérification `if(newsz >= KERNBASE)` dans `allocuvm`. L'appel ultérieur à `loaduvm` transmettrait `ph.vaddr` directement, sans ajouter `ph.memsz` et sans vérifier `ph.vaddr` avec `KERNBASE`, et copierait ainsi des données depuis le binaire ELF dans le noyau. Ceci pourrait être exploité par un programme utilisateur pour exécuter un code utilisateur arbitraire avec les privilèges du noyau. Comme l'illustre cet exemple, la vérification des arguments doit être faite avec beaucoup de soin. Il est facile pour un développeur du noyau d'omettre une vérification cruciale, et les noyaux réels possèdent une longue histoire de vérifications manquantes, dont l'absence peut être exploitée par des programmes utilisateurs pour obtenir les privilèges du noyau. Il est probable que xv6 n'effectue pas un travail complet de validation des données utilisateurs fournies au noyau, ce qu'un programme utilisateur malveillant pourrait exploiter pour contourner

l'isolation offerte par xv6.

2.9 Le monde réel

Comme la plupart des systèmes d'exploitation, xv6 utilise la MMU pour la protection de la mémoire et la traduction des adresses. La plupart des systèmes d'exploitation utilise la MMU x86 64 bits (qui possède 3 niveaux de traduction). Des espaces d'adressages de 64 bits permettent une disposition mémoire moins restrictive que celle de xv6 ; il serait par exemple plus simple de supprimer la limite de 2 Gigaoctets de mémoire physique de xv6. La plupart des systèmes d'exploitation font une utilisation bien plus sophistiquée de la pagination que xv6 ; par exemple, xv6 ne fait pas de pagination depuis le disque, de fork avec copie à l'écriture, de mémoire partagée, d'allocation paresseuse des pages ni d'extension automatique de la pile. Le x86 supporte la traduction d'adresses en utilisant la segmentation (voir annexe B), mais xv6 n'utilise la segmentation que pour l'astuce courante d'implémentation de variables propres aux processeurs comme `proc`, qui sont à des adresses fixées mais ont des valeurs différentes pour chaque processeur (voir `seginit`). Les implémentations d'un stockage par processeur (ou par thread) sur des architectures non segmentées dédieraient un registre pour stocker un pointeur sur une zone donnée propre au processeur, mais le x86 possède si peu de registres généraux que le supplément d'effort requis par la segmentation en vaut la peine.

Xv6 associe le noyau à l'espace d'adressage de chaque processus utilisateur, mais il l'installe de sorte que la partie noyau de l'espace d'adressage soit inaccessible lorsque le processeur est en mode utilisateur. Cette organisation est pratique car après qu'un processus a basculé de l'espace utilisateur vers l'espace noyau, le noyau peut facilement accéder à la mémoire utilisateur en lisant directement les emplacements en mémoire. Il est toutefois probablement mieux, pour la sécurité, d'avoir une table des pages pour le noyau et d'utiliser cette table des pages lorsqu'on entre dans le noyau depuis le mode utilisateur, de sorte que les processus noyaux et utilisateurs soient davantage séparés les uns des autres. Une telle conception par exemple, aiderait à atténuer les canaux auxiliaires, exposés par la vulnérabilité Meltdown qui permet à un processus utilisateur de lire n'importe quelle partie de la mémoire du noyau.

Sur des machines avec beaucoup de mémoire, il pourrait être logique d'utiliser les « super-pages » de 4 mégaoctets du x86. L'utilisation de petites pages est logique lorsque la mémoire physique est restreinte pour permettre une granularité fine de l'allocation et de l'évincement des pages sur le disque. Par exemple, si un programme n'utilise que 8 Ko de mémoire, lui donner des pages physiques de 4 Mo serait du gaspillage. L'utilisation de pages plus grandes a du sens sur des machines avec beaucoup de RAM, et peut réduire les coûts liés à la manipulation des tables de pages. Xv6 n'utilise les super-pages qu'à un endroit : la table des pages initiale [1306]. L'initialisation du tableau modifie deux des 1024 PDE, aux indices 0 et 512 (`KERNBASE»PDXSHIFT`), laissant les autres à zéro. Xv6 active l'indicateur `PDE_PS` dans ces deux PDE pour les marquer comme étant des super-pages. Le noyau indique également à la MMU d'autoriser les super-pages en activant le bit `CR_PSE` (*Page Size Extension*) dans le registre `%cr4`.

Xv6 devrait déterminer la configuration actuelle de la RAM, au lieu de la supposer à 224 Mo. Sur le x86, il existe au minimum trois algorithmes classiques : le premier est de sonder l'espace d'adressage physique en cherchant des zones qui se comportent comme de la mémoire, conservant les valeurs qu'on y écrit ; le deuxième consiste à lire le nombre de kilo-octets de mémoire en dehors d'un emplacement connu de 16 bits dans la RAM non volatile du PC ; et le troisième est de chercher dans la mémoire du BIOS la table d'agencement de la mémoire faisant partie des tables multiprocesseurs. Lire le tableau de disposition de la mémoire est compliqué.

L'allocation mémoire était un sujet brûlant il y a longtemps, le problème de base étant un usage efficient d'une quantité de mémoire limitée et la préparation de futures requêtes inconnues ; cf. Knuth. Aujourd'hui, on se soucie plus de la vitesse que de l'optimisation mémoire. De plus, un noyau plus élaboré aurait probablement alloué plusieurs tailles différentes de petits blocs, plutôt que simplement des blocs de 4 096 octets (comme dans xv6) ; un vrai allocateur noyau devrait avoir à gérer de petites allocations comme des grandes.

2.10 Exercices

1. Renseignez-vous sur le fonctionnement de réels systèmes d'exploitation pour voir comment ils dimensionnent la mémoire.
2. Si xv6 n'avait pas utilisé les super-pages, quelle aurait été la bonne déclaration pour `entrypgdir` ?
3. Écrivez un programme utilisateur qui agrandit son espace d'adressage de 1 octet en appelant `sbrk(1)`. Exécutez le programme et examinez la table des pages du programme avant et après l'appel à `sbrk`. Combien d'espace le noyau a-t-il alloué ? Que contient le `pte` de la nouvelle mémoire ?
4. Modifiez xv6 afin que les pages pour le noyau soient partagées entre les processus, ce qui réduit la consommation mémoire.
5. Modifiez xv6 afin que lorsqu'un programme utilisateur déréférence un pointeur nul, il reçoive une erreur ; c'est-à-dire, modifiez xv6 afin que l'adresse virtuelle 0 ne soit plus traduite pour les programmes utilisateurs.
6. Les implémentations Unix de `exec` incluent traditionnellement un traitement spécial pour les scripts shell. Si le fichier à exécuter commence par « `#!` », alors la première ligne est considérée comme étant un programme à exécuter pour interpréter le fichier. Si par exemple, `exec` était appelé pour exécuter `myprog arg1` et que la première ligne de `myprog` était `#!/interp`, alors `exec` devrait exécuter `interp` avec la ligne de commande `/interp myprog arg1`. Implémentez la prise en charge de cette convention dans xv6.
7. Supprimez la vérification `if(ph.vaddr + ph.memsz < ph.vaddr)` dans `exec.c` et construisez un programme utilisateur qui exploite l'absence de cette vérification.
8. Modifiez xv6 afin que les processus utilisateurs s'exécutent avec une partie minimale de la traduction des adresses du noyau, et de sorte à ce que le noyau s'exécute avec sa propre table des pages qui n'inclut pas le processus utilisateur.
9. Comment amélioreriez-vous la disposition mémoire de xv6 si xv6 s'exécutait sur un processeur 64 bits ?

Chapitre 3

Trappes, interruptions et périphériques

Lorsqu'il exécute un processus, un processeur effectue la boucle classique : lire une instruction, avancer le compteur ordinal, exécuter l'instruction, et recommencer. Mais il y a des événements pour lesquels le contrôle d'un programme utilisateur doit être transféré au noyau au lieu d'exécuter l'instruction suivante. Ces évènements sont par exemple un périphérique signalant qu'il réclame de l'attention, un programme utilisateur faisant quelque chose d'illégal (par exemple faisant référence à une adresse virtuelle pour laquelle il n'existe pas d'entrée de la table des pages) ou un programme utilisateur demandant un service au noyau à l'aide d'un appel système. Traiter ces évènements engendre trois défis principaux : 1) le noyau doit s'arranger pour qu'un processeur bascule du mode utilisateur vers le mode noyau (puis inversement) ; 2) le noyau et les périphériques doivent coordonner leurs activités parallèles ; et 3) le noyau doit comprendre l'interface des périphériques. Résoudre ces trois défis nécessite une compréhension détaillée du matériel et une programmation soignée, et peut opacifier le code du noyau. Ce chapitre explique comment xv6 aborde ces trois défis.

3.1 Appels système, exceptions et interruptions

Le contrôle doit être transféré d'un programme utilisateur au noyau dans trois cas. Premièrement, un appel système : lorsqu'un programme utilisateur demande un service au système d'exploitation, comme nous l'avons vu à la fin du chapitre précédent. Deuxièmement, une *exception* : lorsqu'un programme effectue une action interdite. Les exemples d'actions interdites incluent la division par zéro, une tentative d'accès à la mémoire alors que l'entrée de la table des pages n'est pas présente, etc. Troisièmement, une *interruption* : lorsqu'un périphérique génère un signal pour réclamer l'attention du système d'exploitation. Par exemple, la puce d'horloge peut générer une interruption toutes les 100 ms pour permettre au noyau d'implémenter le partage des ressources, ou lorsque le contrôleur de disque a lu un bloc, il génère une interruption pour informer le système d'exploitation que le bloc est prêt à être récupéré.

Plutôt que laisser les processus gérer certaines interruptions, le noyau les traite toutes car, dans la plupart des cas, seul le noyau possède les privilèges et états [1] requis. Par exemple, dans le but de découper temporellement les processus en réponse aux interruptions de l'horloge, le noyau doit être impliqué, ne serait-ce que pour forcer les processus non coopératifs à libérer le processeur.

Pour tous ces cas, le système doit être conçu afin de respecter le déroulement suivant : le système doit sauvegarder les registres du processeur pour une reprise ultérieure transparente ; le

1. NdT : Le noyau connaît l'ensemble des requêtes en cours et en attente.

système doit être exécuté dans le noyau ; le système doit choisir un emplacement pour que le noyau puisse commencer à s'exécuter ; le noyau doit être capable de récupérer les informations à propos de l'évènement, comme par exemple les arguments de l'appel système ; tout doit être effectué de façon sécurisée ; le système doit maintenir l'isolation entre les processus utilisateurs et le noyau.

Pour atteindre cet objectif, le système d'exploitation doit connaître en détail la manière dont le matériel prend en compte les appels système, les exceptions et les interruptions. Dans la plupart des processeurs, ces trois évènements sont gérés par le même mécanisme matériel. Par exemple, sur le x86, un programme demande un appel système en générant une interruption par l'utilisation de l'instruction `int`. De façon similaire, les exceptions génèrent également une interruption. Ainsi, si le système d'exploitation sait gérer les interruptions, il peut gérer de même les appels système et les exceptions.

Le schéma de base est le suivant. Une interruption arrête la boucle normale du processeur et commence l'exécution d'une nouvelle séquence appelée *gestionnaire d'interruptions*. Avant de démarrer le gestionnaire d'interruptions, le processeur sauvegarde ses registres, afin que le système d'exploitation puisse les restaurer après avoir terminé le traitement de l'interruption. Un défi de la transition vers et depuis le gestionnaire d'interruptions est que le processeur doit basculer du mode utilisateur au mode noyau, puis inversement.

Un mot sur la terminologie : bien que le terme officiel de x86 soit exception, xv6 utilise principalement le terme *trappe* car il s'agit du terme employé par le PDP11/40 et est donc le terme Unix conventionnel. De plus, ce chapitre utilise indifféremment les termes trappe et interruption, mais il est important de se souvenir que les trappes sont provoquées par le processus courant s'exécutant sur le processeur (le processus fait par exemple un appel système et il en résulte une trappe) alors que les interruptions sont provoquées par les périphériques et peuvent ne pas avoir de rapport avec le processus en cours d'exécution. Par exemple, un disque peut générer une interruption lorsqu'il a fini de récupérer un bloc pour un processus, mais au moment de l'interruption, un autre processus pourrait être en cours d'exécution. Cette propriété des interruptions rend les raisonnements plus délicats que pour les trappes, car les interruptions arrivent en même temps que d'autres activités. Les interruptions comme les trappes reposent toutefois sur le même mécanisme pour transférer de façon sécurisée le contrôle entre modes utilisateur et noyau, ce dont nous allons discuter dans la suite.

3.2 Protection du x86

Le x86 dispose de 4 niveaux de protection, numérotés de 0 (privilèges maximum) à 3 (privilèges minimum). En pratique, la plupart des systèmes d'exploitation n'utilisent que deux niveaux : 0 et 3, qui sont donc appelés respectivement *mode noyau* et *mode utilisateur*. Le niveau de privilèges courant, avec lequel le x86 exécute les instructions, est indiqué par le champ CPL du registre `%cs`.

Sur le x86, les gestionnaires d'interruptions sont définis dans la table des descripteurs d'interruption ou IDT (*Interrupt Descriptor Table*). La table IDT a 256 entrées, chacune indiquant les valeurs de `%cs` et `%eip` à utiliser pour traiter l'interruption correspondante.

Pour effectuer un appel système sur le x86, un programme utilise l'instruction `int n` où `n` représente l'index dans la table IDT. L'instruction `int` effectue les étapes suivantes :

- chercher le $n^{ème}$ descripteur de IDT, où `n` est l'argument de `int` ;
- vérifier que CPL (dans `%cs`) $\leq$ DPL, où DPL représente le niveau de privilèges du descripteur ;
- sauvegarder `%esp` et `%ss` dans les registres internes du processeur, mais uniquement si

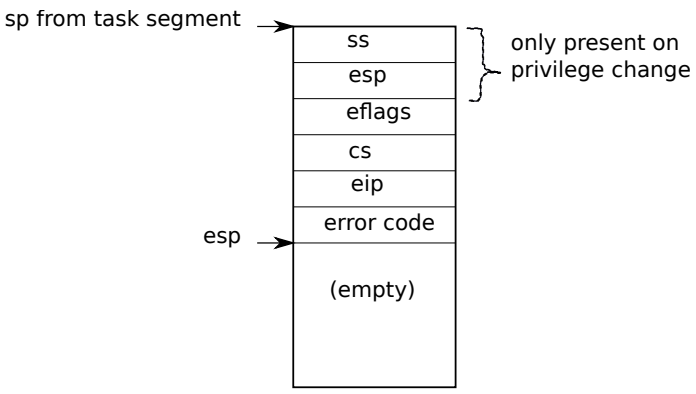

FIGURE 3.1 – Pile noyau après l'instruction `int`.

le niveau de privilèges du sélecteur de segment cible PL $<$ CPL ;
— charger `%ss` et `%esp` à partir du descripteur de segment de tâche ;
— empiler `%ss` ;
— empiler `%esp` ;
— empiler `%eflags` ;
— empiler `%cs` ;
— empiler `%eip` ;
— mettre à 0 le bit IF du registre `%eflags`, mais uniquement dans le cas d'une interruption ;
— mettre les valeurs issues du descripteur dans `%cs` et `%eip`.

L'instruction `int` est une instruction complexe, et on pourrait se demander si toutes ces actions sont nécessaires. Par exemple, la vérification CPL $\leq$ DPL permet au noyau d'interdire les appels à `int` sur des entrées inappropriées de la table IDT comme les routines d'interruption de périphériques. Pour qu'un programme utilisateur puisse exécuter `int`, le DPL de l'entrée de la table IDT doit valoir 3. Si le programme utilisateur n'a pas les privilèges appropriés, alors l'instruction `int` se soldera par un `int 13`, qui correspond à une erreur fatale [2]. De même, l'instruction `int` ne peut pas utiliser la pile utilisateur pour sauvegarder des valeurs, car le processus pourrait ne pas avoir de pointeur de pile valide ; à la place, le matériel utilise la pile spécifiée dans le segment de tâche, qui est défini par le noyau.

La figure 3.1 montre la pile à la fin de l'instruction `int` avec un changement de niveau de privilèges (le niveau des privilèges dans le descripteur est plus bas que le CPL). Si l'instruction `int` ne nécessitait pas de changement de niveau de privilèges, le x86 ne sauvegarderait pas `%ss` et `%esp`. Dans les deux cas, `%eip` pointe sur l'adresse spécifiée dans la table des descripteurs et l'instruction à cette adresse sera la prochaine instruction à exécuter et la première instruction du gestionnaire pour `int n`. C'est au système d'exploitation d'implémenter ces gestionnaires, et nous allons voir ci-après ce que fait xv6.

Un système d'exploitation peut utiliser l'instruction `iret` pour revenir d'une instruction `int`. `Iret` dépile les valeurs sauvegardées par `int` et continue l'exécution à partir de la valeur restaurée de `%eip`.

2. NdT : Le nom exact de l'erreur relevée par le processeur est *general protection fault*.

3.3 Code : Le premier appel système

Le chapitre 1 s'est terminé avec une demande d'appel système dans `initcode.S` (page 31). Revoyons cela [8414]. Le processus a empilé les arguments pour un appel à `exec` et stocké le numéro de l'appel système dans `%eax`. Ce numéro correspond à l'entrée dans le tableau des appels système, un tableau de pointeurs sur fonction [3672]. Il faut faire en sorte que l'instruction `int` fasse basculer le processeur du mode utilisateur au mode noyau, que le noyau fasse appel à la bonne fonction du noyau (c'est-à-dire `sys_exec`) et que le noyau puisse récupérer les arguments pour `sys_exec`. Les prochaines sections décrivent comment xv6 organise cela pour les appels système, puis nous verrons qu'il est possible de réutiliser le même code pour les interruptions et les exceptions.

3.4 Code : Gestionnaire de trappes (partie assembleur)

Xv6 doit configurer le x86 pour faire quelque chose de sensé lorsqu'il rencontre une instruction `int`, qui provoquera la génération d'une trappe par le processeur. Le x86 permet 256 interruptions différentes. Les interruptions entre 0 et 31 sont réservées aux exceptions des applications, comme les erreurs de division ou les tentatives d'accès à une adresse mémoire invalide. Xv6 lie les 32 interruptions matérielles à la plage 32-63 et utilise l'interruption 64 comme interruption des appels système.

La fonction `tvinit` [3367], appelée depuis `main`, initialise les 256 entrées de la table `idt`. L'interruption `i` est gérée par le code à l'adresse indiquée par `vectors[i]`. Chaque point d'entrée est différent, car x86 ne fournit pas le numéro de trappe au gestionnaire d'interruptions. Utiliser 256 gestionnaires différents est le seul moyen de distinguer les 256 cas.

La fonction `tvinit` gère `T_SYSCALL`, la trappe d'appel système, de façon spécifique : elle indique que la porte[3] est de type « trappe » en passant la valeur 1 comme deuxième argument. Les portes de type « trappe » ne mettent pas à 0 l'indicateur `IF`, permettant ainsi la prise en compte des interruptions durant le traitement de l'appel système.

Le noyau fixe également le privilège de la porte d'appel système à `DPL_USER`, ce qui permet à un programme utilisateur de générer des trappes à l'aide d'une instruction explicite `int`. Xv6 ne permet pas aux processus de générer d'autres interruptions (comme des interruptions de périphériques) avec `int` : s'ils essayent, ils rencontreront une erreur fatale, qui sera redirigée vers le vecteur 13.

Lors du changement du niveau de protection depuis le mode utilisateur vers le mode noyau, le noyau ne doit pas utiliser la pile du processus utilisateur puisqu'elle pourrait ne pas être valide. Le processus utilisateur pourrait être malveillant ou avoir un bogue, de sorte que le registre `%esp` utilisateur pourrait contenir une adresse ne faisant pas partie de la mémoire du processus utilisateur. Xv6 programme le x86 pour effectuer un changement de pile lors d'une trappe en configurant un descripteur de segment de tâche par lequel le matériel charge un sélecteur de segment de pile et une nouvelle valeur pour `%esp`. La fonction `switchuvm` [1860] sauvegarde l'adresse du sommet de la pile noyau du processus utilisateur dans le descripteur de segment de tâche.

Lorsqu'une trappe se produit, le processeur exécute les actions suivantes. Si le processeur était en mode utilisateur, il charge `%esp` et `%ss` depuis le descripteur de segment de tâche et empile les anciens `%ss` et `%esp` sur la nouvelle pile. Si le processeur était en mode noyau, rien de ceci

3. NdT : Dans la terminologie x86, le mot « porte » (*trap gate* ou simplement *gate*) est une entrée de la table des descripteurs d'interruption (IDT).

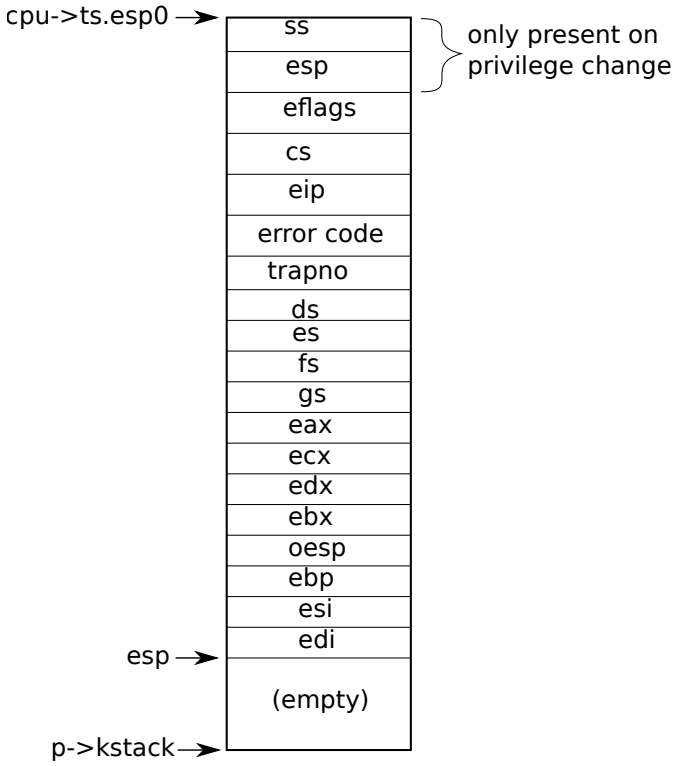

FIGURE 3.2 – Contexte de trappe sur la pile noyau.

ne se produit. Ensuite, le processeur empile les registres `%eflags`, `%cs` et `%eip`. Pour certaines trappes (comme par exemple un défaut de page), le processeur empile aussi un mot d'erreur. Le processeur charge ensuite `%eip` et `%cs` depuis la bonne entrée de la table IDT.

Xv6 utilise un script Perl [3250] pour générer les points d'entrée sur lesquels pointent les cases de la table IDT. Chaque point d'entrée empile un code d'erreur si ce n'est pas fait par le processeur, empile le numéro d'interruption puis continue à `alltraps`.

`Alltraps` [3304] continue de sauvegarder les registres processeurs : elle empile `%ds`, `%es`, `%fs`, `%gs`, et les registres à usage général [3305-3310]. Le résultat de cet effort est que la pile noyau contient maintenant une structure `struct trapframe` [0602] contenant les registres processeurs au moment de la trappe (voir figure 3.2). Le processeur empile `%ss`, `%esp`, `%eflags`, `%cs` et `%eip`. Le processeur ou le vecteur de trappe empile un numéro d'erreur et `alltraps` empile le reste. Le contexte de trappe contient toutes les informations nécessaires pour restaurer les registres processeurs du mode utilisateur lorsque le noyau retournera au processus courant, afin que le processeur puisse continuer l'exécution exactement là où il était lorsque la trappe a commencé. Souvenez-vous du chapitre 1 (page 28) où `userinit` construit un contexte de trappe « à la main » pour atteindre cet objectif (voir figure 1.4, page 28).

Dans le cas du premier appel système, la valeur de `%eip` sauvegardée[4] est l'adresse de l'instruction juste après `int`. `%cs` contient le sélecteur de segment de code. `%eflags` est le contenu du registre `%eflags` au moment de l'exécution de l'instruction `int`. Durant la sauvegarde des registres à usage général, `alltraps` sauvegarde aussi le registre `%eax` qui contient le numéro de l'appel système pour que le noyau puisse l'utiliser ultérieurement.

Maintenant que les registres processeurs du mode utilisateur sont sauvegardés, `alltraps` peut terminer la configuration du processeur pour exécuter le code C du noyau. Le processeur

4. NdT : Le `%eip` sauvegardé par l'instruction `int` dans `initcode.S`, juste avant l'étiquette `exit`.

avait initialisé les sélecteurs `%cs` et `%ss` avant d'entrer dans le gestionnaire ; `alltraps`, quant à elle, initialise `%ds` et `%es` [3313-3315].

Une fois les segments correctement initialisés, `alltraps` peut appeler `trap`, le gestionnaire de trappe en C. `Alltraps` empile `%esp`, qui pointe sur le contexte de trappe fraîchement construit, pour être l'argument de `trap` [3318]. Puis elle appelle `trap` [3319]. Après le retour de `trap`, `alltraps` dépile l'argument par addition au pointeur de pile [3320], puis commence à exécuter le code à l'étiquette `trapret`. Nous avons déjà suivi ce que faisait ce code dans le chapitre 1 (page 28) lorsque le premier processus utilisateur l'a exécuté pour sortir vers l'espace utilisateur. La même séquence se produit ici : dépiler le contexte de trappe restaure les registres du mode utilisateur, puis `iret` fait continuer l'exécution dans l'espace utilisateur.

L'explication était jusqu'ici focalisée sur le traitement des trappes alors que le processeur était en mode utilisateur, mais les trappes peuvent également survenir lorsque le noyau s'exécute. Dans ce cas, le matériel ne change pas la pile ni ne sauvegarde le pointeur de pile et le sélecteur de segment de pile ; à part cela, le processeur effectue les mêmes étapes que pour une trappe depuis le mode utilisateur, et le même code de gestion de trappe de xv6 s'exécute. Lorsque `iret` restaurera un `%cs` en mode noyau, le processeur continuera son exécution en mode noyau.

3.5 Code : Gestionnaire de trappes (partie C)

Nous avons vu dans la dernière section que chaque gestionnaire initialise un contexte de trappe puis appelle la fonction C `trap`. `Trap` [3401] se base sur le numéro de trappe matériel `tf->trapno` pour comprendre pourquoi il a été appelé ainsi que ce qui doit être fait. Si la trappe est `T_SYSCALL`, `trap` appelle le gestionnaire d'appels système `syscall`. Nous reverrons le test `proc->killed` dans le chapitre 5.

Après avoir vérifié s'il s'agit d'un appel système, `trap` vérifie les interruptions matérielles (dont nous parlerons ci-après). En plus des périphériques matériels connus, une trappe peut être provoquée par une fausse interruption, une interruption matérielle non désirée.

Si la trappe n'est ni un appel système, ni un périphérique réclamant de l'attention, `trap` suppose qu'elle a été provoquée par un mauvais comportement (une division par zéro par exemple) par le code exécuté au moment de la trappe. Si le code qui a provoqué la trappe était un programme utilisateur, xv6 affiche les détails sur ce programme, puis met `proc->killed` à 1 pour se souvenir de nettoyer le processus utilisateur. Nous étudierons comment xv6 effectue ce nettoyage dans le chapitre 5.

Si c'était le noyau qui était en cours d'exécution, il doit y avoir un bogue du noyau : `trap` affiche les détails à propos de cet événement inattendu, puis appelle `panic`.

3.6 Code : Appels système

Pour les appels système, `trap` fait appel à `syscall` [3701]. `Syscall` récupère le numéro de l'appel système depuis le contexte de trappe, qui contient le registre sauvegardé `%eax`, et parcourt la table des appels système. Pour le premier appel système, `%eax` contenait la valeur `SYS_exec` [3507], et `syscall` appelait la `SYS_exec`$^{\text{ème}}$ entrée du tableau des appels système, ce qui correspond à faire appel à `sys_exec`.

`Syscall` stocke la valeur de retour de la fonction de l'appel système dans `%eax`. Lors du retour de trappe dans l'espace utilisateur, elle va charger les valeurs depuis `cp->tf` dans les registres du processeur. Ainsi, lorsque `exec` retourne, il va renvoyer la valeur que lui a renvoyé le ges-

tionnaire de l'appel système [3708]. Par convention, les appels système renvoient des nombres négatifs pour signaler des erreurs et des nombres positifs en cas de succès. Si le numéro d'appel système est invalide, `syscall` affiche une erreur puis renvoie -1.

Les chapitres suivants vont étudier l'implémentation de certains appels système particuliers. Ce chapitre se concentre sur les mécanismes des appels système. Il reste un mécanisme que nous n'avons pas étudié : récupérer les arguments des appels système. Les fonctions auxiliaires `argint`, `argptr`, `argstr` et `ardfd` récupèrent le n^{ème} argument d'un appel système comme un entier, un pointeur, une chaîne de caractères ou un descripteur de fichiers. `Argint` utilise le registre `%esp` de l'espace utilisateur pour localiser le n^{ème} argument : `%esp` pointe sur l'adresse de retour de la petite fonction de bibliothèque qui utilise `int` pour implémenter l'appel système. Les arguments sont juste au-dessus, à `esp+4`. Ainsi, le n^{ème} argument se trouve à `%esp+4+4*n`.

`Argint` appelle `fetchint` pour lire la valeur à cette adresse dans la mémoire utilisateur et l'écrire dans `*ip`. `Fetchint` peut simplement convertir l'adresse en un pointeur car l'utilisateur et le noyau partagent la même table des pages, mais le noyau doit s'assurer que le pointeur pointe vers la partie utilisateur de l'espace d'adressage. Le noyau a configuré la MMU pour s'assurer que le processus ne peut pas accéder à de la mémoire en dehors de sa mémoire locale privée : si un programme utilisateur tente de lire ou d'écrire de la mémoire à une adresse à `p->sz` ou plus haut, le processeur va provoquer une trappe de segmentation, et `trap` va terminer le processus, comme vu précédemment. Le noyau peut quant à lui accéder à toute adresse que l'utilisateur pourrait avoir indiquée, il doit donc explicitement vérifier que l'adresse soit inférieure à `p->sz`.

`Argptr` récupère le n^{ème} argument de l'appel système et vérifie que cet argument soit bien un pointeur valide dans l'espace utilisateur. Il est à noter que deux vérifications sont effectuées durant l'appel à `argptr`. Premièrement, le pointeur sur la pile utilisateur est vérifié pendant la récupération de l'argument. Puis l'argument, lui-même un pointeur utilisateur, est vérifié.

`Argstr` interprète le n^{ème} argument comme étant un pointeur. Elle s'assure que le pointeur pointe sur une chaîne de caractères se terminant par un octet nul et que l'intégralité de cette chaîne est située en dessous de la fin de la partie utilisateur de l'espace d'adressage.

Finalement, `argfd` [6071] utilise `argint` pour récupérer un numéro de descripteur de fichier, vérifie s'il s'agit d'un descripteur de fichier valide et renvoie la structure `struct file` correspondante.

Les implémentations des appels système (par exemple `sysproc.c` et `sysfile.c`) sont généralement des surcouches : elles décodent les arguments en utilisant `argint`, `argptr` et `argstr` puis appellent l'implémentation réelle. Dans le chapitre 2, `sys_exec` utilise ces fonctions pour accéder à ses arguments.

3.7 Code : Interruptions

Les périphériques sur la carte mère peuvent générer des interruptions, et xv6 doit configurer le matériel pour gérer ces interruptions. Les périphériques provoquent généralement des interruptions afin de signaler au noyau qu'un certain évènement matériel s'est produit, comme la fin d'une entrée/sortie. Les interruptions sont théoriquement optionnelles, dans le sens où le noyau pourrait, à la place, interroger périodiquement (ou *poller* [5]) les périphériques pour vérifier si de nouveaux évènements ont eu lieu. Les interruptions sont préférables au polling

5. NdT : Nous emploierons dans la suite les barbarismes issus du verbe anglais *to poll* pour désigner cette vérification périodique, faute de terme communément admis.

lorsque les évènements sont relativement rares, car le polling gaspille du temps processeur. La gestion des interruptions partage en partie du code existant pour les appels système et les exceptions.

Les interruptions sont similaires aux appels système, si ce n'est que les périphériques peuvent les générer à n'importe quel moment. Il y a un composant sur la carte mère pour alerter le processeur lorsqu'un périphérique réclame de l'attention (par exemple lorsque l'utilisateur a tapé un caractère sur le clavier). Il faut programmer les périphériques pour qu'ils génèrent des interruptions et faire en sorte que le processeur les reçoive.

Regardons maintenant le périphérique horloge et ses interruptions. Nous aimerions que l'horloge génère une interruption, disons 100 fois par seconde, de sorte que le noyau puisse comptabiliser le temps qui passe et donner ainsi des tranches de temps aux différents processus en cours d'exécution. Le choix de 100 fois par seconde donne une performance interactive décente tout en évitant de surcharger le processeur avec des interruptions à traiter.

Comme le processeur x86 lui-même, les cartes mères des PC ont évolué, et la façon dont les interruptions sont acheminées a également évolué. Les premières cartes avaient un simple contrôleur d'interruptions programmable (appelé PIC). Avec l'avènement des cartes multi-processeurs, une nouvelle méthode de gestion des interruptions était nécessaire, car chaque processeur doit disposer de son contrôleur d'interruptions pour traiter les interruptions qui lui sont envoyées et il doit y avoir une méthode pour acheminer les interruptions vers les processeurs. Cette méthode est composée de 2 parties : une située dans le système des entrées/sorties (appelée IO APIC[6], `ioapic.c`), et l'autre rattachée à chaque processeur (appelée APIC local, `lapic.c`). Xv6 est conçu pour une carte multi-processeurs : il ignore les interruptions du PIC et configure l'IOAPIC et les APIC locaux.

L'IOAPIC contient un tableau et le processeur peut en programmer les entrées grâce à l'adressage des entrées/sorties en mémoire. Pendant l'initialisation, xv6 programme les portes : `tvinit` [3367] initialise l'entrée 0 de la table IDT et la lie à `vector0`, et ainsi de suite. Les périphériques spécifiques permettent des interruptions particulières et spécifient à quel processeur elles doivent être envoyées. Par exemple, xv6 envoie les interruptions du clavier au processeur 0 [8274], et les interruptions des disques vers le plus haut processeur du système, comme nous le verrons ultérieurement.

La puce horloge se situe à l'intérieur du LAPIC afin que chaque processeur puisse recevoir des interruptions de l'horloge de façon indépendante. Xv6 les active dans `lapicinit` [7408]. La ligne clef est celle qui programme l'horloge [7421]. Cette ligne dit au LAPIC de générer périodiquement une interruption vers IRQ_TIMER, qui correspond à IRQ 0. La ligne 7451 autorise les interruptions sur le LAPIC de chaque processeur, ce qui lui permettra de recevoir les interruptions.

Un processeur peut contrôler s'il veut recevoir les interruptions grâce à l'indicateur `IF` du registre `%eflags`. L'instruction `cli` désactive les interruptions sur le processeur en effaçant `IF`, et l'instruction `sti` les active. Xv6 désactive les interruptions durant le démarrage du processeur principal [9112] et des autres processeurs [1124]. L'ordonnanceur xv6 associé à chaque processeur active les interruptions [2766]. Pour empêcher que certaines parties de code soient interrompues, xv6 désactive temporairement les interruptions (voir par exemple `switchuvm` [1860]).

L'horloge envoie des interruptions via le vecteur 32 (que xv6 a choisi pour gérer IRQ 0), ce que xv6 configure dans `idtinit` [1255]. La seule différence entre vecteur 32 et vecteur 64 (le vecteur des appels système) est que le vecteur 32 est une porte d'interruption et non de

6. NdT : « APIC » est l'acronyme de *Advanced Programmable Interrupt Controller*. Il s'agit d'un composant matériel du PC.

trappe. Les portes d'interruption initialisent `IF` à 0 de sorte que le processeur interrompu ne reçoive plus d'interruption pendant le traitement de l'interruption courante. À partir d'ici et jusqu'à `trap`, les interruptions suivent le même chemin de code que les appels système et les exceptions, construisant un contexte de trappe.

Pour une interruption de l'horloge, `trap` n'effectue que deux actions : la première est l'incrémentation de la variable `ticks` [3417] et l'appel de `wakeup`. La deuxième action, comme nous le verrons dans le chapitre 5, peut provoquer le retour de l'interruption dans un processus différent.

3.8 Pilotes

Un *pilote* est le code au sein d'un système d'exploitation pour gérer un périphérique particulier : il demande au périphérique matériel d'effectuer des opérations, le configure pour générer des interruptions lorsqu'il a terminé les opérations et traite les interruptions résultantes. Il peut être délicat d'écrire des pilotes, car ces derniers s'exécutent de manière concurrente avec les périphériques. De plus, le pilote doit connaître l'interface des périphériques (par exemple quel port d'entrée/sortie fait quoi) et cette interface peut être complexe et peu documentée.

Le pilote de disque est un bon exemple. Il copie des données depuis et vers le disque. Les disques présentent habituellement les données comme une séquence numérotée de *secteurs* de 512 octets : le secteur 0 comprend les 512 premiers octets, le secteur 1 les 512 suivants, etc. La taille des blocs qu'utilise un système d'exploitation pour son système de fichiers peut être différente de la taille des secteurs du disque, mais en général la taille des blocs est un multiple de celle des secteurs. Les blocs de xv6 ont la même taille que les secteurs disques. Pour représenter un bloc, xv6 utilise une structure `struct buf` [3850]. Les données dans cette structure sont souvent désynchronisées par rapport au disque : elles peuvent ne pas encore avoir été lues depuis le disque (le disque est en train de les lire, mais il n'a pas encore renvoyé le contenu des secteurs), ou elles peuvent avoir été mises à jour, mais pas encore écrites. Le pilote doit s'assurer que le reste de xv6 ne soit pas troublé lorsque la structure est désynchronisée par rapport au disque.

3.9 Code : Pilote de disque

Le périphérique IDE fournit un accès aux disques connectés au contrôleur IDE standard du PC. IDE est maintenant démodé par rapport à SCSI et SATA, mais l'interface IDE est simple et permet de se concentrer sur la structure globale d'un pilote, plutôt que sur les détails d'un composant matériel particulier.

Xv6 représente les blocs de son système de fichiers en utilisant des `struct buf` [3850]. `BSIZE` [4055] est identique à la taille des secteurs IDE et ainsi, chaque tampon représente le contenu d'un secteur sur un disque particulier. Les champs `dev` et `sector` indiquent le numéro de périphérique et de secteur et le champ `data` est une copie en mémoire du secteur disque. Bien que le système de fichier de xv6 ait choisi une valeur de `BSIZE` identique à la taille d'un secteur IDE, le pilote peut gérer un `BSIZE` qui soit un multiple de la taille des secteurs. Les systèmes d'exploitation utilisent souvent des blocs plus grands que 512 octets pour obtenir un meilleur débit avec les disques.

Le champ `flags` donne les relations entre la mémoire et le disque : l'indicateur `B_VALID` signifie que `data` a été lu, et l'indicateur `B_DIRTY` signifie que `data` doit être écrit sur le disque.

Le noyau initialise le pilote de disque lors du démarrage, en appelant `ideinit` [4251] de-

puis `main` [1232]. `Ideinit` appelle `ioapicenable` pour autoriser les interruptions `IDE_IRQ` [4256]. L'appel à `ioapicenable` autorise les interruptions uniquement sur le dernier processeur (`ncpu-1`) : sur un système à deux processeurs, c'est le processeur 1 qui gère les interruptions des disques.

Ensuite, la fonction `ideinit` interroge les disques. Elle commence en appelant `idewait` [4257] pour attendre que le disque soit prêt à accepter des commandes. Une carte mère PC présente les bits d'état du disque sur le port d'entrée/sortie `0x1F7`. `Idewait` [4238] interroge périodiquement ces bits d'états jusqu'à ce que le bit d'activité (`IDE_BUSY`) soit nul et que le bit prêt (`IDE_DRDY`) soit à 1.

Maintenant que le contrôleur de disque est prêt, `ideinit` peut déterminer le nombre de disques présents [7]. Il suppose que le disque 0 est présent, puisque le chargeur d'amorçage ainsi que le noyau ont tous deux été chargés depuis ce disque 0, mais il doit vérifier la présence du disque 1. Il écrit sur le port d'entrée/sortie `0x1F6` pour sélectionner le disque 1, puis attend un peu que le bit d'état montre que le disque est prêt [4259-4266]. S'il ne l'est pas, `ideinit` suppose que le disque est absent.

Après `ideinit`, le disque n'est plus utilisé jusqu'à ce que le *buffer-cache* [8] appelle `iderw`, qui met à jour un tampon verrouillé de la façon spécifiée par les indicateurs : si `B_DIRTY` est activé, `iderw` écrit le tampon sur le disque ; si `B_VALID` n'est pas activé, `iderw` lit le tampon depuis le disque.

Les accès disques prennent en général plusieurs millisecondes, soit un temps très long pour un processeur. Le chargeur d'amorçage émet des commandes de lecture de disque et lit en boucle les bits d'état jusqu'à ce que les données soient prêtes (voir annexe B). Ce *polling* ou *attente active* est acceptable dans un chargeur d'amorçage qui n'a rien de mieux à faire. En revanche, dans un système d'exploitation, il est plus efficace de laisser un autre processus s'exécuter sur le processeur et s'arranger pour recevoir une interruption lorsque le disque aura terminé ses opérations. `Iderw` opte pour cette seconde approche, en conservant la liste des requêtes disques en attente dans une file et en utilisant les interruptions pour déterminer le moment où se termine chaque requête. Bien que `iderw` conserve une file de requêtes, le contrôleur disque IDE ne peut, lui, s'occuper que d'une opération à la fois. Le pilote de disque conserve l'invariant qu'il a envoyé le tampon situé en tête de la file vers le disque ; les autres attendent simplement leur tour.

`Iderw` [4354] ajoute le tampon `b` en queue de la file [4367-4371]. Si le tampon est en tête de la file, `iderw` doit l'envoyer vers le disque en appelant `idestart` [4373-4375] ; autrement le tampon sera traité une fois que les tampons avant lui auront été pris en charge.

`Idestart` [4274] démarre soit une lecture, soit une écriture (en fonction des indicateurs) pour le périphérique et le secteur du tampon. Si l'opération est une écriture, `idestart` doit fournir les données [4296]. `Idestart` copie les données dans un tampon du contrôleur de disque en utilisant l'instruction `outsl` ; utiliser des instructions du processeur pour copier les données depuis ou vers le périphérique est appelé « entrées/sorties programmées ». Finalement, le disque émettra une interruption pour signaler que les données ont été écrites sur le disque. Si l'opération est une lecture, l'interruption signalera que les données sont prêtes, et le gestionnaire pourra les lire. Il est à noter que `idestart` possède une connaissance détaillée du périphérique IDE, et qu'il écrit les bonnes valeurs sur les bons ports. Si n'importe lequel de ces `outb` était faux, le périphérique IDE ferait quelque chose de différent de ce qui est désiré. Connaître tous ces détails et les utiliser convenablement est l'une des raisons qui rend l'écriture des pilotes de périphériques difficile.

7. NdT : Avec le contrôleur IDE, il ne peut y avoir au maximum que deux disques sur la carte mère.
8. NdT : Comme évoqué dans le chapitre 1, nous préférons utiliser le terme original non traduit.

Ayant ajouté la requête à la file et l'ayant démarrée si nécessaire, `iderw` doit attendre le résultat. Comme évoqué ci-dessus, le *polling* ne permet pas une utilisation efficace du processeur. À la place, `iderw` libère le processeur pour les autres processus en utilisant `sleep`, attendant que le gestionnaire d'interruption modifie les indicateurs du tampon pour signaler que l'opération est terminée [4378-4379]. Pendant que ce processus est endormi, xv6 ordonnancera d'autres processus pour garder le processeur occupé.

Finalement, le disque terminera l'opération et génèrera une interruption. `Trap` appellera `ideintr` pour la gérer [3424]. `Ideintr` [4304] consulte le premier tampon de la file pour déterminer quelle opération était en cours. Si le tampon était en cours de lecture et que le contrôleur de disque a des données en attente, `ideintr` lit ces données depuis le tampon du contrôleur de disque vers la mémoire avec `insl` [4317-4319]. Maintenant, le tampon est prêt : `ideintr` active `B_VALID`, désactive `B_DIRTY` et réveille tous les processus en attente du tampon [4321-4324]. Enfin, `ideintr` doit traiter le prochain tampon en attente sur le disque [4326-4328].

3.10 Le monde réel

Supporter tous les périphériques d'une carte mère de PC dans toute sa splendeur est un travail considérable, car il existe beaucoup de périphériques, ils ont beaucoup de fonctionnalités et les protocoles entre périphériques et pilotes peuvent être complexes. Dans nombre de systèmes d'exploitation, l'ensemble des pilotes représente plus de lignes de code que le cœur du noyau.

Les pilotes de périphériques réels sont bien plus complexes que le pilote de disque de ce chapitre, mais l'idée de base reste la même : les périphériques sont généralement plus lents que le processeur, aussi le matériel utilise les interruptions pour informer le système d'exploitation d'un changement de statut. Les contrôleurs de disques modernes acceptent généralement un lot de requêtes simultanées et vont jusqu'à réordonnancer ces requêtes eux-mêmes pour une utilisation plus efficace du bras de disque. Lorsque les disques étaient plus simples, les systèmes d'exploitation réordonnançaient souvent eux-mêmes la file.

Beaucoup de systèmes d'exploitation possèdent des pilotes pour SSD (*Solid-State Disks*) car ils fournissent un accès plus rapide aux données. Mais, bien qu'un SSD fonctionne complètement différemment d'un disque mécanique traditionnel, ces deux types de périphériques fournissent une interface basée sur des blocs, et lire/écrire des blocs sur un SSD est toujours plus coûteux que de lire/écrire dans la RAM.

Les autres matériels sont étonnamment similaires aux disques : les tampons des périphériques réseaux contiennent des paquets, les tampons des périphériques audio contiennent des extraits sonores, les tampons des cartes graphiques contiennent des données vidéos et des séquences de commandes. Les périphériques à bande passante élevée – disques, cartes graphiques et cartes réseaux – utilisent souvent l'accès direct à la mémoire ou DMA (*Direct Acces Memory*) au lieu des entrées/sorties programmées (`insl`, `outsl`). Le DMA permet aux périphériques d'accéder directement à la mémoire physique. Le pilote donne au périphérique l'adresse physique des données du tampon et le périphérique copie directement vers ou depuis la mémoire principale, générant une interruption lorsque la copie est terminée. Le DMA est plus rapide et plus efficace que les entrées/sorties programmées et est moins coûteux pour les caches mémoires du processeur.

Certains pilotes échangent dynamiquement entre *polling* et interruptions, car utiliser des interruptions peut être coûteux, mais utiliser du *polling* peut engendrer un délai jusqu'à ce que le pilote traite un évènement. Par exemple, un pilote réseau qui reçoit une rafale de paquets peut passer des interruptions au *polling* puisqu'il sait que d'autres paquets doivent être immédiatement traités et qu'il est moins coûteux de les prendre en compte en utilisant le *polling*. Une

fois qu'il n'y aura plus de paquets à traiter, le pilote peut revenir aux interruptions, pour être immédiatement alerté lors de l'arrivée ultérieure d'un nouveau paquet.

Le pilote IDE route statiquement les interruptions vers un processeur particulier. Certains pilotes configurent l'IOAPIC pour router les interruptions vers plusieurs processeurs pour répartir le traitement des paquets. Par exemple, un pilote réseau pourrait s'arranger pour envoyer les interruptions pour les paquets d'une connexion réseau au processeur qui gère cette connexion, pendant que les interruptions des paquets d'une autre connexion sont envoyées à un autre processeur. Ce routage peut devenir assez sophistiqué ; par exemple, si certaines connexions réseau sont de courte durée tandis que d'autres sont de longue durée et que le système d'exploitation veut garder tous les processeurs actifs pour atteindre un débit élevé.

Si un programme lit un fichier, les données pour ce fichier sont copiées deux fois. Premièrement, elles sont copiées depuis le disque vers la mémoire du noyau par le pilote puis, plus tard, elles sont copiées depuis l'espace noyau vers l'espace utilisateur par l'appel système `read`. Si le programme envoie ensuite les données à travers le réseau, elles seront copiées deux fois de plus : de l'espace utilisateur vers l'espace noyau et de l'espace noyau vers le périphérique réseau. Pour aider les applications pour lesquelles l'efficacité est importante (par exemple servir des images populaires sur le net), les systèmes d'exploitation utilisent des chemins spéciaux dans le code pour éviter les copies. Par exemple, dans les systèmes d'exploitation réels, les tampons ont généralement la même taille que les pages, de sorte que les copies en lecture seule puissent être liées dans l'espace d'adressage d'un processus en utilisant la MMU, sans aucune copie.

3.11 Exercices

1. Mettez un point d'arrêt à la première instruction de `syscall` pour capturer le tout premier appel système (par exemple, `br syscall`). Quelles valeurs sont sur la pile à ce moment ? Expliquez le résultat x/37x $esp à ce poit d'arrêt avec chaque valeur étiquetée de ce qu'elle est (par exemple, `%ebp` pour trappe, contexte de trappe `%eip`, espace de travail...).
2. Ajoutez un nouvel appel système permettant d'obtenir l'heure UTC actuelle et de la renvoyer à un programme utilisateur. Vous voudrez peut-être utiliser la fonction `cmostime` [7552], pour lire l'horloge temps réel. Le fichier `date.h` contient la définition de `struct rtcdate` [0950], que vous fournirez en argument de type pointer à `cmostime`.
3. Écrivez un pilote de disque qui supporte le standard SATA (recherchez SATA sur le Web). Contrairement à IDE, SATA n'est pas obsolète. Utilisez la commande *tagged command queuing* de SATA pour émettre plusieurs commandes au disque, de sorte que le disque puisse les réordonnancer lui-même pour obtenir de meilleures performances.
4. Ajoutez un pilote simple pour une carte Ethernet.

Chapitre 4

Verrouillage

Xv6 est adapté à des multiprocesseurs : des ordinateurs avec plusieurs processeurs tournant indépendamment. Ces différents processeurs partagent la RAM physique et xv6 exploite ce partage pour maintenir des structures de données que tous les processeurs lisent et écrivent. Ce partage engendre le risque qu'un processeur lise une structure de données pendant qu'un autre processeur n'a pas complètement terminé de la modifier, ou même que plusieurs processeurs modifient la même structure de données simultanément ; sans une conception rigoureuse, de tels accès parallèles sont susceptibles de conduire à des résultats incorrects ou une structure de données incohérente. Même sur une machine avec un seul processeur, une routine d'interruption qui utilise les mêmes données que du code interruptible pourrait altérer les données si l'interruption survenait au moment crucial.

Tout code qui accède simultanément à des données partagées doit avoir une stratégie pour préserver l'exactitude malgré la concurrence. Celle-ci peut provenir de l'accès par de multiples cœurs de processeurs, par de multiples threads, ou par du code d'interruption. Xv6 met en œuvre quelques stratégies simples pour contrôler la concurrence ; une plus grande sophistication est possible. Ce chapitre se concentre sur l'une des stratégies abondamment utilisée dans xv6 et dans beaucoup d'autres systèmes : le *verrouillage*.

Un verrou fournit l'exclusion mutuelle, en faisant en sorte qu'un seul processeur à un instant donné puisse obtenir [1] le verrou. Si un verrou est associé avec chaque donnée partagée, et que le code obtient toujours le verrou associé quand il utilise cette donnée, alors on peut être sûr que la donnée est utilisée par un seul processeur à la fois. Dans cette situation, on dit que le verrou protège la donnée.

Le reste de ce chapitre explique pourquoi xv6 a besoin de verrous, comment il les implémente et comment il les utilise. Le point clé, lorsque vous lisez du code de xv6, est de se demander si un autre processeur (ou une interruption) pourrait modifier le comportement attendu du code en modifiant les données (ou les ressources matérielles) dont il dépend. Vous devez garder en tête qu'une seule instruction C peut correspondre à plusieurs instructions machines, et que donc un autre processeur ou une interruption pourrait survenir au milieu d'une instruction C. Vous ne pouvez pas supposer que des lignes de code sur le papier sont exécutées de manière atomique. La concurrence rend les raisonnements sur l'exactitude beaucoup plus difficiles.

1. NdT : Normalement, un verrou physique sur une porte s'ouvre ou se ferme. Dans le contexte de la concurrence, on *possède* ou on *libère* le verrou.

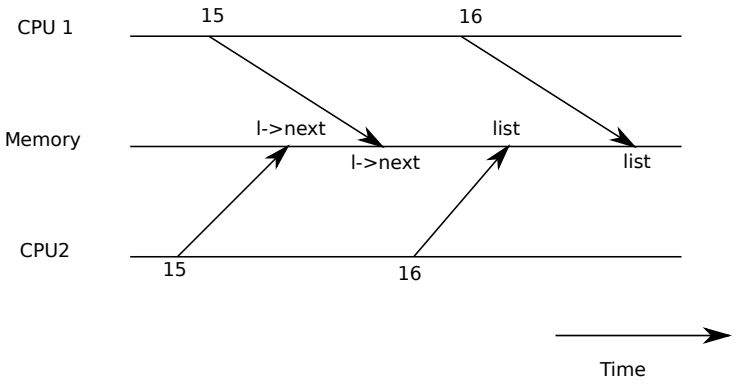

FIGURE 4.1 – Exemple de condition de concurrence.

4.1 Conditions de concurrence

Prenons un exemple pour expliquer pourquoi nous avons besoin de verrous : considérons plusieurs processeurs partageant un disque unique, tel qu'un disque IDE dans xv6. Le pilote du disque maintient une liste chaînée des requêtes disques en attente (4226) et les processeurs peuvent en ajouter de nouvelles en même temps (4354). S'il n'y avait pas de requêtes concurrentes, il serait possible d'implémenter la liste chaînée comme suit :

```
 1   struct list {
 2     int data;
 3     struct list *next;
 4   };
 5
 6   struct list *list = 0;
 7
 8   void
 9   insert(int data)
10   {
11     struct list *l;
12
13     l = malloc(sizeof *l);
14     l->data = data;
15     l->next = list;
16     list = l;
17   }
```

Cette implémentation est correcte si elle est exécutée en isolation. Cependant, le code n'est pas correct si plus d'une copie l'exécute en même temps. Si deux processeurs exécutent `insert` au même moment, il pourrait arriver qu'ils exécutent tous les deux la ligne 15 avant que l'un des deux exécute la ligne 16 (voir figure 4.1). Si cela arrive, il y aura alors deux éléments de la liste pour lesquels `next` vaut la précédente valeur de `list`. Quand les deux affectations à `list` à la ligne 16 seront effectuées, la deuxième aura écrasé la première ; l'élément de la liste utilisé dans la première affectation sera perdu.

La mise à jour perdue en ligne 16 est un exemple de *condition de concurrence* [2]. Une condition de concurrence est une situation dans laquelle un emplacement mémoire est accédé concurremment, et au moins un des accès est une écriture. Une telle condition est souvent le signe d'un bogue, soit d'une mise à jour perdue (si les accès sont des écritures), soit d'une lecture d'une structure de données alors que la mise à jour de celle-ci n'est pas terminée. Le résultat de cette

2. NdT : Le terme original est *race condition*.

condition dépend du timing précis des deux processeurs impliqués et de comment les opérations en mémoire sont ordonnées par le système mémoire, ce qui peut rendre les erreurs dues à des conditions de concurrence difficiles à reproduire. Par exemple, le fait d'ajouter des instructions d'affichage pendant le déboguage de `insert` peut suffisamment changer le timing de l'exécution pour faire disparaître le problème.

La méthode classique pour éviter les conditions de concurrence est d'utiliser un verrou. Les verrous garantissent l'*exclusion mutuelle*, de telle façon qu'un seul processeur à la fois puisse exécuter `insert`; ceci rend le scénario décrit ci-dessus impossible. La version correctement verrouillée du code précédent ajoute juste quelques lignes (non numérotées) :

```
 6   struct list *list = 0;
     struct lock listlock;
 7
 8   void
 9   insert(int data)
10   {
11     struct list *l;
12     l = malloc(sizeof *l);
13     l->data = data;
14
       acquire(&listlock);
15     l->next = list;
16     list = l;
       release(&listlock);
17   }
```

La suite d'instructions entre `acquire` et `release` est souvent appelée une *section critique*, et le verrou protège `list`.

Quand on dit qu'un verrou protège une donnée, nous voulons vraiment dire que le verrou protège un ensemble d'invariants qui s'appliquent à la donnée. Les invariants sont des propriétés d'une structure de données qui sont maintenues à travers des opérations. Typiquement, le comportement correct d'une opération dépend du fait que les invariants sont vérifiés lorsque l'opération débute. L'opération peut temporairement violer les invariants, mais doit les rétablir avant de terminer. Par exemple, dans le cas de la liste chaînée, l'invariant est « `list` pointe sur le premier élément de la liste et le `next` de chaque élément pointe sur l'élément suivant ». L'implémentation de `insert` viole temporairement cet invariant : à la ligne 15, `l` pointe sur l'élément suivant, mais `list` ne pointe pas encore sur `l` (ce qui est rétabli à la ligne 16). La condition de concurrence que nous avons examinée ci-dessus arrive parce qu'un second processeur a exécuté du code qui dépendait de l'invariant de liste, alors que nous l'avions (temporairement) violé. Une utilisation correcte d'un verrou garantit qu'un seul processeur à la fois peut opérer sur la structure de données dans la section critique, de telle façon qu'aucun autre processeur ne puisse exécuter d'opération sur cette structure de données tant que les invariants ne sont pas vérifiés.

On peut assimiler l'utilisation des verrous à une *sérialisation* des sections critiques concurrentes, de telle sorte qu'elles soient exécutées une seule à la fois, et donc en préservant les invariants (en supposant qu'ils soient corrects en isolation). On peut aussi voir les sections critiques comme étant atomiques les unes par rapport aux autres, de telle façon qu'une section critique qui obtient le verrou ultérieurement verra uniquement l'ensemble complet des modifications effectuées par les sections critiques antérieures, et ne verra jamais les mises à jour incomplètes.

Notez qu'il serait également correct de remonter `acquire` plus haut dans `insert`. Par exemple, il est acceptable de remonter l'appel à `acquire` jusqu'à avant la ligne 12. Cela peut réduire le

parallélisme car les appels à `malloc` sont également sérialisés. La section 4.3 (page 59) donne des indications pour savoir où insérer des appels à `acquire` et `release`.

4.2 Code : Verrous

Xv6 dispose de deux types de verrous : les verrous actifs [3] et les verrous passifs [4]. Nous commencerons par les verrous actifs. Xv6 représente un verrou actif par une `struct spinlock` [1501]. Le champ important de cette structure est `locked`, un entier nul lorsque le verrou est libre, et non nul quand le verrou est obtenu. Logiquement, xv6 obtiendra le verrou en exécutant du code comme :

```
21   void
22   acquire(struct spinlock *lk)
23   {
24     for(;;) {
25       if(!lk->locked) {
26         lk->locked = 1;
27         break;
28       }
29     }
30   }
```

Malheureusement, cette implémentation ne garantit pas l'exclusion mutuelle sur un multiprocesseur. Il peut arriver que deux processeurs atteignent simultanément la ligne 25, constatent que `lk->locked` est nul, et donc qu'ils obtiennent tous deux le verrou en exécutant la ligne 26. À ce stade, deux processeurs différents possèdent le verrou, ce qui contredit la propriété d'exclusion mutuelle. Au lieu de nous aider à éviter les conditions de concurrence, cette implémentation de `acquire` exhibe sa propre condition de concurrence. Le problème ici est que les lignes 25 et 26 sont exécutées comme des actions distinctes. Pour que le code ci-dessus soit correct, les lignes 25 et 26 doivent être exécutées dans une étape *atomique* (c'est-à-dire indivisible).

Pour exécuter ces deux lignes de manière atomique, xv6 utilise l'instruction x86 spéciale `xchg` [0569]. En une opération atomique, `xchg` échange un mot en mémoire avec le contenu d'un registre. La fonction `acquire` [1574] répète cette instruction `xchg` dans une boucle ; chaque itération lit `lk->locked` et y met 1 de manière atomique 1581. Si le verrou est déjà obtenu, `lk->locked` vaut déjà 1, donc `xchg` retournera 1 et la boucle continue. Si `xchg` retourne 0, à l'inverse, `acquire` a réussi à obtenir le verrou — `locked` valait 0 et vaut maintenant 1 — donc la boucle peut se terminer. Une fois le verrou obtenu, pour faciliter le débogage, `acquire` enregistre le processeur et la trace des appels dans la pile qui ont conduit à l'obtention du verrou. Si un processus oublie de libérer un verrou, cette information peut aider à identifier le coupable. Ces champs de débogage sont protégés par le verrou et ne doivent être modifiés que lorsque le verrou est obtenu.

La fonction `release` [1602] est l'opposée de `acquire` : elle enlève les informations de débogage, puis libère le verrou. La fonction utilise une instruction assembleur pour mettre `locked` à 0, car cette remise à 0 doit se faire de manière atomique pour que l'instruction `xchg` ne voie pas un sous-ensemble des 4 octets de `locked` mis à jour. Le x86 garantit qu'une instruction `movl` sur 32 bits met à jour de manière atomique la totalité des 4 octets. Xv6 ne peut pas utiliser une affectation normale en C, car la spécification du langage ne précise pas qu'une simple affectation est atomique.

3. NdT : Le terme original est *spin-lock*.
4. NdT : Le terme original est *sleep-lock*.

L'implémentation des verrous actifs dans xv6 est spécifique au x86, et xv6 est donc non portable vers d'autres processeurs. Pour permettre des implémentations portables des verrous actifs, le langage C supporte une bibliothèque d'instructions atomiques ; un système d'exploitation portable utiliserait ces instructions.

4.3 Code : Utilisation des verrous

Xv6 utilise des verrous à beaucoup d'endroits pour éviter les conditions de concurrence. Un exemple simple est dans le pilote IDE. Comme mentionné au début du chapitre, `iderw` [4354] a une file des requêtes disques et les processeurs peuvent y ajouter en même temps de nouvelles requêtes [4369]. Pour protéger cette file et d'autres invariants dans le pilote, `iderw` obtient le verrou `idelock` [4365] et le libère à la fin de la fonction.

L'exercice 1 explore la manière d'exhiber dans le pilote IDE la condition de concurrence que nous avons vue au début du chapitre en déplaçant l'appel à `acquire` après la manipulation de la file. Ça vaut la peine d'essayer cet exercice car il montre qu'il n'est pas si facile d'exhiber le problème, ce qui permet de comprendre pourquoi il est difficile de trouver les bogues dus à des conditions de concurrence. Il n'est pas certain que xv6 soit exempt de problèmes.

Un point délicat de l'utilisation des verrous est de décider combien il faut en utiliser et de déterminer quelles données et quels invariants chaque verrou doit protéger. Il y a quelques principes de base. Premièrement, à chaque fois qu'une variable peut être modifiée par un processeur en même temps qu'un autre processeur peut y lire ou y écrire, un verrou devrait être introduit pour empêcher les deux opérations de se chevaucher. Deuxièmement, il faut se rappeler que les verrous protègent des invariants : si un invariant repose sur plusieurs cases mémoires, toutes celles-là nécessitent typiquement une protection par un verrou unique pour garantir le maintien de l'invariant.

Les règles ci-dessus permettent de savoir quand les verrous sont nécessaires, mais ne disent rien sur quand ils ne le sont pas. Il est important pour l'efficacité de ne pas trop verrouiller, car les verrous réduisent le parallélisme. Si le parallélisme n'était pas important, on pourrait alors n'avoir qu'un seul thread et ne pas s'embêter avec des verrous. Un noyau simple peut faire ceci sur un multiprocesseur en ayant un verrou unique qui doit être obtenu à l'entrée dans le noyau, et libéré lors de la sortie (bien que des appels système comme la lecture dans un tube ou `wait` poseraient des problèmes). Beaucoup de systèmes d'exploitation monoprocesseurs ont été convertis pour fonctionner sur des multiprocesseurs en utilisant cette approche, parfois appelée « verrou géant du noyau », mais elle sacrifie le parallélisme : seul un processeur peut exécuter le code du noyau à la fois. Si le noyau fait n'importe quel calcul long, il est plus efficace d'utiliser un ensemble plus grand de verrous à grain fin, pour que le noyau puisse être exécuté sur plusieurs processeurs simultanément.

Finalement, le choix de la granularité des verrous est un exercice de programmation parallèle. Xv6 utilise quelques verrous à gros grain spécifiques pour des structures de données (voir figure 4.2). Par exemple, xv6 a un verrou pour protéger la table entière des processus et ses invariants, qui sont décrits dans le chapitre 5. Une approche à grain plus fin serait d'avoir un verrou par entrée dans la table des processus, afin que des threads travaillant sur des entrées différentes de la table puissent fonctionner en parallèle. Cela compliquerait cependant les opérations avec des invariants sur la table entière, car elles devraient obtenir plusieurs verrous. Les chapitres suivants aborderont la manière dont chaque partie de xv6 traite la concurrence, illustrant ainsi l'utilisation des verrous.

Verrou	Description
`bcache.lock()`	Protège l'allocation des entrées dans le buffer cache
`cons.lock()`	Sérialise les accès au contrôleur de console, évite les sorties mélangées
`ftable.lock()`	Sérialise l'allocation d'une `struct file` dans la table `file`
`icache.lock()`	Protège l'allocation des entrées dans le cache des inodes
`idelock()`	Sérialise les accès au contrôleur de disque et à la file des requêtes
`kmem.lock()`	Sérialise l'allocation de mémoire
`log.lock()`	Sérialise les opérations sur le journal des transactions
`p.lock()` d'un tube	Sérialise les opérations sur chaque tube
`ptable.lock()`	Sérialise la commutation de contexte, et les opérations sur `proc->state` et la table des processus
`tickslock()`	Sérialise les opérations sur le compteur `ticks`
`ip->lock()`	Sérialise les opérations sur chaque inode et son contenu
`b->lock()`	Sérialise les opérations sur chaque buffer

FIGURE 4.2 – Les verrous de xv6.

4.4 Interblocage et ordre des verrouillages

Si un chemin dans le code du noyau doit obtenir plusieurs verrous en même temps, il est important que tous les chemins dans le code obtiennent ces verrous dans le même ordre. S'ils ne le font pas, il y a un risque d'interblocage. Supposons que deux chemins dans le code de xv6 nécessitent les verrous A et B, mais le chemin 1 obtient les verrous dans l'ordre A puis B, et l'autre chemin les obtient dans l'ordre B puis A. Cette situation peut déboucher sur un interblocage si deux threads exécutent les deux chemins simultanément. Supposons que le thread T1 exécute le chemin 1 et obtient le verrou A, et que le thread T2 exécute le chemin 2 et obtient le verrou B. Ensuite, T1 va tenter d'obtenir le verrou B, et T2 va tenter d'obtenir le verrou A. Les deux requêtes de verrouillage vont bloquer indéfiniment, car dans les deux cas, l'autre thread a obtenu le verrou que l'on attend, et ne le libérera pas avant que son `acquire` réussisse. Pour éviter de tels interblocages, tous les chemins dans le code doivent obtenir les verrous dans le même ordre. La nécessité d'un ordre global d'acquisition des verrous signifie que les verrous font effectivement partie intégrante de la spécification de chaque fonction : les fonctions appelantes doivent appeler les fonctions de telle manière que les verrous soient obtenus dans l'ordre convenu.

Xv6 met en œuvre beaucoup de chaînes de verrouillage de longueur 2 impliquant `ptable.lock`, dues à la manière dont `sleep` fonctionne comme évoqué dans le chapitre 5. Par exemple, `ideintr` verrouille `idelock` et appelle `wakeup`, qui obtient le verrou `ptable.lock`. Le code du système de fichiers abrite les plus longues chaînes de verrouillage de xv6. Par exemple, créer un fichier nécessite le verrouillage du répertoire, de l'inode du nouveau fichier, du buffer contenant le bloc disque, de `idelock` et de `ptable.lock`. Pour éviter un interblocage, le code du système de fichiers obtient toujours les verrous dans l'ordre mentionné dans la précédente phrase.

4.5 Gestionnaires d'interruption

Xv6 utilise des verrous actifs dans de nombreuses situations pour protéger des données manipulées à la fois par des gestionnaires d'interruption et par des threads. Par exemple, une interruption d'horloge pourrait [3414] incrémenter `ticks` en même temps qu'un thread du noyau pourrait consulter cette variable dans `sys_sleep` [3823]. Le verrou `tickslock` sérialise les deux accès.

Les interruptions peuvent être une source de concurrence même sur un monoprocesseur : si les interruptions ne sont pas ignorées, le code du noyau peut être interrompu à tout moment pour laisser la place à un gestionnaire d'interruption. Supposons que la fonction `iderw` ait obtenu le verrou `idelock` et soit ensuite interrompue pour exécuter `ideintr` ; celle-ci essayera alors de verrouiller `idelock`, constatera qu'il est déjà verrouillé, et attendra sa libération. Dans ce cas, `idelock` ne sera jamais libéré : seule `iderw` peut le libérer et elle ne reprendra pas avant que `ideintr` ne se termine. Le processeur, et à la fin le système entier, sera interbloqué.

Pour éviter cette situation, si un verrou actif est utilisé par un gestionnaire d'interruption, il ne faut pas qu'un processeur conserve ce verrou lorsque les interruptions sont autorisées. Xv6 est plus conservateur : quand un processeur entre dans une section critique contrôlée par un verrou actif, le noyau garantit que les interruptions sont ignorées sur ce processeur. Les interruptions peuvent toujours être prises en compte sur les autres processeurs, donc un appel à `acquire` dans un gestionnaire d'interruption peut provoquer une attente, qui sera satisfaite par le déverrouillage dans un thread du noyau ; il ne faut juste pas que ce soit sur le même processeur.

Xv6 autorise à nouveau les interruptions quand un processeur n'a plus aucun verrou actif verrouillé ; il faut faire un peu de comptabilité pour gérer les sections critiques imbriquées. La fonction `acquire` appelle `pushcli` [1667] et `release` appelle `popcli` [1679] pour compter le niveau d'imbrication des verrous sur le processeur courant. Quand ce décompte devient nul, `popcli` restaure l'état d'autorisation des interruptions en vigueur avant le début de la section critique la plus englobante. La fonction `cli` (resp. `sti`) exécute l'instruction du x86 pour ignorer (resp. activer) les interruptions.

Il faut que `acquire` appelle `pushcli` avant `xchg` [1581]. Si les deux appels étaient inversés, le verrou pourrait être obtenu pendant quelques cycles alors que les interruptions sont encore autorisées, et une interruption malencontreuse au mauvais moment pourrait alors interbloquer le système. Symétriquement, il faut que `release` appelle `popcli` seulement après le `movl` qui libère le verrou.

4.6 Instructions et ordonnancement des accès à la mémoire

Dans ce chapitre, nous avons supposé que le code est exécuté dans l'ordre dans lequel il apparaît dans le programme. Cependant, nombre de compilateurs et de processeurs exécutent le code sans respecter l'ordre du programme pour atteindre de meilleures performances. Si une instruction met plusieurs cycles, un processeur peut vouloir démarrer plus tôt cette instruction de telle manière qu'elle s'exécute en même temps que d'autres instructions, et ainsi éviter que le processeur ne soit retardé. Par exemple, un processeur peut détecter que dans une séquence d'instructions A puis B, les deux instructions ne sont pas dépendantes l'une de l'autre, et qu'il peut démarrer B avant A de telle manière que B soit terminée quand A se terminera. Un compilateur peut effectuer un réordonnancement similaire en générant l'instruction B avant l'instruction A dans le fichier exécutable. Cependant, la concurrence peut rendre visible ce réordonnancement au niveau logiciel, ce qui peut conduire à un comportement incorrect.

Par exemple, dans le code ci-après pour `insert`, ce serait désastreux si le compilateur ou le processeur rendait visible par les autres processeurs l'effet de la ligne 4 (ou 2 ou 5) après l'effet de la ligne 6 :

```
1     l = malloc(sizeof *l);
2     l->data = data;
3     acquire(&listlock);
4     l->next = list;
```

```
5     list = l;
6     release(&listlock);
```

Si le matériel ou le compilateur réordonnait, par exemple, les effets de la ligne 4 et les rendait visibles avant les effets de la ligne 6, alors un autre processeur pourrait verrouiller `listlock` et observer que `list` pointe sur `l`, mais il n'observerait pas que `l->next` pointe sur le reste de la liste et ne pourrait donc pas voir le reste de la liste.

Pour indiquer au matériel et au compilateur de ne pas effectuer de tels réordonnancements, xv6 utilise `__sync_synchronize()` à la fois dans `acquire` et dans `release`. La fonction `__sync_synchronize` est une barrière mémoire : elle indique au compilateur et au processeur de ne pas réordonnancer les accès à la mémoire à travers la barrière. Xv6 ne se préoccupe du réordonnancement que dans `acquire` et `release` car les accès concurrents aux structures de données autres que la structure du verrou sont effectués entre `acquire` et `release`.

4.7 Verrous passifs

Le code de xv6 doit parfois conserver un verrou pendant une longue durée. Par exemple, le système de fichiers (voir chapitre 6) maintient le fichier verrouillé pendant la lecture ou l'écriture de son contenu sur le disque, ces opérations pouvant prendre des dizaines de millisecondes. L'efficacité impose de libérer le processeur pendant l'attente afin que d'autres threads puissent progresser, ceci signifie donc que xv6 utilise des verrous devant fonctionner correctement lorsqu'ils sont conservés entre plusieurs commutations de contexte. Xv6 fournit de tels verrous sous la forme de *verrous passifs*.

Les verrous passifs de xv6 supportent le fait de libérer le processeur à l'intérieur de la section critique. Cette propriété constitue une difficulté de conception : si le thread T1 a obtenu le verrou L1 et a libéré le processeur, et que le thread T2 souhaite obtenir L1, nous devons garantir que T1 doit pouvoir s'exécuter pendant que T2 attend, afin que T1 puisse déverrouiller L1. T2 ne peut pas utiliser la fonction `acquire` sur un verrou actif ici : elle boucle en ignorant les interruptions, empêchant ainsi T1 de s'exécuter. Pour éviter cet interblocage, la fonction pour obtenir un verrou passif (appelée `acquiresleep`) libère le processeur pendant l'attente et ne désactive pas les interruptions.

La fonction `acquiresleep` [4622] utilise des techniques qui seront expliquées dans le chapitre 5. Vu de haut, un verrou passif possède un champ `locked` lui-même protégé par un verrou actif, et l'appel à `sleep` par `acquiresleep` réalise de manière atomique la libération du processeur et le déverrouillage du verrou actif. Le résultat est que les autres threads peuvent s'exécuter pendant que `acquiresleep` attend.

Du fait que les verrous passifs laissent les interruptions autorisées, ils ne peuvent pas être utilisés dans les gestionnaires d'interruption. Puisque `acquiresleep` peut libérer le processeur, les verrous passifs ne peuvent pas non plus être utilisés dans des sections critiques protégées par des verrous actifs (bien que l'inverse soit vrai : les verrous actifs peuvent être utilisés dans des sections critiques protégées par des verrous passifs).

La plupart du temps, xv6 utilise les verrous actifs car ils ont un faible *overhead* [5]. Xv6 utilise les verrous passifs uniquement dans le système de fichiers, où il est pratique de pouvoir conserver des verrous pendant des opérations lentes sur les disques.

5. NdT : Nous ne traduisons pas ce terme, largement utilisé et n'ayant pas de traduction établie en Français.

4.8 Limitations des verrous

Les verrous permettent souvent de résoudre proprement les problèmes de concurrence, mais il y a des cas où ils sont gênants. Les chapitres suivants pointeront de telles situations dans xv6 ; cette section expose les grandes lignes des problèmes qui se présenteront.

Il arrive parfois qu'une fonction utilise des données devant être protégées par un verrou, et que cette fonction soit appelée à la fois par du code qui a déjà obtenu ce verrou et par du code qui n'aurait autrement pas besoin de ce verrou. Une manière de gérer ce cas est d'avoir deux variantes de la fonction, une qui obtient explicitement le verrou et une autre qui suppose que l'appelant l'a déjà obtenu. Voir `wakeup1` [2953] par exemple. Une autre approche consiste pour la fonction à avoir comme prérequis que les fonctions appelantes obtiennent le verrou, qu'elles en aient l'utilité ou non, comme avec `sched` [2808]. Les développeurs du noyau doivent avoir conscience de tels prérequis.

Il pourrait sembler possible de simplifier les cas où à la fois l'appelante et l'appelée ont besoin d'un verrou en permettant les *verrous récursifs* : si une fonction a déjà obtenu un verrou, toute fonction qu'elle appelle est autorisée à obtenir à nouveau ce verrou. Cependant, le programmeur devra alors réfléchir à toutes les combinaisons de l'appelante et de l'appelée, car les invariants de la structure de données ne seront pas forcément vérifiés après l'obtention du verrou. La question de savoir si les verrous récursifs sont mieux que l'utilisation par xv6 des conventions sur les fonctions ayant comme prérequis l'obtention d'un verrou n'est pas claire. La plus grande leçon est, comme avec l'ordre global de verrouillage pour éviter les interblocages, que les prérequis sur les verrous ne peuvent pas être privés mais doivent être inclus dans les interfaces des fonctions et des modules.

Un cas où les verrous sont insuffisants est lorsqu'un thread doit attendre qu'un autre thread mette à jour une structure de données, par exemple quand un lecteur dans un tube attend qu'un autre thread écrive dans le tube. Le thread en attente ne peut pas conserver le verrou sur les données, car cela empêcherait la mise à jour que ce thread attend. À la place, xv6 fournit un mécanisme distinct qui gère à la fois le verrou et l'attente de l'événement ; voir la description de `sleep` et `wakeup` dans le chapitre 5.

4.9 Le monde réel

Les primitives relatives à la concurrence et la programmation parallèle sont des domaines de recherche actifs, car la programmation avec les verrous représente toujours un défi. Il est préférable d'utiliser les verrous comme une base pour des constructions de plus haut niveau, telles que les files synchronisées, bien que xv6 ne fasse pas cela. Si vous programmez avec des verrous, il est prudent d'utiliser un outil qui tente d'identifier les conditions de concurrence car il est facile de rater un invariant nécessitant un verrou.

La plupart des systèmes d'exploitation supportent les threads POSIX (pthreads), qui permettent à un processus utilisateur d'avoir plusieurs threads s'exécutant simultanément sur différents processeurs. Les pthreads supportent les verrous en mode utilisateur, les barrières, etc. Supporter les pthreads nécessite du support du système d'exploitation. Par exemple, si un pthread attend dans un appel système, un autre pthread du même processus devrait pouvoir s'exécuter sur le processeur. Un autre exemple est lorsqu'un pthread change l'espace d'adressage du processus (en l'élargissant ou le rétrécissant), le noyau doit alors faire en sorte que les autres processeurs exécutant des threads du même processus mettent à jour leur table des pages pour refléter cette modification. Sur le x86, ceci implique de vider le *Translation Look-aside Buffer* des

autres processeurs en utilisant des interruptions inter-processeurs [6]

Il est possible d'implémenter les verrous sans utiliser des instructions atomiques, mais c'est coûteux et la plupart des systèmes d'exploitation utilisent les instructions atomiques.

Les verrous peuvent être coûteux si beaucoup de processeurs tentent d'obtenir le même verrou au même moment. Si un processeur a un verrou dans son cache local et un autre processeur veut obtenir ce verrou, alors l'instruction atomique qui met à jour la ligne de cache qui contient le verrou doit propager la ligne de cache vers les autres processeurs, et également invalider les autres copies de cette ligne de cache. Récupérer une ligne de cache d'un autre processeur peut être plus coûteux de plusieurs ordres de grandeur que récupérer une ligne du cache local du processeur.

Pour éviter les coûts associés aux verrous, beaucoup de systèmes d'exploitation utilisent des structures de données et des algorithmes *sans verrouillage* [7]. Par exemple, il est possible d'implémenter une liste chaînée comme celle du début du chapitre sans avoir besoin de verrou et en utilisant juste une instruction atomique pour insérer un élément. Cependant, la programmation sans verrouillage est plus complexe qu'avec des verrous ; par exemple, on doit être conscient du réordonnancement des instructions et des accès mémoire. Programmer avec des verrous est déjà difficile, xv6 évite donc la complexité supplémentaire de la programmation sans verrouillage.

4.10 Exercices

1. Déplacez l'appel à `acquire` dans `idrw` avant l'appel à `sleep`. Est-ce qu'il y a une condition de concurrence ? Pourquoi ne l'observez-vous pas lorsque vous démarrez xv6 et lancez `stressfs` ? Ajoutez une boucle vide dans la section critique ; que constatez-vous maintenant ? Expliquez.
2. Retirez l'appel à `xchg` dans `acquire`. Expliquez ce qui se passe lorsque vous démarrez xv6.
3. Écrivez un programme parallèle utilisant les threads POSIX, qui sont supportés sur la plupart des systèmes d'exploitation. Par exemple, implémentez une table de hachage parallèle et vérifiez par la mesure si le nombre d'insertions et de recherches est proportionnel au nombre de cœurs.
4. Implémentez un sous-ensemble des pthreads dans xv6, c'est-à-dire implémentez une bibliothèque en mode utilisateur pour qu'un processus puisse avoir plus d'un thread et faites en sorte que ces threads s'exécutent en parallèle sur des processeurs distincts. Votre conception doit gérer correctement le cas où un thread fait un appel bloquant au système et le cas où un thread change l'espace d'adressage partagé.

6. NdT : Le terme original est *Inter-processor interrupts*, ou IPI.
7. NdT : Le terme original est *lock-free*.

Chapitre 5

Ordonnancement

Tout système d'exploitation est susceptible de fonctionner avec davantage de processus que de processeurs sur l'ordinateur, il faut donc un système pour partager le temps des processeurs entre les processus. Idéalement, le partage devrait être transparent pour les processus utilisateurs. Une approche fréquente consiste à donner à chaque processus l'illusion qu'il dispose de son propre processeur virtuel en *multiplexant* les processus sur les processeurs matériels. Ce chapitre explique comment xv6 effectue ce multiplexage.

5.1 Multiplexage

Xv6 multiplexe en commutant chaque processeur d'un processus vers un autre dans deux cas. Premièrement, le mécanisme de `sleep` et `wakeup` commute lorsqu'un processus attend qu'une entrée/sortie avec un périphérique ou un tube se termine, ou qu'un fils se termine, ou attend dans l'appel système `sleep`. Deuxièmement, xv6 force périodiquement une commutation lorsqu'un processus exécute des instructions en mode utilisateur. Ce multiplexage crée l'illusion que chaque processus dispose de son propre processeur, tout comme xv6 utilise l'allocateur mémoire et les tables des pages du matériel pour créer l'illusion que chaque processus dispose de sa propre mémoire.

Implémenter le multiplexage pose quelques difficultés. Premièrement, comment commuter d'un processus vers un autre ? Bien que l'idée de la commutation de contexte soit simple, son implémentation fait partie des codes les plus opaques de xv6. Deuxièmement, comment commuter de manière transparente entre processus utilisateurs ? Xv6 utilise la technique standard consistant à provoquer les commutations via les interruptions d'horloge. Troisièmement, plusieurs processeurs peuvent être en train de commuter simultanément, et une stratégie de verrouillage est nécessaire pour éviter les conditions de concurrence. Quatrièmement, la mémoire et les autres ressources d'un processus doivent être libérées lorsqu'un processus se termine, mais il ne peut par tout faire lui-même car, par exemple, il ne peut pas libérer sa propre pile noyau alors qu'il l'utilise encore. Finalement, chaque cœur d'une machine multi-cœurs doit se rappeler quel processus il est en train d'exécuter afin que les appels système affectent l'état du bon processus. Xv6 essaye de résoudre ces problèmes aussi simplement que possible, mais le code résultant est néanmoins délicat.

Xv6 doit fournir aux processus les moyens de se coordonner entre eux. Par exemple, un processus parent peut avoir besoin d'attendre que l'un de ses fils se termine, ou un processus lisant dans un tube peut attendre qu'un autre processus écrive dans le tube. Au lieu que le processus en attente gaspille le temps de processeur en vérifiant en boucle si l'événement souhaité est arrivé, xv6 permet à un processus d'abandonner le processeur et de s'endormir en

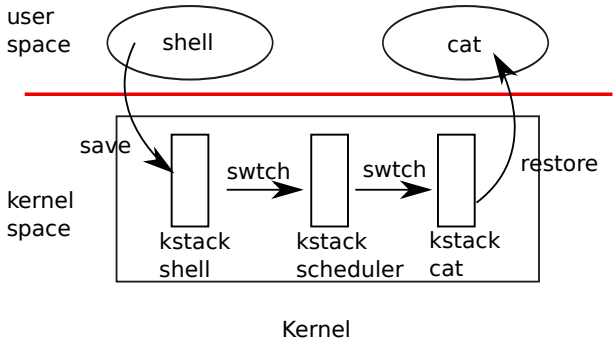

FIGURE 5.1 – Commutation d'un processus utilisateur vers un autre. Dans cet exemple, xv6 tourne sur un monoprocesseur (avec donc un seul thread ordonnanceur).

attendant l'événement, et permet à un autre processus de réveiller le premier. Il faut prendre garde d'éviter les conditions de concurrence pouvant déboucher sur la perte de la notification d'un événement. Comme exemple de ces problèmes et de leur solution, ce chapitre examine l'implémentation des tubes.

5.2 Code : commutation de contexte

La figure 5.1 illustre les principales étapes impliquées dans la commutation d'un processus vers un autre : le thread de l'ancien processus bascule du mode utilisateur vers le mode noyau (appel système ou interruption), puis il y a une commutation de contexte vers le thread ordonnanceur du processeur courant, puis il y a une nouvelle commutation de contexte vers le thread noyau du nouveau processus et enfin un retour de trappe est effectué vers le mode utilisateur dans le nouveau processus. L'ordonnanceur de xv6 dispose de son propre thread (avec sa pile et sa sauvegarde des registres) car il peut arriver que l'utilisation de la pile noyau d'un processus ne soit pas fiable : nous verrons un exemple dans `exit` (voir 5.8 page 75). Dans cette section, nous allons examiner le mécanisme de la commutation entre un thread en mode noyau et le thread ordonnanceur.

Commuter d'un thread vers un autre signifie sauver les registres du processeur utilisés par l'ancien thread et restaurer les registres précédemment sauvegardés du nouveau thread. Le fait que les registres `%esp` et `%eip` soient sauvés et restaurés signifie que le processeur changera de pile et changera de code à exécuter.

La fonction `swtch` effectue la sauvegarde et la restauration pour une commutation de thread. Elle ne sait pas ce qu'est un thread ; elle ne fait que sauvegarder et restaurer des ensembles de registres, appelés *contextes*. Lorsqu'il est temps pour un processus d'abandonner le processeur, le thread noyau du processus appelle `swtch` pour sauvegarder son propre contexte et revenir à celui de l'ordonnanceur. Chaque contexte est représenté par une `struct context*`, un pointeur vers une structure placée dans la pile noyau du processus concerné. La fonction `swtch` prend deux arguments : `struct context **old` et `struct context *new`. Elle empile les registres actuels et sauvegarde le pointeur de pile dans `*old`. Puis, `swtch` copie `new` vers `%esp`, dépile les registres sauvegardés à l'époque et se termine.

Suivons un processus utilisateur à travers `swtch` dans l'ordonnanceur. Nous avons vu dans le chapitre 3 (3.7, page 51) la possibilité qu'à la fin de chaque interruption, `trap` appelle `yield`, qui à son tour appelle `sched`, qui elle-même appelle `swtch` pour sauvegarder le contexte actuel dans `proc->context` et commuter vers le contexte de l'ordonnanceur précédemment

sauvegardé dans `cpu->scheduler` [2822].

La fonction `swtch` [3052] commence par copier ses arguments depuis la pile vers les registres `%eax` et `%edx` [3060-3061] sauvegardés par la fonction appelante ; `swtch` doit faire cela avant de changer le pointeur de pile et de ne plus avoir accès aux arguments via `%esp`. Puis, `swtch` empile l'état des registres, créant ainsi une structure `context` sur la pile actuelle. Seuls les registres sauvegardés par la fonction appelée doivent être copiés ; sur x86, la convention est que ce sont les registres `%ebp`, `%ebx`, `%esi`, `%edi` et `%esp`. La fonction `swtch` empile explicitement les 4 premiers [3064-3067] ; elle sauvegarde implicitement le dernier puisqu'il s'agit du pointeur `struct context*` écrit dans `*old` [3070]. Il reste un registre important : le compteur ordinal `%eip`. Il a déjà été sauvegardé dans la pile par l'instruction `call` ayant provoqué l'appel à `swtch`. Une fois l'ancien contexte sauvegardé, `swtch` est prête à restaurer le nouveau. Elle copie le pointeur vers le nouveau contexte dans le pointeur de pile [3071]. La nouvelle pile a la même forme que l'ancienne que `swtch` a cessé d'utiliser — la nouvelle pile *était* l'ancienne lors d'un précédent appel à `swtch` — donc `swtch` peut inverser la séquence pour restaurer le nouveau contexte. Elle dépile les valeurs pour `%edi`, `%esi`, `%ebx` et `%ebp`, puis se termine [3074-3078]. Puisque `swtch` a changé le pointeur de pile, les valeurs restaurées et l'adresse de retour sont celles du nouveau contexte.

Dans notre exemple, `sched` a appelé `swtch` pour commuter vers `cpu->scheduler`, le contexte propre à chaque processeur. Ce contexte a été sauvé par l'ordonnanceur lorsqu'il a appelé `switch` [2781]. Lorsque l'appel de `swtch` que nous avons suivi se termine, le retour se fait non pas dans la fonction `sched`, mais dans la fonction `scheduler`, et le pointeur de pile pointe vers la pile de l'ordonnanceur du processus courant.

5.3 Code : ordonnancement

La section précédente décrivait les détails de bas niveau de `swtch` ; maintenant, prenons `swtch` comme une donnée et examinons la commutation d'un processus vers un autre processus à travers l'ordonnanceur. Un processus qui désire abandonner le processeur doit obtenir le verrou `ptable.lock` associé à la table des processus, libérer tous les autres verrous qu'il pourrait avoir obtenu, mettre à jour son propre état (`proc->state`), et ensuite appeler `sched`. La fonction `yield` [2828] suit cette convention, tout comme `sleep` et `exit` que nous examinerons plus loin. La fonction `sched` vérifie deux fois ces conditions [2813-2818], puis vérifie une implication de ces conditions : puisqu'un verrou est obtenu, le processeur devrait ignorer les interruptions. Finalement, `sched` appelle `swtch` pour sauver le contexte dans `proc->context` et commute vers le contexte de l'ordonnanceur situé dans `cpu->scheduler`. La fonction `swtch` retourne avec la pile de l'ordonnanceur comme si le `swtch` appelé par l'ordonnanceur s'était terminé [2781]. L'ordonnanceur continue sa boucle `for`, trouve un processus prêt à tourner, commute vers ce processus et le cycle se répète.

Nous venons de voir que xv6 conserve le verrou `ptable.lock` à travers les appels à `swtch` : l'appelant de `swtch` doit déjà posséder le verrou, et son contrôle passe au code exécuté dans le nouveau contexte. Cette convention est inhabituelle avec les verrous ; habituellement, le thread qui obtient un verrou est également responsable de sa libération, ce qui facilite les raisonnements sur la justesse. Pour la commutation de contexte, il devient nécessaire de passer outre cette convention puisque `ptable.lock` protège les invariants (sur l'état du processus et les champs du contexte) qui ne sont pas vérifiés pendant l'exécution de `swtch`. Un exemple de problème pouvant arriver si `ptable.lock` n'était pas verrouillé pendant `swtch` : un autre processeur pourrait décider d'exécuter le processus après que `yield` ait mis son état à `RUNNABLE`, mais avant que `swtch` n'ait pu arrêter d'utiliser sa propre pile noyau. Le résultat serait que deux processeurs utiliseraient alors la même pile, ce qui ne peut pas être correct.

Un thread du noyau abandonne toujours le processeur dans `sched` et commute toujours vers le même emplacement dans l'ordonnanceur, qui (presque) toujours commute vers un thread du noyau qui avait antérieurement appelé `sched`. Donc, si on voulait afficher les numéros de ligne où xv6 commute les threads, on observerait le motif simple : 2781, 2822, 2781, 2822, et ainsi de suite. Les procédures dans lesquelles cette commutation stylisée entre deux threads se produit sont parfois appelées *coroutines* ; dans cet exemple, `sched` et `scheduler` sont des coroutines l'une de l'autre.

Il y a un cas où l'appel à `swtch` par l'ordonnanceur ne se termine pas dans `sched`. Nous l'avons vu dans le chapitre 1 (1.6, page 28) : quand un nouveau processus est ordonnancé pour la première fois, il commence son exécution dans `forkret` [2853]. Cette fonction `forkret` est là pour libérer `ptable.lock` ; sinon, le nouveau processus pourrait démarrer à `trapret`.

La fonction `scheduler` [2758] contient une boucle simple : trouver un processus prêt à tourner, l'exécuter jusqu'à ce qu'il abandonne le processeur, et recommencer. Elle verrouille `ptable.lock` pour la plupart de ses actions, mais le déverrouille (et autorise explicitement les interruptions) une fois par itération de la boucle englobante. C'est important dans le cas spécial où le processeur est inactif (c'est-à-dire qu'il ne trouve aucun processus prêt à tourner). Si un ordonnanceur sans processus prêt pouvait rester dans la boucle avec le verrou toujours verrouillé, aucun autre processeur exécutant un processus ne pourrait effectuer une commutation de contexte ou utiliser d'appel système relatif aux processus, et en particulier marquer un processus comme `RUNNABLE` (prêt à tourner) pour arrêter la boucle du processeur ordonnanceur. La raison pour laquelle les interruptions sont autorisées périodiquement est que, sur un processeur inactif, il peut n'y avoir aucun processus `RUNNABLE` car les processus (par exemple le Shell) peuvent être en attente d'entrée/sortie ; si l'ordonnanceur ignorait en permanence les interruptions, la fin de l'entrée/sortie souhaitée ne serait jamais notifiée.

L'ordonnanceur parcourt la table des processus pour détecter un processus prêt à tourner, c'est-à-dire un qui a `p->state==RUNNABLE`. Une fois le processus trouvé, l'ordonnanceur positionne la variable propre au processeur `c->proc` pour indiquer le processus courant, change la table des pages pour celle du processus avec `switchuvm`, marque le processus comme `RUNNING`, puis appelle `swtch` pour démarrer son exécution [2774-2781].

Une manière de voir la structure du code d'ordonnancement est qu'il maintient un ensemble d'invariants sur chaque processus et verrouille `ptable.lock` quand ces invariants ne sont pas vérifiés. Un des invariants est « si un processus est `RUNNING`, un appel à `yield` par l'interruption de l'horloge doit pouvoir commuter ce processus pour un autre » ; ceci signifie que les registres du processeur doivent contenir les valeurs des registres du processus (c'est-à-dire qu'elles ne sont pas dans un `context`), `%cr3` doit pointer sur la table des pages du processus, `%esp` doit pointer dans la pile noyau du processus de telle manière que `swtch` puisse empiler les registres correctement, et `c->proc` doit pointer vers l'entrée du processus dans la table des processus `ptable.proc[]`. Un autre invariant est « si un processus est `RUNNABLE`, un ordonnanceur sur un processeur inactif doit pouvoir l'exécuter » ; ceci signifie que `p->context` doit contenir les variables du thread noyau du processus, qu'aucun processeur n'est en cours d'utilisation de la pile noyau du processus, qu'aucun `%cr3` de processeur ne pointe sur la page des tables du processus et enfin qu'aucun `c->proc` de processeur ne pointe sur le processus.

Maintenir les invariants ci-dessus est la raison pour laquelle xv6 verrouille `ptable.lock` dans un thread (souvent dans `yield`) et le déverrouille dans un thread différent (le thread de l'ordonnanceur ou un autre thread noyau). Une fois que le code a commencé à modifier l'état d'un processus pour le mettre à `RUNNABLE`, il doit conserver le verrou jusqu'à ce qu'il a terminé la restauration des invariants : le point de libération le plus tôt se situe après la fin de l'utilisation de la table des pages du processus et la remise à 0 de `c->proc` dans `scheduler`. De la même façon, une fois que `scheduler` a commencé à convertir un processus vers l'état `RUNNING`, le

verrou ne peut être libéré jusqu'à ce que le thread noyau s'exécute normalement (après l'appel à `swtch` dans `yield`).

Le verrou `ptable.lock` protège également d'autres choses : l'allocation des identificateurs de processus et des emplacements disponibles dans la table des processus. L'interaction entre `exit` et `wait`, la machinerie pour éviter de perdre des événements de réveil (voir 5.5, page 71), et probablement d'autres choses encore. Il pourrait être intéressant de se demander si les différentes utilisations de `ptable.lock` peuvent être scindées, notamment pour la clarté et sans doute pour les performances.

5.4 Code : `mycpu` et `myproc`

Xv6 dispose d'une `struct cpu` pour chaque processeur afin de mémoriser le processus actuellement sur le processeur (le cas échéant), l'identifiant matériel unique (`apicid`), et d'autres informations. La fonction `mycpu` [2437] retourne la `struct cpu` du processeur courant. Elle l'obtient en lisant l'identifiant du processeur dans l'APIC local et en le cherchant dans le tableau de `struct cpu`. La valeur de retour de `mycpu` est fragile : si l'horloge interrompait le processeur et provoquait une migration du thread vers un processeur différent, la valeur de retour ne serait alors plus correcte. Pour éviter ce problème, xv6 impose que les appelants de `mycpu` désactivent les interruptions, et ne les réactivent qu'après qu'ils aient fini d'utiliser la `struct cpu` récupérée.

La fonction `myproc` [2457] renvoie un pointeur vers la `struct proc` du processus actuellement sur le processeur. Cette fonction désactive les interruptions, appelle `mycpu`, récupère le pointeur vers le processus courant (`c->proc`) hors de la `struct cpu`, puis réactive les interruptions. Si aucun processus n'est en cours d'exécution, car l'appelant est dans `scheduler`, `myproc` retourne un pointeur nul. La valeur de retour de `myproc` peut être utilisée sereinement même si les interruptions sont autorisées : si une interruption d'horloge migre le processus appelant vers un autre processeur, le pointeur vers sa `struct proc` restera identique.

5.5 Sleep et wakeup

L'ordonnancement et les verrous aident à dissimuler l'existence d'un processus vis-à-vis des autres mais, jusqu'ici, nous ne disposons pas d'abstraction pour permettre à des processus d'interagir intentionnellement. Les fonctions `sleep` et `wakeup` remplissent ce vide, permettant à un processus de s'endormir en attendant un événement, et un autre processus le réveillera lorsque l'événement sera arrivé. Cet endormissement et ce réveil sont des mécanismes souvent nommés *coordination séquentielle* ou *synchronisation conditionnelle*, et il existe beaucoup d'autres mécanismes similaires dans la littérature relative aux systèmes d'exploitation.

Pour illustrer ce que cela signifie, considérons une simple file producteur/consommateur. Cette file est similaire à celle que les processus utilisent pour alimenter le pilote IDE (voir chapitre 3, 3.9, page 52) mais s'abstrait de tout code spécifique au pilote IDE. La file permet à un processus d'envoyer un pointeur non nul vers un autre processus. S'il y avait un seul émetteur et un seul récepteur s'exécutant sur des processeurs différents, et si le compilateur n'optimisait pas trop agressivement, l'implémentation ci-dessous serait correcte :

```
100   struct q {
101     void *ptr;
102   };
103
```

```
104   void*
105   send(struct q *q, void *p)
106   {
107     while(q->ptr != 0)
108       ;
109     q->ptr = p;
110   }
111
112   void*
113   recv(struct q *q)
114   {
115     void *p;
116
117     while((p = q->ptr) == 0)
118       ;
119     q->ptr = 0;
120     return p;
121   }
```

La fonction `send` boucle jusqu'à ce que la file soit vide (`ptr ==`) et ajoute alors le pointeur `p` à la file. La fonction `recv` boucle jusqu'à ce que la file ne soit plus vide, et en extrait le pointeur. Lorsqu'ils tournent sur différents processeurs, `send` et `recv` modifient tous deux `q->ptr`, mais `send` ne le modifie que lorsqu'il est nul alors que `recv` ne le modifie que lorsqu'il est non nul, donc aucune mise à jour n'est perdue.

L'implémentation ci-dessus est coûteuse. Si l'émetteur émet rarement, le récepteur va passer la plupart du temps dans la boucle `while` en attendant un pointeur. Le processeur du récepteur pourrait faire un travail plus productif que l'*attente active* qui surveille en boucle `q->ptr`. Pour éviter l'attente active, il faut que le récepteur abandonne le processeur et le reprenne uniquement lorsque `send` dépose un pointeur.

Imaginons une paire d'appels `sleep` et `wakeup` qui fonctionnent comme suit. `Sleep(chan)` s'endort sur une valeur arbitraire `chan`, nommée *canal d'attente*[1]. Cette fonction `sleep` met le processus en sommeil, en libérant le processeur qui peut se consacrer à d'autres tâches. `Wakeup(chan)` réveille tous les processus endormis sur `chan` (s'il y en a), provoquant la sortie de `sleep`. Si aucun processus n'attend sur `chan`, `wakeup` ne fait rien. Nous pouvons alors modifier l'implémentation de la file pour utiliser `sleep` et `wakeup` :

```
201   void*
202   send(struct q *q, void *p)
203   {
204     while(q->ptr != 0)
205       ;
206     q->ptr = p;
207     wakeup(q);  /* wake recv */
208   }
209
210   void*
211   recv(struct q *q)
212   {
213     void *p;
214
215     while((p = q->ptr) == 0)
216       sleep(q);
```

1. NdT : Le terme original est *wait channel*, parfois abrégé *wchan*, que l'on trouve par exemple dans l'affichage de la commande Unix `ps` ou `top`.

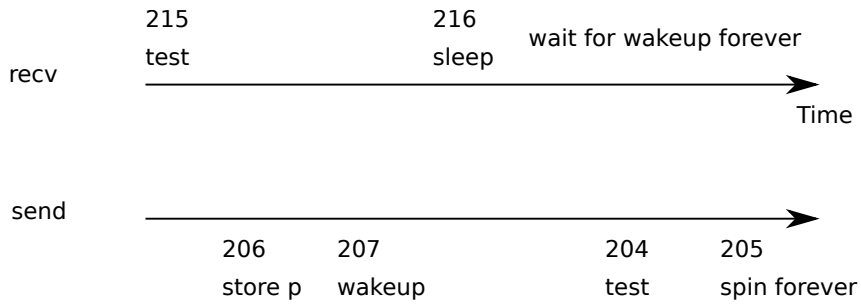

FIGURE 5.2 – Exemple de problème de réveil perdu.

```
217     q->ptr = 0;
218     return p;
219   }
```

La fonction `recv` libère maintenant le processeur au lieu de boucler, ce qui est bien. Cependant, il s'avère que la conception de `sleep` et `wakeup` n'est pas si simple avec cette interface et sans souffrir du problème connu comme « réveil perdu » (voir figure 5.2). Supposons que `recv` détecte que `q>ptr == 0` en ligne 215. Si pendant que `recv` est entre les lignes 215 et 216, `send` tourne sur un autre processeur : `send` modifie `q->ptr` pour y mettre une valeur non nulle et appelle `wakeup`, qui ne trouve aucun processus endormi et ne fait donc rien. Maintenant, `recv` continue son exécution à la ligne 216 : elle appelle `sleep` et s'endort. Ceci pose un problème : `recv` est endormi en attendant un pointeur qui est déjà arrivé. Le prochain appel à `send` attendra alors que `recv` consomme le pointeur dans la file : à ce stade, le système sera *interbloqué*.

La racine de ce problème est que l'invariant « `recv` ne s'endort que lorsque `q->ptr == 0` » est violé par `send` qui s'exécute au mauvais moment. Une manière incorrecte de protéger l'invariant serait de modifier le code pour `recv` comme suit :

```
300   struct q {
301     struct spinlock lock;
302     void *ptr;
303   };
304
305   void*
306   send(struct q *q, void *p)
307   {
308     acquire(&q->lock);
309     while(q->ptr != 0)
310       ;
311     q->ptr = p;
312     wakeup(q);
313     release(&q->lock);
314   }
315
316   void*
317   recv(struct q *q)
318   {
319     void *p;
320
321     acquire(&q->lock);
322     while((p = q->ptr) == 0)
323       sleep(q);
```

```
324     q->ptr = 0;
325     release(&q->lock);
326     return p;
327   }
```

On pourrait espérer que cette version de `recv` éviterait le réveil perdu puisque le verrou empêche `send` de s'exécuter entre les lignes 322 et 323. Il fait effectivement cela, mais il provoque un interblocage : `recv` conserve le verrou pendant le sommeil, donc l'émetteur attendra indéfiniment le déverrouillage.

Nous allons corriger le précédent schéma en modifiant l'interface de `sleep` : l'appelant doit passer un verrou à `sleep` de façon que celle-ci puisse le libérer après que le processus appelant ait été marqué comme endormi sur le canal. Le verrou forcera un `send` concurrent à attendre jusqu'à ce que le récepteur ait fini de se mettre lui-même en sommeil, de façon que `wakeup` trouve le récepteur endormi et le réveille. Une fois le récepteur réveillé, `sleep` verrouille à nouveau le verrou avant de se terminer. Notre nouveau schéma, correct, s'utilise comme suit :

```
400   struct q {
401     struct spinlock lock;
402     void *ptr;
403   };
404
405   void*
406   send(struct q *q, void *p)
407   {
408     acquire(&q->lock);
409     while(q->ptr != 0)
410       ;
411     q->ptr = p;
412     wakeup(q);
413     release(&q->lock);
414   }
415
416   void*
417   recv(struct q *q)
418   {
419     void *p;
420
421     acquire(&q->lock);
422     while((p = q->ptr) == 0)
423       sleep(q, &q->lock);
424     q->ptr = 0;
425     release(&q->lock);
426     return p;
427   }
```

Le fait que la fonction `recv` conserve le verrou `q->lock` empêche `send` d'essayer de réveiller `recv` entre le test de `q->ptr` et l'appel à `sleep`. Il faut que `sleep`, de manière atomique, libère `q->lock` et mette le processus récepteur en attente.

Une implémentation complète de l'émetteur et du récepteur utiliserait également `sleep` dans le code de `send` pour attendre qu'un récepteur consomme la valeur d'un précédent `send`.

5.6 Code : `sleep` et `wakeup`

Regardons l'implémentation de `sleep` [2874] et `wakeup` [2953]. L'idée de base est que `sleep` marque le processus courant comme `SLEEPING`, puis appelle `sched` pour libérer le processeur ; `wakeup` cherche un processus endormi sur le canal d'attente indiqué et le marque comme `RUNNABLE`. Les appelants de `sleep` et `wakeup` peuvent utiliser comme canal un nombre pratique pour les deux parties. Xv6 utilise souvent l'adresse d'une structure de données impliquée dans l'attente.

La fonction `sleep` [2874] commence par quelques contrôles sanitaires : il doit y avoir un processus courant [2878] et `sleep` doit recevoir un verrou [2881-2882]. Ensuite, `sleep` obtient le verrou `ptable.lock` [2891]. À ce stade, le processus qui va s'endormir a bien obtenu à la fois `ptable.lock` et `lk`. Conserver `lk` était nécessaire dans l'appelant (dans l'exemple, `recv`) : cela garantissait qu'aucun autre processus (dans l'exemple, celui exécutant `send`) ne pouvait appeler `wakeup(chan)`. Maintenant que `sleep` a obtenu `ptable.lock`, il n'y a aucun danger à libérer `lk` : un autre processus peut appeler `wakeup(chan)`, mais cette fonction `wakeup` ne pourra rien faire avant d'obtenir `ptable.lock`, elle attendra donc que `sleep` ait fini de mettre le processus en sommeil, empêchant ainsi le `wakeup` de rater le `sleep`.

Il y une petite complication : si `lk` vaut `&ptable.lock`, alors `sleep` s'interbloquera en essayant de l'obtenir comme `&ptable.lock` pour le libérer ensuite comme `lk`. Dans ce cas, `sleep` considère que le verrouillage et le déverrouillage s'annulent l'un l'autre et les saute entièrement [2890]. Par exemple, `wait` [2707] appelle `sleep` avec `&ptable.lock`.

Maintenant que `sleep` a verrouillé `ptable.lock` et plus aucun autre verrou, elle peut mettre le processus en sommeil en mémorisant le canal d'attente, en modifiant l'état du processus et en appelant `sched` [2895-2898].

Ultérieurement, un processus appellera `wakeup(chan)`. La fonction `wakeup` [2964] verrouille `ptable.lock` et appelle `wakeup1` qui fait tout le travail. Il faut que la fonction `wakeup` obtienne `ptable.lock` à la fois parce qu'elle manipule les états des processus et parce que, comme nous venons de le voir, `ptable.lock` permet à `sleep` et à `wakeup` de ne pas se rater. `Wakeup1` est une fonction distincte car, parfois, l'ordonnanceur doit réveiller un processus alors qu'il a déjà verrouillé `ptable.lock` ; nous verrons un exemple plus loin. Cette fonction `wakeup1` [2953] parcourt la table des processus. Lorsqu'elle en trouve un dans l'état `SLEEPING` avec le canal d'attente recherché, elle modifie son état à `RUNNABLE`. La prochaine fois que l'ordonnanceur s'exécutera, il verra que le processus est prêt à tourner.

Le code de xv6 appelle toujours `wakeup` en conservant le verrou qui protège la condition d'attente ; dans l'exemple ci-dessus, ce verrou est `q->lock`. À strictement parler, c'est suffisant si `wakeup` suit toujours l'appel à `acquire` (autrement dit, on pourrait appeler `wakeup` après `release`). Pourquoi les règles de verrouillage pour `sleep` et `wakeup` garantissent-elles qu'un processus endormi ne ratera pas le `wakeup` dont il a besoin ? Le processus qui va s'endormir conserve soit le verrou sur la condition, soit `ptable.lock`, ou encore les deux entre le point avant le test de la condition et le point après le marquage comme endormi. Si un thread concurrent met la condition à vrai, ce thread doit posséder le verrou sur la condition soit avant que le thread en cours d'endormissement ne l'ait obtenu, soit après qu'il l'ait libéré dans `sleep`. Si c'est avant, le thread en cours d'endormissement doit déjà avoir vu la nouvelle valeur de condition et il a quand même décidé de s'endormir, donc ça n'a aucune importance s'il rate le réveil. Si c'est après, alors le processus qui réveille ne peut pas obtenir le verrou sur la condition avant que `sleep` ait obtenu `ptable.lock`, donc le verrouillage de `ptable.lock` par `wakeup` doit attendre que `sleep` ait complètement terminé de mettre le processus en sommeil. Ensuite, `wakeup` verra le processus en sommeil et le réveillera (à moins qu'un autre processus ne le réveille en premier).

Il arrive parfois que de multiples processus s'endorment sur le même canal ; par exemple il peut y avoir plusieurs processus lisant dans un même tube. Un simple appel à `wakeup` les réveillera tous. L'un d'eux s'exécutera en premier et obtiendra le verrou avec lequel `sleep` a été appelé et, dans le cas des tubes, lira les données disponibles dans le tube. Les autres processus verront qu'il n'y a plus de données à lire bien qu'ils aient été réveillés. De leur point de vue, le réveil était « infondé »[2] et ils doivent se rendormir. C'est la raison pour laquelle `sleep` est toujours appelée à l'intérieur d'une boucle qui teste une condition.

Il n'y a pas de danger à utiliser le même canal pour `sleep` et `wakeup` avec deux conditions différentes : les processus verront des réveils infondés, mais la boucle décrite ci-dessus tolérera le problème. Une grande partie du charme de `sleep` et `wakeup` est qu'elles sont à la fois légères (pas besoin de créer des structures de données spécifiques pour les canaux d'attente) et fournissent un niveau d'indirection (les appelants n'ont pas besoin de savoir avec quel processus spécifique ils interagissent).

5.7 Code : les tubes

La file simple que nous avons utilisée plus tôt dans ce chapitre était un jouet, mais xv6 contient deux vraies files utilisant `sleep` et `wakeup` pour synchroniser les lecteurs et les écrivains. L'une des deux est dans le pilote IDE : un processus ajoute une requête pour le disque dans la file, puis appelle `sleep`. Le gestionnaire de l'interruption IDE utilise `wakeup` pour signaler au processus que sa requête est achevée.

L'implémentation des tubes constitue un exemple plus complexe. Nous avons vu l'interface des tubes dans le chapitre 0 (voir 0.3, page 15) : les octets écrits à l'une des extrémités d'un tube sont copiés dans un tampon du noyau et peuvent ensuite être lus à l'autre extrémité du tube. Les chapitres ultérieurs détailleront le support des descripteurs de fichiers pour les tubes, mais examinons maintenant l'implémentation de `pipewrite` et `piperead`.

Chaque tube est représenté par une `struct pipe` qui contient un verrou `lock` et un tampon `data`. Les champs `nread` et `nwrite` comptabilisent le nombre d'octets lus et écrits dans le tampon. Celui-ci est circulaire : l'octet après `buf[PIPESIZE-1]` est `buf[0]`. Les compteurs ne reviennent pas à 0. Cette convention permet de distinguer un tampon plein (`nwrite == nread+PIPESIZE`) d'un tampon vide (`nwrite == nread`), mais cela signifie que l'accès dans le tampon doit être réalisé avec `buf[nread%PIPESIZE]` et non simplement `buf[nread]` (idem pour `nwrite`). Supposons que des appels à `piperead` et à `pipewrite` arrivent simultanément sur deux processeurs distincts.

La fonction `pipewrite` [6830] commence par obtenir le verrou du tube qui protège les compteurs, les données et leurs invariants associés. De son côté, `piperead` [6851] tente en vain également d'obtenir le verrou : elle attend activement dans `acquire` [1574]. Pendant cette attente, `pipewrite` parcourt les octets à écrire — `addr[0]`, `addr[1]`..., `addr[n-1]` — en les ajoutant au fur et à mesure dans le tube [6844]. Dans cette boucle, le tampon peut devenir plein [6836] : si c'est le cas, `pipewrite` appelle `wakeup` pour signaler à des lecteurs en attente la présence de données dans le tampon, puis se met en sommeil sur `&p->nwrite` pour attendre qu'un lecteur retire quelques octets du tampon. La fonction `sleep` libère `p->lock` lorsqu'elle endort le processus qui a appelé `pipewrite`.

Puisque `p->lock` est maintenant libéré, `piperead` le verrouille et entre dans sa section critique : elle détecte que `p->nread != p->nwrite` [6856] (`pipewrite` s'était endormie car `p->nwrite == p->nread+PIPESIZE` [6836]), donc elle continue dans la boucle `for`, extrait les données du tube [6863-6867] et incrémente `nread` avec le nombre d'octets recopiés. Les em-

2. NdT : Le terme original est *spurious*.

placements libérés sont maintenant disponibles pour écrire, donc piperead appelle wakeup [6868] pour réveiller tous les écrivains en attente avant de se terminer. La fonction wakeup trouve un processus en attente sur &p->nwrite, celui qui avait appelé pipewrite mais qui avait dû s'arrêter lorsque le tampon est devenu plein ; elle le marque comme RUNNABLE.

Le code des tubes utilise des canaux d'attente distincts pour le lecteur et l'écrivain (p->nread et p->nwrite) ce qui a pour but de rendre le système plus efficace dans le cas peu probable où il y a beaucoup de lecteurs et d'écrivains en attente pour le même tube. Ce code appelle sleep à l'intérieur de la boucle qui teste la condition ; s'il y a de multiples lecteurs ou écrivains, tous les processus à l'exception du premier réveillé verront que la condition est toujours fausse et se rendormiront.

5.8 Code : wait, exit et kill

Les fonctions sleep et wakeup peuvent être utilisées dans de nombreuses catégories d'attentes. Nous avons vu un exemple intéressant dans le chapitre 0 (voir 0.1 page 11) : l'appel système wait est utilisé par un processus parent pour attendre la terminaison d'un fils. Lorsque le fils se termine, il ne disparaît pas immédiatement. À la place, le processus passe dans l'état ZOMBIE jusqu'à ce que le parent appelle wait pour être informé de la terminaison. Le parent est ensuite responsable de la libération de la mémoire associée au processus et de la préparation de la struct proc pour une réutilisation ultérieure. Si le parent se termine avant le fils, le processus init adopte le processus fils et attend sa terminaison, de façon que chaque fils ait un parent pour nettoyer les lieux après sa terminaison.

Les problèmes de concurrence entre wait par le parent et exit par le fils, comme entre exit et exit, constituent une difficulté d'implémentation. La fonction wait commence par verrouiller ptable.lock, puis elle parcourt la table des processus à la recherche d'un fils. Si elle détecte que le processus courant a des fils, mais qu'aucun n'est terminé, elle appelle sleep pour attendre la terminaison de l'un d'entre eux [2707], puis elle recommence le parcours. Ici, le verrou libéré dans sleep est ptable.lock, il s'agit du cas spécial que nous avons déjà examiné.

La fonction exit verrouille ptable.lock, puis réveille tous les processus endormis sur le canal d'attente correspondant à l'entrée du parent du processus courant dans la table des processus [2651] ; s'il y a un tel processus en sommeil, il sera le parent dans wait. Ceci peut sembler prématuré, puisque exit n'a pas encore marqué le processus courant comme ZOMBIE. Cependant, il n'y a aucun risque : bien que wakeup puisse provoquer l'exécution du parent, la boucle dans wait ne peut pas reprendre avant que exit libère ptable.lock en appelant sched pour rentrer dans l'ordonnanceur, donc wait ne peut pas accéder au processus en cours de terminaison avant que exit ait mis son état à ZOMBIE [2663]. Avant que exit ne libère le processeur, cette fonction doit parcourir tous les fils du processus courant pour changer leur parent à initproc [2653-2660]. Finalement, exit appelle sched pour abandonner le processeur.

Si le processus parent était endormi dans wait, l'ordonnanceur finira par le remettre en exécution. L'appel à sleep se termine en ayant reverrouillé ptable.lock ; la fonction wait parcourt à nouveau la table des processus et détecte le fils terminé avec state == ZOMBIE [2657]. Elle mémorise le pid du fils, puis nettoie la struct proc, libérant la mémoire associée au processus [2687-2694].

Le processus fils pourrait faire l'essentiel du nettoyage dans exit, mais il faut que le parent soit celui qui libère p->kstack et p->pgdir : quand le fils exécute exit, sa pile est dans la mémoire désignée par p->kstack et il utilise sa propre table des pages. Celles-ci ne peuvent être libérées que lorsque le fils aura terminé son exécution pour la dernière fois en appelant swtch (via sched). C'est l'une des raisons pour lesquelles l'ordonnanceur utilise sa propre

pile plutôt que la pile du thread ayant appelé `sched`.

Alors que `exit` permet à un processus de se terminer lui-même, `kill` [2975] permet à un processus de demander qu'un autre se termine. Il serait trop complexe de laisser `kill` détruire directement le processus victime, puisqu'il peut être en cours d'exécution sur un autre processeur ou endormi au milieu d'une mise à jour de structures de données du noyau. Pour relever ces défis, `kill` en fait très peu : cette fonction positionne le `p->killed` du processus victime et, s'il est endormi, le réveille. Le processus victime finira par entrer ou sortir du noyau, auquel cas le code de `trap` appellera `exit` si `p->killed` est non nul. Si le processus victime est en cours d'exécution en mode utilisateur, il entrera bientôt dans le noyau en faisant un appel système ou parce que l'horloge (ou un autre périphérique) signalera une interruption.

Si le processus victime est dans `sleep`, l'appel à `wakeup` provoquera la sortie de `sleep`. C'est potentiellement dangereux car la condition d'attente peut être encore vraie. Cependant, les appels à `sleep` dans xv6 sont toujours englobés dans une boucle `while` qui teste à nouveau la condition après un retour de `sleep`. Certains appels à `sleep` testent également `p->killed` dans la boucle et abandonnent l'activité en cours si la valeur est non nulle. C'est le cas seulement lorsqu'un tel abandon est correct. Par exemple, le code de lecture et d'écriture dans les tubes [6837] se termine si `p->killed` est non nul. Le code finira par se terminer et revenir dans `trap`, qui vérifiera à nouveau `p->killed` et provoquera la terminaison.

Certaines boucles autour de `sleep` dans xv6 ne testent pas `p->killed` car le code est au milieu d'un appel système en plusieurs étapes qui devrait être atomique. Le pilote IDE [4379] est un exemple : il ne teste pas `p->killed` car une opération disque peut être l'une des écritures qui sont toutes nécessaires afin que le système de fichiers reste dans un état correct. Pour éviter la complexité d'un nettoyage après une opération partiellement terminée, xv6 retarde la terminaison d'un processus qui est dans le pilote IDE jusqu'à un point ultérieur lorsqu'il est plus facile de terminer le processus (c'est-à-dire lorsque l'opération sur le système de fichiers est totalement terminée et que le processus est sur le point de revenir en mode utilisateur).

5.9 Le monde réel

L'ordonnanceur de xv6 implémente une politique d'ordonnancement simple, qui donne la main à chaque processus à son tour, nommée *tourniquet*[3]. Les systèmes d'exploitation réels implémentent des politiques plus sophistiquées permettant par exemple d'attribuer des priorités aux processus. L'idée est qu'un processus prêt à tourner avec une priorité élevée sera choisi par l'ordonnanceur de préférence à un processus prêt à tourner avec une priorité faible. Ces politiques peuvent rapidement devenir complexes car il y a souvent des objectifs contradictoires : par exemple, le système pourrait vouloir garantir l'équité en même temps qu'une grande capacité. De plus, des politiques complexes peuvent conduire à des interactions inattendues comme l'*inversion de priorité* et les *convois*. L'inversion de priorité peut survenir lorsqu'un processus de faible priorité et un processus de priorité élevée partagent un verrou qui, lorsqu'il est obtenu par le processus de faible priorité, empêche celui de priorité élevée de progresser. Un long convoi peut se former si beaucoup de processus de priorité élevée attendent un processus de faible priorité qui a déjà obtenu le verrou partagé ; une fois le convoi formé, il peut persister pendant un long moment. Pour éviter ce genre de problèmes, des mécanismes supplémentaires doivent être ajoutés dans les ordonnanceurs sophistiqués.

Les fonctions `sleep` et `wakeup` constituent une méthode de synchronisation simple et effective, mais il en existe beaucoup d'autres. Le premier défi pour toutes les méthodes est d'éviter le problème des « réveils perdus » que nous avons vu au début de ce chapitre. La fonction

3. NdT : Le terme original est *round robin*.

`sleep` des noyaux Unix originaux désactivait simplement les interruptions, ce qui suffisait car Unix tournait sur des ordinateurs monoprocesseurs. Puisque xv6 fonctionne sur des multiprocesseurs, il ajoute un verrou explicite à `sleep`. La fonction `msleep` de FreeBSD a adopté la même approche. La fonction `sleep` de Plan9 utilise une fonction de *callback*[4] appelée avec le verrou de l'ordonnanceur juste avant l'endormissement ; cette fonction sert de test de dernière minute pour la condition d'attente, pour éviter les réveils perdus. La fonction `sleep` de Linux utilise une file de processus à la place d'un canal d'attente ; cette file dispose de son propre verrou interne.

Parcourir la liste complète des processus dans `wakeup` pour trouver ceux qui ont le bon canal d'attente est inefficace. Une meilleure solution serait de remplacer le canal dans `sleep` et `wakeup` par une structure de données mémorisant la liste des processus endormis sur cette structure. Les fonctions `sleep` et `wakeup` de Plan9 appellent cette structure un point de rendez-vous[5] (`Rendez`). De nombreuses bibliothèques de threads utilisent la même structure sous le nom de « variable condition » ; dans ce contexte, les opérations `sleep` et `wakeup` sont appelées `wait` et `signal`. Tous ces mécanismes partagent le même style : la condition d'attente est protégée par un genre de verrou abandonné de manière atomique durant l'attente.

L'implémentation de `wakeup` réveille tous les processus attendant sur un canal particulier : il se peut que de nombreux processus attendent sur ce canal. Le système d'exploitation ordonnancera tous ces processus et ils entreront en compétition pour tester la condition d'attente. Les processus qui se comportent de cette manière sont parfois appelés un « *troupeau en délire* », et il vaut mieux éviter cette situation. La plupart des « variables conditions » ont deux primitives pour le réveil : `signal` pour réveiller un processus, et `broadcast` pour réveiller tous les processus en attente.

Les sémaphores sont un autre mécanisme de coordination répandu. Un sémaphore est une valeur entière avec deux opérations, l'incrémentation et la décrémentation. Il est toujours possible d'incrémenter un sémaphore, mais la valeur n'est pas autorisée à descendre en dessous de zéro : la décrémentation d'un sémaphore nul provoque l'attente jusqu'à ce qu'un autre processus l'incrémente, et ces deux opérations s'annulent. La valeur entière correspond typiquement à un compteur réel, tel qu'un nombre d'octets disponibles dans un tampon de tube ou le nombre de fils zombies d'un processus. Utiliser un compteur explicite dans le cadre de l'abstraction évite le problème de « réveil perdu » : on compte explicitement le nombre de réveils. Le compteur évite également les réveils infondés et les troupeaux en délire.

La terminaison des processus et leur nettoyage introduit beaucoup de complexité dans xv6. Dans la plupart des systèmes d'exploitation, c'est encore plus complexe parce que, par exemple, le processus victime peut être endormi dans un endroit profond du noyau, et dévider sa pile nécessite une programmation très soignée. La plupart des systèmes dévident la pile en utilisation des mécanismes explicites pour la gestion des exceptions, comme `longjmp`. De plus, d'autres événements peuvent provoquer le réveil d'un processus endormi, même si l'événement attendu n'est pas encore arrivé. Par exemple, lorsqu'un processus Unix dort, un autre processus peut lui envoyer un signal. Dans ce cas, le processus reviendra de l'appel système interrompu avec la valeur -1 et et le numéro d'erreur initialisé à `EINTR`. L'application peut tester ces valeurs et décider de l'action à effectuer. Xv6 ne supporte pas les signaux et cette complexité ne se présente pas.

Le support de `kill` par xv6 n'est pas totalement satisfaisant : certaines boucles avec `sleep` devraient probablement tester `p->killed`. Un problème connexe est que, même pour des boucles avec `sleep` testant `p->killed`, il y a une compétition entre `sleep` et `kill` ; cette dernière peut positionner `p->killed` et tenter de réveiller le processus victime juste après

4. NdT : Nous avons choisi de ne pas traduire ce terme, très répandu.
5. NdT : En Français dans le texte.

le test de `p->killed` et avant l'appel à `sleep`. Si ce problème survient, le processus victime ne verra pas `p->killed` jusqu'à ce que la condition sur laquelle il attend survienne, ce qui peut arriver bien plus tard (par exemple lorsque le pilote IDE a récupéré le bloc disque que la victime attendait) voire jamais (par exemple si la victime attend une saisie sur la console mais que l'utilisateur ne tape rien).

5.10 Exercices

1. La fonction `sleep` doit tester `lk != &ptable.lock` pour éviter un interblocage [2890-2893]. Supposez que le cas spécial soit éliminé en remplaçant

   ```
   if(lk != &ptable.lock){
     acquire(&ptable.lock);
     release(lk);
   }
   ```

   par :

   ```
   release(lk);
   acquire(&ptable.lock);
   ```

   Est-ce que cela serait correct ? Pourquoi ?
2. La plus grande partie partie du nettoyage de processus pourrait être effectuée soit par `exit`, soit par `wait`, mais nous avons vu que `exit` ne doit pas libérer `p->stack`. Il s'avère que `exit` est l'endroit où sont doivent être fermés les fichiers ouverts. Pourquoi ? La réponse passe par les tubes.
3. implémentez les sémaphores dans xv6. Vous pouvez utiliser des verrous d'exclusion mutuelle, mais n'utilisez pas `sleep` et `wakeup`. Remplacez les utilisations de `sleep` et `wakeup` dans xv6 par des sémaphores. Évaluez le résultat.
4. Corrigez la compétition mentionnée dans le texte entre `kill` et `sleep`, de façon que l'appel système en cours soit abandonné si `kill` arrive, dans la boucle d'endormissement de la victime, après le test de `p->killed` et avant l'appel à `sleep`.
5. Imaginez un système pour que chaque boucle d'endormissement teste `p->killed` afin que, par exemple, si un processus est dans le pilote IDE, il puisse revenir rapidement de sa boucle `while` si un autre processus utilise `kill` pour terminer ce processus.
6. Imaginez un scénario qui utilise une seule commutation de contexte lors d'une commutation d'un processus vers un autre. Ce plan implique d'exécuter la procédure de l'ordonnanceur sur la pile noyau du processus utilisateur, au lieu d'une pile dédiée pour l'ordonnanceur. La principale difficulté est de nettoyer le processus correctement. Mesurez le gain de performances obtenu en évitant une des deux commutations de contexte.
7. Modifiez xv6 pour désactiver un processeur lorsqu'il est inutilisé, au lieu de juste boucler dans `scheduler`.
8. Le verrou `p->lock` protège de nombreux invariants, ce qui rend peu lisible la compréhension de l'invariant concerné par tel morceau de code. Concevez un schéma plus clair, sans doute en remplaçant l'unique `p->lock` par plusieurs verrous.

Chapitre 6

Système de fichiers

Le but d'un système de fichiers est d'organiser et de stocker les données. Les systèmes de fichiers supportent le partage des données entre les utilisateurs et les applications, ainsi que la *persistance* pour que les données restent toujours disponibles après un redémarrage.

Le système de fichiers de xv6 fournit des fichiers, des répertoires et des chemins comparables à ceux d'Unix (voir chapitre 0), et stocke ses données sur un disque IDE (voir chapitre 3) pour la persistance. Le système de fichiers doit relever plusieurs défis.

— Il faut des structures de données sur le disque pour représenter l'arborescence des noms de fichiers et de répertoires, pour enregistrer l'identité des blocs qui forment le contenu de chaque fichier, et pour savoir quelles sont les zones libres du disque.
— Le système de fichiers doit supporter la *récupération après sinistre*[1] : si un crash (par exemple une coupure électrique) survient, le système de fichiers doit toujours fonctionner correctement après le redémarrage. Le risque est qu'un crash pourrait interrompre une séquence de mises à jour et laisser les structures de données sur le disque dans un état incohérent (par exemple un bloc à la fois utilisé dans un fichier et marqué comme étant libre).
— Plusieurs processus distincts pouvant simultanément agir sur le système de fichiers, il faut des synchronisations pour maintenir les invariants.
— Les accès au disque étant plusieurs ordres de grandeur plus lents que les accès à la mémoire, le système de fichiers doit conserver les blocs populaires dans un cache en mémoire.

Le reste de ce chapitre explique comment xv6 résout ces défis.

6.1 Présentation

L'implémentation du système de fichiers de xv6 est organisé en 7 couches comme illustré sur la figure 6.1. La couche « disque » lit et écrit des blocs sur le disque dur IDE. La couche « buffer cache » met en cache les blocs du disque et synchronise les accès à ces blocs en garantissant qu'un seul processus en mode noyau à la fois peut modifier les données d'un bloc. La couche « journalisation » (*logging*) permet aux couches supérieures de réunir les mises à jour de plusieurs blocs dans une unique *transaction*, et garantit que les blocs sont mis à jour de manière atomique même en cas de crash (c'est-à-dire que tous les blocs sont mis à jour, ou aucun). La couche « inode » fournit des fichiers individuels, chacun représenté comme un *inode* doté d'un numéro unique et de blocs contenant les données du fichier. La couche « répertoire » (*di-*

1. NdT : Le terme original est *crash recovery*.

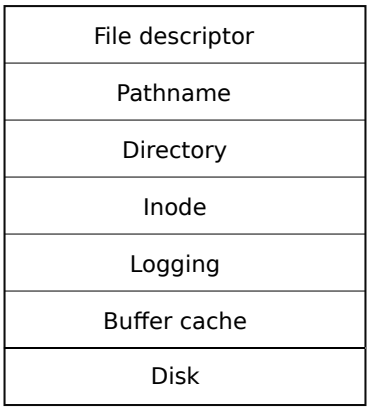

FIGURE 6.1 – Les couches du système de fichiers de xv6.

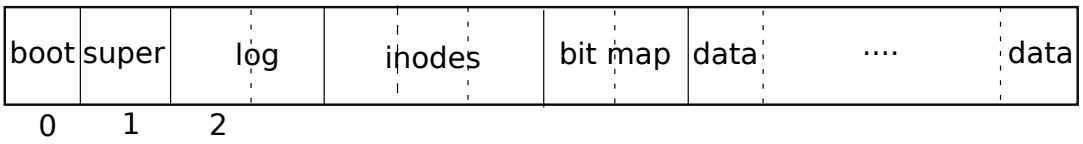

FIGURE 6.2 – Structure du système de fichiers de xv6. Le fichier `fs.h` [4050] contient les constantes et les structures de données décrivant l'agencement exact du système de fichiers.

rectory) implémente chaque répertoire comme une catégorie spéciale d'inode dont le contenu est une suite d'entrées de répertoires, chacune contenant un nom de fichier et un numéro d'inode. La couche « chemin » (*pathname*) fournit des noms de chemins hiérarchiques comme `/usr/rtm/xv6/fs.c` et résout ces noms grâce à un parcours récursif. La couche « descripteur de fichier » abstrait nombre de ressources Unix (par exemple : tubes, périphériques, fichiers, etc.) en utilisant l'interface du système de fichiers, simplifiant ainsi la vie des programmeurs d'applications.

Le système de fichiers doit déterminer où il place les inodes et les blocs de contenu sur le disque. Pour ce faire, xv6 divise le disque en plusieurs sections, comme le montre la figure 6.2. Le système de fichiers n'utilise pas le bloc 0 (il contient le secteur d'amorçage). Le bloc 1 est appelé le *superbloc* ; il contient des méta-données sur le système de fichiers (sa taille en nombre de blocs, les nombres de blocs de données et d'inodes, et le nombre de blocs dans le journal). Le journal commence au bloc 2. Les inodes sont situés après le journal, avec plusieurs inodes par bloc. Après cela viennent des blocs de *bitmap* pour identifier les blocs de données utilisés. Les blocs restants sont les blocs de données ; chaque bloc est soit marqué comme libre dans la bitmap, soit utilisé pour le contenu d'un fichier ou d'un répertoire. Le superbloc est rempli par un programme séparé appelé `mkfs`, qui construit un système de fichier initial.

Le reste de ce chapitre décrit chaque couche, en partant du buffer cache. Soyez attentif pour identifier les cas où des abstractions bien choisies dans des couches basses facilitent la conception des couches supérieures.

6.2 La couche du buffer cache

Le buffer cache a deux missions :

1. synchroniser les accès aux blocs du disque pour garantir qu'une seule copie d'un bloc

est en mémoire et qu'au maximum un seul thread noyau utilise cette copie ;

2. conserver les blocs populaires en mémoire afin qu'il n'y ait pas besoin de les relire à partir du disque lent.

Le code est dans `bio.c`.

La principale interface exportée par le buffer cache correspond aux fonctions `bread` et `bwrite` ; la première renvoie un buffer contenant la copie d'un bloc qui peut être lu ou modifié en mémoire, et la deuxième écrit un buffer modifié sur le disque dans le bloc approprié. Lorsqu'il a terminé d'utiliser un buffer, un thread noyau doit le libérer en appelant `brelse`. Le buffer cache utilise un verrou passif par buffer pour garantir que chaque buffer (et donc chaque bloc disque) ne peut être utilisé que par un seul thread à la fois ; `bread` renvoie un buffer verrouillé, et `brelse` libère le verrou.

Revenons au buffer cache. Il comprend un nombre fixé de buffers pour contenir les blocs de disque, ce qui signifie que si le système de fichiers demande un bloc qui n'est pas déjà dans le cache, le buffer cache doit recycler un buffer contenant actuellement un autre bloc. Pour ce nouveau bloc, le buffer cache recycle le buffer le moins récemment utilisé. L'hypothèse est que ce buffer est le moins susceptible d'être utilisé dans un proche avenir.

6.3 Code : le buffer cache

Le buffer cache est une liste doublement chaînée de buffers. La fonction `binit`, appelée par `main` [1230], initialise la liste avec les `NBUF` buffers du tableau statique `buf` [4450-4459]. Tous les autres accès au buffer cache utiliseront la liste chaînée via `bcache.head`, et non le tableau `buf`.

Un buffer dispose de deux bits pour représenter son état : `B_VALID` indique si le buffer contient une copie du bloc et `B_DIRTY` indique que le contenu du buffer a été modifié et doit être sauvegardé sur le disque.

La fonction `bread` [4502] appelle `bget` pour obtenir un buffer pour le bloc indiqué [4506]. Si le bloc doit être lu depuis le disque, `bread` appelle `iderw` pour le faire avant de renvoyer le buffer.

La fonction `bget` [4466] parcourt la liste des buffers pour chercher celui contenant le périphérique et le numéro de bloc indiqués [4472-4480]. S'il est trouvé, `bget` obtient le verrou passif pour le buffer, et renvoie le buffer verrouillé.

S'il n'y a pas de buffer dans le cache pour le numéro de bloc indiqué, `bget` doit en construire un, probablement en réutilisant un buffer contenant un bloc différent. La fonction parcourt la liste des buffers une seconde fois pour en chercher un qui ne soit ni verrouillé, ni modifié : tout buffer correspondant à ces deux critères peut être réutilisé. Ensuite, `bget` modifie les métadonnées du buffer pour enregistrer le nouveau périphérique et le nouveau numéro de bloc, puis elle obtient le verrou passif. Notez que l'affectation à `flags` efface le bit `B_VALID`, donc garantit que `bread` lira le bloc à partir du disque et n'utilisera pas les anciennes données du buffer.

Il est important qu'il y ait au plus un buffer dans le cache par bloc du disque pour garantir que les lecteurs voient les écritures, et parce que le système de fichiers utilise des verrous sur les buffers pour la synchronisation. La fonction `bget` garantit cet invariant en conservant le verrou `bcache.lock` continuellement à partir de la première boucle de test de présence dans le cache jusqu'à la déclaration, dans la deuxième boucle, que le bloc est maintenant en cache (en positionnant `dev`, `blockno`, et `refcnt`). Ce verrouillage rend atomique le test de la présence du bloc et, s'il n'est pas présent, la désignation d'un buffer pour contenir le bloc.

Il n'y a aucun danger lorsque `bget` obtient le verrou du buffer en dehors de la section critique protégée par `bcache.lock`, car la valeur non nulle de `b->refcnt` empêche le buffer d'être réutilisé pour un bloc différent. Le verrou passif par buffer protège les lectures et les écritures du contenu du bloc en mémoire, alors que `bcache.lock` protège les informations sur quels blocs sont dans le cache.

Si tous les buffers sont occupés, c'est qu'il y a trop de processus utilisant simultanément les fonctions du système de fichiers ; la fonction `bget` panique. Une réponse plus gracieuse serait de s'endormir jusqu'à ce qu'un buffer se libère, bien qu'il y aurait alors un risque d'interblocage.

Une fois que `bread` a lu le disque (si nécessaire) et a renvoyé le buffer à la fonction appelante, celle-ci dispose de l'utilisation exclusive du buffer et peut y lire ou écrire des octets de données. Si elle modifie le buffer, elle doit appeler `bwrite` pour écrire les données modifiées sur le disque avant de libérer le buffer. La fonction `bwrite` [4515] appelle `iderw` pour parler au contrôleur disque, après avoir mis le bit `B_DIRTY` pour indiquer que `iderw` doit écrire (et non lire).

Quand la fonction appelante a terminé avec un buffer, elle doit appeler `brelse` pour le libérer (le nom `brelse`, contraction de *b-release*, est cryptique mais vaut la peine d'être expliqué : il est issu d'Unix et est utilisé dans BSD, Linux, et Solaris également). Cette fonction `brelse` [4526] libère le verrou et déplace le buffer au devant de la liste chaînée [4537-4542]. Déplacer le buffer ainsi fait que la liste reste ordonnée suivant la plus récente utilisation (i.e. libération) du buffer : le premier buffer dans la liste est le plus récemment utilisé. Les deux boucles dans `bget` profitent de cet ordre : la recherche d'un buffer existant doit parcourir la liste entière dans le pire des cas, mais tester les buffers le plus récemment utilisés en premier (en démarrant depuis `bcache.head` et en suivant les pointeurs `next`) réduit la durée de la recherche s'il y a une bonne localité de références. La recherche pour trouver un buffer à réutiliser prend le moins récemment utilisé en parcourant la liste à l'envers (en suivant les pointeurs `prev`).

6.4 La couche de journalisation

Un des problèmes les plus intéressants dans la conception de systèmes de fichiers est la restauration après sinistre. Ce problème existe car nombre d'opérations du système de fichiers impliquent de multiples écritures sur le disque et un crash survenant après un sous-ensemble des écritures peut laisser le système de fichiers sur le disque dans un état incohérent. Par exemple, supposons qu'un crash survienne pendant la troncature d'un fichier (mettre la taille d'un fichier à zéro et libérer les blocs de contenu) : en fonction de l'ordre des écritures sur le disque, le crash peut soit laisser un inode avec une référence à un bloc de contenu marqué comme étant libre, ou il peut laisser un bloc de contenu alloué mais non référencé.

Ce dernier cas est relativement bénin, mais un inode qui référence un bloc libre posera vraisemblablement un sérieux problème après le redémarrage : le noyau pourrait allouer ce bloc à un autre fichier, et nous aurions alors deux fichiers différents référençant involontairement le même bloc. Si xv6 supportait plusieurs utilisateurs, cette situation pourrait constituer un problème de sécurité car le propriétaire de l'ancien fichier pourrait lire et écrire des blocs du nouveau fichier, appartenant à un autre utilisateur.

Xv6 résout le problème des crashes pendant les opérations du système de fichiers avec une forme simple de journalisation. Un appel système de xv6 n'écrit pas directement dans les structures de données du système de fichiers sur le disque. À la place, il construit une description de toutes les écritures sur disque qu'il souhaiterait faire dans un *journal* lui-même placé sur le disque. Une fois que l'appel système a enregistré toutes ses écritures, il écrit un enregistrement

spécial « *commit* »[2] sur le disque indiquant que le journal contient désormais une opération complète. À ce stade, l'appel système copie les écritures dans les structures de données du système de fichiers sur le disque. Une fois ces écritures terminées, l'appel système efface le journal sur le disque.

Si le système crashe et redémarre, le code du système de fichiers se rétablit comme suit, avant d'exécuter tout processus. Si le journal indique contenir une opération complète, alors le code de restauration copie les écritures à l'endroit où elles étaient prévues sur le disque. Si le journal ne contient pas une opération complète, la restauration l'ignore. Le code de restauration termine en effaçant le journal.

Pourquoi le journal de xv6 résout-il le problème des crashes pendant les opérations du système de fichiers ? Si le crash survient avant le *commit* de l'opération, alors le journal sur disque n'est pas marqué comme complet et le code de restauration l'ignore, ce qui fait que le disque sera dans l'état comme si l'opération n'avait jamais démarré. Si le crash survient après le *commit* de l'opération, alors la restauration rejouera toutes les opérations d'écriture, peut-être en en répétant certaines si ces opérations avaient déjà commencé. Dans les deux cas, le journal rend les opérations atomiques par rapport aux crashes : après la restauration, soit toutes les opérations d'écriture apparaissent, soit aucune n'apparaît.

6.5 Conception du journal

Le journal est placé à un endroit connu, fixe et spécifié dans le superbloc. Il consiste en un bloc d'en-tête suivi par une suite de copies de blocs modifiés (les blocs *journalisés*). Le bloc d'en-tête contient un tableau de numéros de blocs, un pour chacun des blocs journalisés, et le nombre de blocs actuellement dans le journal. Ce nombre de blocs dans l'en-tête est soit nul, indiquant qu'il n'y a aucune transaction dans le journal, soit non nul, indiquant que le journal contient une transaction complète et *commitée* avec le nombre de blocs journalisés. Xv6 écrit le bloc d'en-tête lorsqu'une transaction est *commitée*, mais pas avant, et met le compteur à zéro après avoir copié les blocs journalisés dans le système de fichiers. Donc, un crash au milieu d'une transaction donnera un nombre nul dans le bloc d'en-tête du journal. Un crash après un *commit* donnera un nombre non nul.

Chaque code d'appel système indique le début et la fin d'une séquence d'écritures, qui doit être atomique par rapport aux crashes. Pour permettre l'exécution concurrente d'opérations du système de fichiers par différents processus, le système de journalisation doit accumuler les écritures de plusieurs appels système dans une seule transaction. Un seul *commit* peut donc comprendre les écritures de plusieurs appels système. Pour éviter d'éclater un appel système sur plusieurs transactions, le système de journalisation n'effectue un *commit* que si plus aucun appel système n'est en cours sur le système de fichiers.

L'idée de *commiter* plusieurs transactions ensemble est connue sous le nom *commit de groupe*. Cela réduit le nombre d'opérations sur le disque en répartissant le coût fixe d'un *commit* sur plusieurs opérations. Le *commit* de groupe augmente également la concurrence des écritures sur disque, permettant peut-être de toutes les réaliser en une seule rotation du disque. Le pilote IDE de xv6 ne supporte pas ce *groupage*[3], mais la conception du système de fichiers le permettrait.

Xv6 dédie un espace fixe sur le disque pour le journal. Le nombre total de blocs écrits par les appels système dans une transaction doit tenir dans cet espace. Ceci a deux conséquences. Aucun appel système ne peut être autorisé à écrire davantage de blocs différents qu'il n'y a

2. NdT : Le terme original est laissé, faute de traduction appropriée.
3. NdT : Le terme original est *batching*.

de place dans le journal. Ce n'est pas un problème pour la plupart des appels système, mais deux d'entre eux peuvent potentiellement écrire de nombreux blocs : `write` et `unlink`. Une grande écriture dans un fichier peut écrire de nombreux blocs de données et de bitmap ainsi qu'un bloc d'inode ; supprimer un grand fichier peut écrire de nombreux blocs de bitmap et un inode. L'appel système `write` de xv6 découpe les grandes écritures en plusieurs plus petites qui tiennent dans le journal, et `unlink` ne pose pas de problème car, en pratique, le système de fichiers de xv6 utilise seulement un bloc de bitmap. L'autre conséquence de la limitation de taille du journal est que le système de journalisation ne peut pas autoriser un appel système à démarrer s'il n'est pas certain que les écritures de l'appel système tiennent dans l'espace restant dans le journal.

6.6 Code : journalisation

Dans un appel système, l'utilisation typique du journal ressemble à :

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

La fonction `begin_op` [4828] attend que le système de journalisation ait fini de *commiter* et qu'il y ait suffisamment de place disponible dans le journal pour contenir les écritures de cet appel système. La variable `log.outstanding` compte le nombre d'appels système ayant réservé de l'espace dans le journal ; l'espace total réservé est `log.outstanding` × `MAXOPBLOCKS`. Augmenter `log.outstanding` a comme effet à la fois de réserver de l'espace et d'empêcher un *commit* d'être réalisé pendant l'appel système. Le code suppose prudemment que chaque appel système pourrait écrire jusqu'à `MAXOPBLOCKS` blocs différents.

La fonction `log_write` [4922] agit comme un intermédiaire avec `bwrite`. Elle enregistre le numéro de bloc en mémoire, lui réserve un emplacement dans le journal sur le disque et marque le buffer `B_DIRTY` pour empêcher le cache de l'évincer. Le bloc doit rester dans le cache jusqu'à ce qu'il soit *commité* : d'ici là, la copie en cache est le seul enregistrement de la modification, le bloc ne peut pas être copié à sa place sur le disque avant le *commit*, et enfin les autres lecteurs dans la même transaction doivent voir les modifications. La fonction `log_write` détecte lorsqu'un bloc est écrit plusieurs fois dans la même transaction, et alloue à ce bloc le même emplacement dans le journal. Cette optimisation est souvent appelée *absorption*. Il est courant, par exemple, qu'un bloc du disque contenant les inodes de plusieurs fichiers soit écrit plusieurs fois au cours de la même transaction. En absorbant plusieurs écritures disques en une seule, le système de fichiers économise de l'espace dans le journal et obtient de meilleures performances puisqu'une seule copie du bloc doit être écrite sur le disque.

La fonction `end_op` [4853] décrémente en premier le nombre d'appels système en suspens. Si ce nombre devient nul, elle *commite* la transaction en appelant `commit`, qui fonctionne en quatre étapes :

1. `write_log` [4885] copie chaque bloc modifié dans la transaction depuis le buffer cache vers son emplacement dans le journal sur le disque ;
2. `write_head` [4804] écrit le bloc d'en-tête sur le disque ;
3. `install_trans` [4821] lit chaque bloc depuis le journal et l'écrit à sa place dans le système de fichiers ;

4. finalement, `end_op` écrit l'en-tête du journal avec un compteur nul.

Ceci doit être fait avant que la transaction suivante ne commence à écrire des blocs journalisés, afin qu'un crash n'entraîne pas une restauration en utilisant l'ancien en-tête de journal et les blocs journalisés de la transaction suivante.

La fonction `recover_from_log` [4818] est appelée par `initlog` [4756] lors du démarrage et avant l'exécution du premier processus utilisateur [2865]. Elle lit l'en-tête du journal, et reproduit les actions effectuées par `end_op` si l'en-tête indique que le journal contient une transaction *commitée*.

La fonction `filewrite` [6002] contient un exemple d'utilisation du journal. La transaction ressemble à :

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

Ce code est englobé dans une boucle qui, pour éviter de saturer le journal, découpe les grandes écritures dans des transactions individuelles de quelques blocs seulement. L'appel à `writei` écrit de nombreux blocs dans le cadre de cette transaction : l'inode du fichier, un ou plusieurs blocs de bitmap et des blocs de données.

6.7 Code : allocateur de blocs

Le contenu d'un fichier ou d'un répertoire est stocké dans des blocs sur le disque qui doivent être allouées à partir d'un réservoir de blocs libres. L'allocateur de blocs de xv6 maintient une bitmap des blocs libres sur le disque, avec un bit par bloc. Un bit à zéro indique que le bloc correspondant est disponible, un bit à un indique qu'il est utilisé. Le programme `mkfs` met à un les bits correspondant au secteur de boot, au superbloc, aux blocs du journal, aux blocs des inodes et aux blocs de la bitmap.

L'allocateur de blocs fournit deux fonctions : `balloc` alloue un nouveau bloc disque et `bfree` libère un bloc. Dans `balloc`, la boucle [5022] parcourt chaque bloc, à partir du bloc 0 jusqu'à `sb.size`, le nombre de blocs dans le système de fichiers. Elle cherche un bloc dont le bit vaut zéro dans la bitmap, indiquant qu'il est disponible. Si `balloc` trouve un tel bloc, elle met à jour la bitmap et renvoie le numéro du bloc. La boucle est découpée en deux parties pour plus d'efficacité : la boucle externe lit chaque bloc de la bitmap et la boucle interne teste les `BPB` bits d'un bloc individuel de la bitmap. La condition de concurrence qui pourrait se produire si deux processus tentaient en même temps d'allouer un bloc est évitée par le fait que le buffer cache ne permet qu'à un seul processus d'utiliser un bloc de bitmap à un instant donné.

La fonction `bfree` [5052] trouve le bloc de bitmap et met à zéro le bit approprié. À nouveau, l'exclusivité fournie par `bread` et `brelse` évite d'avoir recours à un verrouillage explicite.

Comme avec la plus grande partie du code décrit dans le reste du chapitre, `balloc` et `bfree` doivent être appelées à l'intérieur d'une transaction.

6.8 La couche des inodes

Le terme *inode* a deux significations : il peut faire référence à la structure de données sur le disque contenant la taille d'un fichier et la liste des numéros des blocs de données, ou il peut

faire référence à l'inode en mémoire qui abrite une copie de l'inode sur le disque ainsi que des informations supplémentaires nécessaires à l'intérieur du noyau.

Les inodes sur le disque sont compactés dans une zone contiguë du disque appellée les blocs d'inodes. Chaque inode a une taille identique, il est donc facile de trouver le n-ème inode sur le disque étant donné n. En fait, ce nombre n, appelé le numéro d'inode, correspond à la manière dont les inodes sont identifiés dans l'implémentation.

Les inodes sur le disque sont définis par une `struct dinode` [4078]. Le champ `type` différencie les fichiers, les répertoires et les fichiers spéciaux (périphériques). Un type nul indique que l'inode sur le disque est disponible. Le champ `nlink` comptabilise le nombre d'entrées de répertoires qui référencent cet inode, afin de détecter quand l'inode et ses blocs de données doivent être libérés Le champ `size` mémorise le nombre d'octets contenus dans le fichier. Le champ `addrs` est un tableau stockant les numéros des blocs sur le disque supportant le contenu du fichier.

Le noyau conserve l'ensemble des inodes actifs en mémoire ; la `struct inode` [4162] est la copie en mémoire d'une `struct dinode` sur le disque. Le noyau conserve un inode en mémoire uniquement s'il y a des pointeurs en C qui le référencent. Le champ `ref` comptabilise le nombre de pointeurs référençant cet inode en mémoire, et le noyau supprime l'inode de la mémoire si ce nombre devient nul. Les fonctions `iget` et `iput` obtiennent et libèrent des pointeurs sur un inode, en modifiant le nombre de références. Les pointeurs sur un inode peuvent provenir des descripteurs de fichiers, des répertoires courants, ou d'une utilisation temporaire par le noyau comme dans `exec`.

Il y a quatre mécanismes de verrouillage ou assimilés dans le code des inodes de xv6. Le verrou `icache.lock` protège les deux invariants « au maximum une seule copie de l'inode est présente dans le cache » et « le champ `ref` contient le nombre de pointeurs en mémoire vers cet inode dans le cache ». Chaque inode en mémoire contient un verrou passif qui garantit l'accès exclusif aux champs de l'inode (tels que la longueur du fichier) ainsi qu'aux blocs de données du fichier ou répertoire indiqué par l'inode. Le champ `ref` d'un inode, s'il est non nul, fait que le système conserve l'inode dans le cache et ne réutilise pas l'entrée pour un inode différent. Finalement, chaque inode contient un champ `nlink` (sur disque ou en mémoire) qui comptabilise le nombre d'entrées dans des répertoires référençant ce fichier ; xv6 ne libèrera pas l'inode si ce nombre de liens est plus grand que zéro.

Le pointeur vers une `struct inode` renvoyé par la fonction `iget` est garanti valide jusqu'à l'appel correspondant de `iput` : l'inode ne sera pas détruit et la mémoire référencée par le pointeur ne sera pas réutilisée pour un autre inode. La fonction `iget` fournit un accès non exclusif à un inode : il peut donc y avoir plusieurs pointeurs vers le même inode. De nombreuses parties du code du système de fichiers dépendent de ce comportement de `iget`, à la fois pour conserver des références aux inodes (tels que des fichiers ouverts ou des répertoires courants) sur une longue durée et pour prévenir des problèmes de concurrence en évitant les verrouillages lorsque le code manipule plusieurs inodes (comme par exemple lors de la recherche d'un chemin).

La `struct inode` renvoyée par `iget` peut ne pas avoir de contenu utilisable : pour garantir que l'inode contient bien la copie de l'inode sur le disque, il faut appeler `ilock`. Cette fonction verrouille l'inode afin qu'aucun autre processus ne puisse obtenir un `iget` dessus, puis lit l'inode depuis le disque si ça n'a pas déjà été fait. La fonction `iunlock` libère le verrou de l'inode. Séparer l'obtention d'un pointeur sur inode du verrouillage contribue à éviter les interblocages dans certains cas, comme par exemple lors de la recherche d'un chemin. De multiples processus peuvent avoir un pointeur C vers un inode retourné par `iget`, mais un seul peut verrouiller l'inode à la fois.

Le cache des inodes conserve uniquement les inodes référencés par des pointeurs C dans le code du noyau ou des structures de données. Sa mission principale est de synchroniser les accès par plusieurs processus, le fait de cacher les informations est secondaire. Si un inode est utilisé fréquemment, le buffer cache contiendra vraisemblablement le bloc concerné en mémoire s'il ne l'est pas par le cache des inodes. Celui-ci est en mode *write-through*[4], ce qui signifie que lorsque le code modifie un inode en cache, il doit l'écrire immédiatement avec `iupdate`.

6.9 Code : inodes

Pour allouer un nouvel inode, par exemple lors de la création d'un fichier, xv6 appelle `ialloc` [5204]. Cette fonction est similaire à `balloc` : elle parcourt les structures d'inode sur le disque, un bloc à la fois, en cherchant un inode marqué comme disponible. Lorsqu'elle en trouve un, elle le prend en écrivant une nouvelle valeur dans son champ `type` sur le disque et renvoie une entrée dans le cache avec l'appel terminal à `iget`[5218]. Le fonctionnement correct de `ialloc` dépend du fait qu'un seul processus à la fois peut obtenir une référence à `bp` : `ialloc` peut donc être sure qu'aucun autre processus ne peut voir en même temps l'inode disponible et tenter de le prendre.

La fonction `iget` [5254] parcourt le cache des inodes à la recherche de l'entrée active (`ip->ref > 0`) correspondant aux numéros de périphérique et d'inode désirés. Si l'entrée est trouvée, `iget` renvoie la nouvelle référence vers cet inode [5263-5267]. Pendant le parcours, `iget` enregistre la position du premier emplacement vide [5268-5269] qu'il utilisera au besoin pour allouer une entrée dans le cache.

Il faut verrouiller l'inode avec `ilock` [5303] avant de lire ou d'écrire ses métadonnées ou son contenu. Cette fonction utilise un verrou passif pour ce faire. Une fois qu'elle a obtenu l'accès exclusif, elle lit l'inode depuis le disque (plus probablement depuis le buffer cache) si besoin. La fonction `iunlock` [5331], quant à elle, libère le verrou passif, ce qui provoque le réveil des processus éventuellement en attente.

La fonction `iput` [5358] libère un pointeur C vers un inode en décrémentant le nombre de références [5376]. Si c'est la dernière référence, l'entrée de l'inode dans le cache est maintenant disponible et peut être réutilisée pour un autre inode.

Si `iput` constate qu'il n'y a aucune référence d'un pointeur C vers un inode et que cet inode n'est référencé par aucun lien (aucune occurrence dans un répertoire), alors l'inode et ses blocs de données peuvent être libérés. Cette fonction appelle `itrunc` pour tronquer le fichier à zéro octets, libérant ainsi les blocs de données, puis met le type de l'inode à 0 (disponible) et écrit enfin l'inode sur le disque [5366].

Le protocole de verrouillage dans `iput` mérite une attention particulière dans le cas où cette fonction libère l'inode. L'un des risques est qu'un thread concurrent pourrait être en attente dans `ilock` pour utiliser cet inode (par exemple pour lire le fichier ou lister le contenu du répertoire), et n'est pas préparé à trouver l'inode s'il n'est désormais plus alloué. Ceci ne peut arriver car un appel système ne peut obtenir un pointeur vers un inode dans le cache s'il n'y a pas de lien vers icelui et que `ip->ref` vaut 1. La seule et unique référence est celle obtenue par le thread ayant appelé `iput`. Il est vrai que `iput` vérifie que le nombre de références vaut 1 en dehors de sa section critique protégée par `icache.lock` mais, à ce stade, le nombre de liens vaut 0 donc aucun thread ne tentera d'obtenir une nouvelle référence. L'autre risque est qu'un appel concurrent à `ialloc` pourrait choisir le même inode que celui que `iput` est en train de libérer. Ceci pourrait survenir après que `iupdate` ait écrit sur le disque, donc lorsque le type est nul. Cette condition de concurrence est bénigne : le thread en cours d'allocation

4. NdT : Le terme original est conservé, faute de traduction communément admise.

attendra poliment l'obtention du verrou passif de l'inode avant de lire ou d'écrire l'inode et, à ce moment, `iput` en aura fini avec lui.

La fonction `iput` peut écrire sur le disque, ce qui signifie que tout appel système utilisant le système de fichiers peut écrire sur le disque puisque l'appel système peut être le dernier à avoir une référence sur le fichier. Même des appels comme `read`, qui sont apparemment en lecture seule, peuvent finir par appeler `iput`. Par conséquent, même des appels système ne faisant que lire doivent comporter des transactions s'ils utilisent le système de fichiers.

Il y a une difficulté dans l'interaction entre `iput` et les crashes. Cette fonction ne tronque pas le fichier immédiatement lorsque le nombre de liens devient nul, car un processus pourrait toujours avoir une référence à l'inode en mémoire : il pourrait toujours être en train de lire et d'écrire dans le fichier puisqu'il avait réussi à l'ouvrir. Mais si un crash survient avant que le dernier processus ait fermé le descripteur de fichier, le fichier sera marqué comme alloué sur le disque alors qu'aucune entrée de répertoire ne pointe sur lui.

Les systèmes de fichiers gèrent ce cas de deux manières. La solution simple est que, lors de la restauration après le redémarrage, le système de fichiers parcoure l'ensemble du disque en cherchant les fichiers marqués comme étant alloués et sans entrée de répertoire pointant dessus. Si de tels fichiers sont trouvés, le noyau peut les libérer.

La deuxième solution ne nécessite pas de parcourir le disque : il suffit que le système de fichiers enregistre sur le disque (par exemple dans le superbloc) le numéro d'inode d'un fichier dont le nombre de liens tombe à zéro et dont le nombre de références est non nul. Si le système de fichiers supprime le fichier quand son nombre de références tombe à son tour à zéro, le noyau met à jour la liste sur le disque en y supprimant le numéro d'inode. Lors de la restauration, le système de fichiers libère tous les fichiers de la liste.

Xv6 n'implémente aucune de ces deux solutions, ce qui signifie que des inodes peuvent rester alloués sur le disque même s'ils ne sont plus utilisés. Ceci signifie également que, sur la durée, xv6 court le risque de tomber à court d'espace disque.

6.10 Code : contenu des inodes

La structure d'un inode sur le disque, `struct dinode`, contient la taille et un tableau `addrs` de numéros de blocs (voir figure 6.3). Les données du fichier sont localisées dans les blocs listés dans ce tableau. Les `NDIRECT` premiers blocs de données occupent les `NDIRECT` premiers emplacement du tableau ; ces blocs sont appelés les *blocs directs*. Les `NINDIRECT` blocs de données suivants ne sont pas listés dans l'inode, mais dans un bloc de données appelé *bloc d'indirection*. La dernière entrée du tableau `addrs` donne l'adresse du bloc d'indirection. Ainsi, les 6 premiers kilo-octets (`NDIRECT`×`BSIZE`) d'un fichier peuvent être chargés à partir des blocs indiqués dans l'inode, alors que les 64 kilo-octets suivants (`NINDIRECT`×`BSIZE`) peuvent être chargés uniquement après consultation du bloc d'indirection. C'est une bonne représentation sur le disque, mais une représentation complexe pour les clients. La fonction `bmap` gère cette représentation de manière que les routines de plus haut niveau telles que `readi` et `writei`, que nous verrons prochainement [5], n'aient pas à s'en préoccuper. La fonction `bmap` renvoie le numéro du bloc sur le disque du `bn`-ème bloc de données de l'inode `ip`. Si `ip` n'a pas encore un tel bloc, `bmap` en alloue un.

La fonction `bmap` [5410] commence par traiter le cas facile : les `NDIRECT` premiers blocs sont listés dans l'inode lui-même [5415-5419]. Les `NINDIRECT` blocs suivants sont listés dans le bloc d'indirection situé à `ip->addrs[NDIRECT]`. La fonction le lit depuis le disque [5426],

5. NdT : La suite de cette phrase est absente du texte original (bogue), c'est donc un ajout du traducteur.

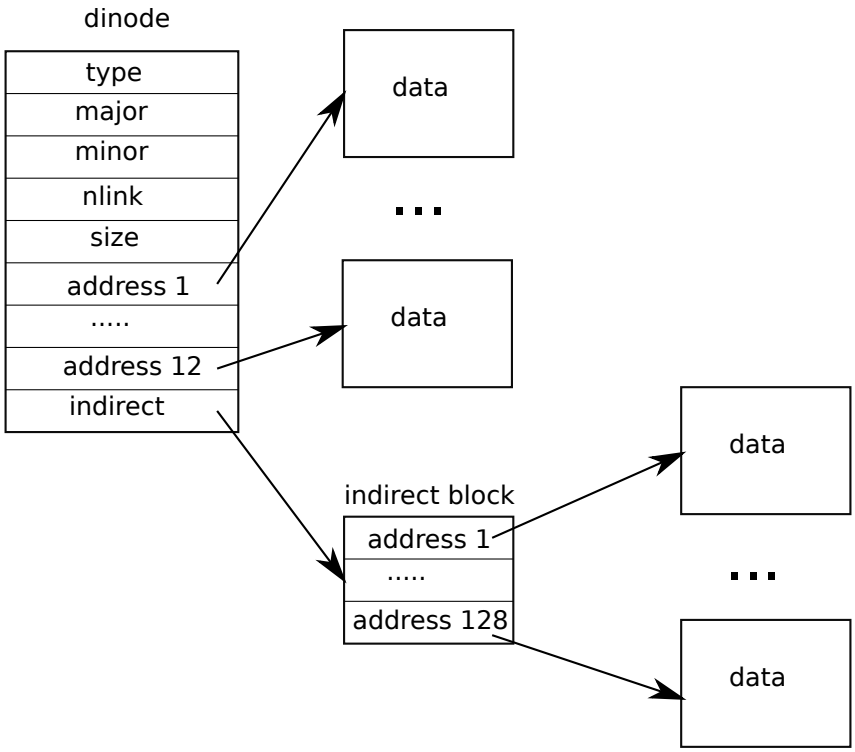

FIGURE 6.3 – Représentation d'un fichier sur le disque.

puis lit le numéro de bloc dans l'emplacement approprié [5427]. Si le numéro de bloc dépasse `NDIRECT+NINDIRECT`, `bmap` panique ; `writei` contient le test qui empêche ce débordement d'arriver [5566].

Cette fonction `bmap` alloue les blocs en cas de besoin. Un emplacement à zéro dans `ip->addrs[]` ou dans le bloc d'indirection indique qu'aucun bloc n'est alloué. Lorsque `bmap` rencontre un zéro, il le remplace par le numéro d'un bloc fraîchement alloué [5416-5417, 5424-5425].

La fonction `itrunc` [5456] libère les blocs d'un fichier et remet la taille dans l'inode à zéro. Elle commence par libérer les blocs directs [5462-5467], puis les blocs listés dans le bloc indirect [5472-5475] et enfin le bloc indirect lui-même [5477-5478].

La fonction `bmap` prépare le terrain pour que `readi` et `writei` récupèrent les données de l'inode. La fonction `readi` [5503] commence par s'assurer que l'offset plus le nombre d'octets à lire ne dépasse pas la fin du fichier. Les lectures qui démarrent au-delà de la fin du fichier renvoient une erreur [5514-5515] alors que les lectures qui démarrent à la fin du fichier ou qui la traversent renvoient moins d'octets que demandé [5516-5517]. La boucle principale traite chaque bloc du fichier, copiant les données du buffer dans `dst` [5519-5524]. La fonction `writei` [5553] est identique à `readi` avec trois exceptions : les écritures qui démarrent à la fin du fichier ou qui la traversent agrandissent le fichier, jusqu'à la taille maximum des fichiers [5566-5567] ; la boucle copie les données dans les buffers et non sur le disque [5572] ; et enfin, si l'écriture étend le fichier, `writei` doit mettre à jour sa taille [5577-5580].

Les deux fonctions `readi` et `write` commencent par tester `ip->type == T_DEV`. Ce test gère les fichiers spéciaux de type « périphérique » dont les données ne résident pas dans le système de fichiers ; nous reviendrons sur ce cas dans la couche des descripteurs de fichiers.

La fonction `stati` [5488] copie les métadonnées de l'inode dans la structure `stat` exposée aux programmes utilisateurs par l'appel système `stat`.

6.11 Code : la couche des répertoires

Un répertoire est semblable, en termes d'implémentation interne, à un fichier. L'inode a le type `T_DIR` et les données sont une suite d'entrées de répertoire. Chaque entrée est une `struct dirent` [4115] contenant un nom et un numéro d'inode. Le nom fait au maximum `DIRSIZ` (14) caractères ; s'il est plus court, il est terminé par un octet nul. Les entrées de répertoire dont le numéro d'inode est nul sont libres.

La fonction `dirlookup` [5611] recherche une entrée par son nom dans un répertoire. Si elle en trouve une, elle renvoie un pointeur, non verrouillé, vers l'inode correspondant et initialise `*poff` à l'offset (en octets) de l'entrée dans le répertoire pour le cas où l'appelante souhaiterait la modifier. Si `dirlookup` trouve l'entrée avec le bon nom, elle met à jour `*poff`, libère le bloc et renvoie l'inode non verrouillé obtenu par `iget`. La fonction `dirlookup` est la raison pour laquelle `iget` renvoie des inodes non verrouillés. La fonction appelante a verrouillé `dp`, donc si la recherche portait sur `".."`, l'alias du répertoire courant, la tentative de verrouiller l'inode avant de le renvoyer aurait comme effet de tenter de verrouiller à nouveau `dp` et conduirait inévitablement à un interblocage (il y a d'autres scénarios d'interblocage plus compliquées impliquant de multiples processus et `"..."`, l'alias du répertoire parent ; `"."` n'est pas le seul problème). La fonction appelante peut déverrouiller `dp` puis verrouiller `ip`, en faisant en sorte de ne conserver qu'un seul verrou à la fois.

La fonction `dirlink` [5652] écrit une nouvelle entrée dans le répertoire `dp` avec le nom et le numéro d'inode indiqués. Si le nom existe déjà, `dirlink` renvoie une erreur [5658-5662]. La boucle principale lit les entrées pour en trouver une disponible. Si c'est le cas, elle arrête la boucle prématurément [5622-5623] avec `off` initialisé à l'offset de l'entrée disponible. Sinon, la boucle se termine avec `off` initialisé à `dp->size`. Dans les deux cas, `dirlink` ajoute ensuite la nouvelle entrée dans le répertoire en écrivant à l'offset `off` [5672-5675].

6.12 Code : chemins

La recherche de chemin implique une succession d'appels à `dirlookup`, un pour chaque composant du chemin. La fonction `namei` [5790] évalue `path` et renvoie l'inode correspondant. La fonction `nameiparent` en est une variante : elle arrête avant le dernier élément, renvoie l'inode du répertoire parent et copie l'élément final dans `name`. Elles appellent toutes les deux `namex`, la fonction générale, pour faire réellement le travail.

Cette fonction `namex` [5755] commence par décider où l'évaluation du chemin doit débuter. Si le chemin commence par un slash, l'évaluation débute à la racine, sinon c'est à partir du répertoire courant [5759-5762]. Puis elle utilise `skipelem` pour prendre en compte chaque élément du chemin à son tour [5764]. Chaque itération de la boucle doit chercher `name` dans l'inode courant `ip`. L'itération commence par verrouiller `ip` et tester que c'est bien un répertoire. Dans le cas contraire, la recherche échoue [5765-5769] (le verrouillage de `ip` est nécessaire, non parce que `ip->type` peut changer sous nos pieds — il ne le peut pas — mais parce que jusqu'à ce que `ilock` soit exécuté, `ip->type` n'est pas forcément déjà chargé depuis le disque). Si la fonction appelante est `nameiparent` et qu'il s'agit du dernier élément du chemin, la boucle s'arrête prématurément conformément à la spécification de `nameiparent` ; l'élément final a déjà été copié dans `name`, donc `namex` doit juste renvoyer le pointeur `ip` non verrouillé [5770-5774]. Finalement, la boucle recherche l'élément du chemin en utilisant `dirlookup` et prépare l'itération suivante en modifiant `ip = next` [5775-5780]. Lorsque la boucle est à court d'éléments dans le chemin, la valeur `ip` est renvoyée.

La fonction `namex` peut durer longtemps car elle pourrait nécessiter plusieurs lectures d'inodes

et de répertoires sur le disque pour les répertoires traversés dans le chemin (s'ils ne sont pas dans le buffer cache). Xv6 est soigneusement conçu pour que si un appel à `named` par un thread noyau est endormi sur une entrée/sortie disque, un autre thread noyau puisse effectuer une recherche de chemin en même temps. La fonction `namex` verrouille chaque répertoire dans le chemin séparément de façon que les recherches de chemin puissent être effectuées en parallèle.

Cette concurrence introduit plusieurs difficultés. Par exemple, lorsqu'un thread noyau effectue la recherche d'un chemin alors qu'un autre modifie l'arborescence en supprimant un répertoire. Le risque potentiel est qu'il pourrait y avoir une recherche dans un répertoire supprimé par un autre thread noyau, et que les blocs de ce répertoire aient été réutilisés pour un autre fichier ou répertoire.

Xv6 évite de telles conditions de concurrence. Par exemple, lorsque `named` appelle `dirlookup`, le thread conserve le verrou sur le répertoire et `dirlookup` renvoie un inode obtenu par `iget`, qui incrémente le compteur de références de l'inode. La fonction `namex` ne libère le verrou sur le répertoire que lorsque `dirlookup` a renvoyé l'inode ; maintenant, un autre thread peut supprimer l'inode dans le répertoire, mais xv6 ne supprimera pas immédiatement l'inode puisque son compteur de références est toujours supérieur à zéro.

Un autre risque est l'interblocage. Par exemple, `next` pointe sur le même inode que `ip` lors d'une recherche de `".."`. Verrouiller `next` avant de libérer le verrou sur `ip` provoquerait un interblocage. Pour l'éviter, `namex` déverrouille le répertoire avant de verrouiller `next`. Ici encore, nous pouvons voir combien il est important de séparer `iget` et `ilock`.

6.13 La couche des descripteurs de fichiers

Un aspect agréable de l'interface Unix est que la plupart de ressources sont représentées comme des fichiers, y compris les périphériques comme la console, les tubes, et bien sûr les fichiers réels. La couche des descripteurs de fichiers est celle qui permet cette uniformisation.

Comme nous l'avons vu dans le chapitre 0 (voir 0.2, page 12), xv6 donne à chaque processus sa propre table des ouvertures de fichiers, ou descripteurs de fichiers. Chaque ouverture de fichier est représentée par une `struct file` [4150], qui est un emballage autour soit d'un inode, soit d'un tube, avec un offset d'entrée/sortie. Chaque appel à `open` crée une nouvelle ouverture de fichier (une nouvelle `struct file`) : si plusieurs processus ouvrent indépendamment le même fichier, les différentes ouvertures auront différents offsets. D'un autre côté, une ouverture unique (la même `struct file`) peut apparaître plusieurs fois dans la table d'un processus, ainsi que dans les tables de plusieurs processus : ceci peut arriver si un processus a utilisé `open` pour ouvrir un fichier et a ensuite créé un alias avec `dup`, ou partagé l'ouverture avec un fils en utilisant `fork`. Un compteur conserve le nombre de références à une ouverture particulière. Un fichier peut être ouvert en lecture, en écriture, ou dans les deux sens. Les champs `readable` et `writable` conservent cette information.

Toutes les ouvertures de fichiers dans le système sont conservées dans une table globale `ftable`. Celle-ci a une fonction pour l'allocation d'une ouverture (`filealloc`), la duplication d'une référence (`filedup`), la libération d'une référence (`fileclose`), et la lecture et l'écriture de données (`fileread` et `filewrite`).

Les trois premières suivent une forme maintenant familière. La fonction `filealloc` [5876] parcourt la table des ouvertures à la recherche d'une entrée non référencée (`f->ref == 0`) et renvoie une nouvelle référence, `filedup` [5902] incrémente le compteur de références et `fileclose` [5914] le décrémente. Lorsque le compteur de références d'une ouverture devient nul, `fileclose` libère, suivant son type, le tube ou l'inode sous-jacent.

Les fonctions `filestat`, `fileread`, et `filewrite` implémentent les opérations `stat`, `read`, et `write` sur les fichiers. La fonction `filestat` [5952] est autorisée uniquement sur les inodes et appelle `stati`. Les fonctions `fileread` et `filewrite` vérifient que l'opération est autorisée par le mode d'ouverture, puis passent l'appel à la fonction appropriée pour traiter le cas des tubes ou le cas des inodes. Si l'ouverture référence un inode, `fileread` et `filewrite` utilisent l'offset pour l'opération et l'avancent ensuite [5975-5976, 6015-6016], alors que les tubes n'ont pas de concept d'offset. Rappelez-vous que les fonctions sur les inodes attendent que l'appelant gère le verrouillage [5955-5957, 5974-5977, 6025-6028]. Ce verrouillage de l'inode présente l'effet de bord pratique que les offsets de lecture et d'écriture sont mis à jour de manière atomique : plusieurs écritures simultanées utilisant la même ouverture ne peuvent donc pas écraser les données des autres, bien que les écritures puissent à la fin être entrelacées.

6.14 Code : appels système

Avec les fonctions fournies par les couches inférieures, l'implémentation de la plupart des appels système est triviale (voir `sysfile.c`). Quelques appels système méritent toutefois une attention plus particulière.

Les fonctions `sys_link` et `sys_unlink` modifient des répertoires en créant ou supprimant des références à des inodes. Elles constituent un autre bel exemple de la puissance des transactions. La fonction `sys_link` [6202] commence par récupérer ses arguments, deux chaînes de caractères `old` et `new` [6207]. En supposant que `old` existe et n'est pas un répertoire [6211-6214], `sys_link` incrémente son compteur `ip->nlink`. Puis elle appelle `nameiparent` pour trouver le répertoire parent et l'élément final du chemin de `new` [6227], et crée une nouvelle entrée de répertoire pointant sur l'inode de `old` [6230]. Le répertoire parent de `new` doit exister et résider sur le même périphérique que l'inode existant : les numéros d'inode sont un identifiant unique seulement sur le même disque. Si une erreur comme celle-ci survient, `sys_link` doit revenir en arrière et décrémenter `ip->nlink`.

Les transactions simplifient l'implémentation car il faut mettre à à jour plusieurs blocs sur le disque et nous n'avons pas à nous inquiéter de l'ordre dans lequel nous faisons ces mises à jour. Par exemple, sans les transactions, incrémenter `ip->nlink` avant de créer le lien mettrait temporairement le système de fichiers dans un état non cohérent, et un crash au milieu serait un désastre. Avec les transactions, nous n'avons pas ce souci.

La fonction `sys_link` crée un nouveau nom pour un inode existant. La fonction `create` [6357] crée également un nouveau nom mais pour un nouvel inode. C'est la généralisation de trois appels système de création de fichier : `open` avec le flag `O_CREATE` pour créer un fichier ordinaire, `mkdir` pour un nouveau répertoire et `mkdev` pour un nouveau fichier périphérique. Comme `sys_link`, `create` commence par appeler `nameiparent` pour récupérer l'inode du répertoire parent, puis appelle `dirlookup` pour vérifier si le nom existe déjà [6366]. Si c'est le cas, le comportement de `create` dépend de quel appel système l'a appelée : `open` a une sémantique différente de `mkdir` et `mkdev`. Si `create` est utilisée pour le compte de `open` (`type == T_FILE`) et que le nom existant est lui-même un fichier régulier, alors `open` considère que c'est un succès, donc `create` doit faire de même [6370]. Sinon, elle renvoie une erreur [6371-6372]. Si le nom n'existe pas déjà, `create` alloue maintenant un nouvel inode avec `ialloc` [6375]. Si ce nouvel inode doit être un répertoire, `create` l'initialise avec les entrées `.` et `..`. Finalement, maintenant que ses données sont correctement initialisées, `create` peut lier l'inode dans le répertoire parent [6388]. Comme `sys_link`, `create` a deux inodes simultanément verrouillés : `ip` et `dp`. Il n'y a pas de risque d'interblocage car l'inode `ip` vient d'être alloué : aucun autre processus dans le système ne peut verrouiller `ip` puis tenter de verrouiller `dp`.

En utilisant `create`, il est facile d'implémenter `sys_open`, `sys_mkdir`, et `sys_mknod`. La fonction `sys_open` [6401] est la plus complexe car créer un nouveau fichier est la partie la plus petite de ce qu'elle doit faire. Si le flag `O_CREATE` est passé à `open`, elle appelle `create` [6414]. Autrement, elle appelle `namei` [6420]. La fonction `create` renvoie un inode verrouillé, mais `namei` ne le fait pas, donc `sys_open` doit verrouiller elle-même l'inode. Ceci donne un endroit pratique pour vérifier que les répertoires sont ouverts uniquement en lecture et pas en écriture. En supposant que l'inode ait été obtenu d'une manière ou de l'autre, `sys_open` alloue une ouverture de fichier et un descripteur [6432] puis initialise l'ouverture [6442-6446]. Notons qu'aucun autre processus ne peut accéder à l'entrée puisqu'elle figure uniquement dans le descripteur du processus courant.

Le chapitre 5 a détaillé l'implémentation des tubes (voir 5.7, page 74) avant même d'avoir vu le système de fichiers. La fonction `sys_pipe` connecte cette implémentation au système de fichiers en permettant de créer une paire de descripteurs. Son argument est un pointeur sur un espace de deux entiers où elle enregistrera les deux nouveaux descripteurs. Puis elle alloue le tube et installe les descripteurs.

6.15 Le monde réel

Les buffer caches dans les systèmes d'exploitation du monde réel sont significativement plus complexes que celui de xv6, mais ils ont les deux mêmes missions : cacher et synchroniser les accès au disque. Celui de xv6, comme celui d'Unix v6, utilise une politique simple d'éviction LRU (*least recently used*, ou moins récemment utilisé) ; nombre de politiques plus complexes peuvent être implémentées, chacune étant adaptée à un certain type de charge et pas à d'autres. Un cache LRU plus performant éliminerait la liste chaînée au profit d'une table de hachage pour les recherche et d'un tas pour les évictions LRU. Les buffer caches modernes sont typiquement intégrés dans le système de mémoire virtuelle pour supporter les fichiers projetés en mémoire.

Le système de journalisation de xv6 est peu performant. Un *commit* ne peut être réalisé en même temps qu'un appel système sur des fichiers. Le système journalise des blocs entiers, même si seulement quelques octets sont modifiés. Il écrit le journal de manière synchrone, un bloc à la fois, et chaque écriture va probablement durer le temps d'une rotation entière du disque. Les systèmes de journalisation réels résolvent tous ces problèmes.

La journalisation n'est pas la seule manière d'avoir une restauration après sinistre. Les premiers systèmes de fichiers utilisaient un réparateur lors du démarrage (par exemple, le programme Unix `fsck`) pour examiner chaque fichier et répertoire, ainsi que les listes de blocs et d'inodes libres, à la recherche les incohérences pour les résoudre. De tels parcours peuvent prendre des heures sur des grands systèmes de fichiers, et il y a des cas où il n'est pas possible de résoudre les incohérences de manière que les appels système originaux soient atomiques. La restauration depuis un journal est beaucoup plus rapide et fait que les appels système sont atomiques face à un crash.

Xv6 utilise le même format d'inode et de répertoire sur le disque que les premiers systèmes Unix ; ce schéma a été remarquablement persistant dans le temps. Le système UFS/FFS de BSD et les Ext2/3 de Linux utilisent essentiellement les mêmes structures de données. La partie la moins efficace du format du système de fichiers est le répertoire, qui nécessite un parcours linéaire de tous les blocs sur le disque à chaque recherche. C'est raisonnable lorsque les répertoires font seulement quelques blocs disques, mais c'est coûteux pour des répertoires contenant beaucoup de fichiers. Les systèmes NTFS de Microsoft Windows, HFS de MacOS X, et ZFS de Solaris, pour ne citer qu'eux, implémentent les répertoires comme des arbres équilibrés de blocs

sur le disque. C'est compliqué, mais garantit des recherche en temps logarithmique.

Xv6 est naïf à propos des erreurs de disque : si une opération disque échoue, xv6 panique. Savoir si c'est raisonnable dépend du matériel : si un système d'exploitation repose sur un matériel spécifique assurant la redondance pour masquer les erreurs de disque, peut-être que le système d'exploitation voit tellement rarement des échecs que le fait de paniquer soit suffisant.

D'un autre côté, les systèmes d'exploitation utilisant des disques simples devraient s'attendre à des échecs et mieux les gérer, de façon que la perte d'un bloc dans un fichier n'affecte pas le reste du système de fichiers.

Xv6 suppose que le système de fichiers tient sur un disque et ne change pas de taille. Comme les grandes bases de données et les fichiers multimédia imposent des exigences toujours plus grandes, les systèmes d'exploitation développent des approches pour éliminer le goulot d'étranglement de « un disque par système de fichiers ». L'approche de base est de combiner plusieurs disques en un seul disque logique. Des solutions matérielles comme le RAID sont toujours les plus populaires, mais la tendance actuelle est d'aller vers l'implémentation de cette logique en logiciel autant que possible. Ces implémentations logicielles ont typiquement de riches fonctionnalités comme l'agrandissement ou le rétrécissement de disques logiques en ajoutant ou supprimant des disques à chaud. Naturellement, une couche de stockage qui peut s'agrandir ou rétrécir à chaud nécessite un système de fichiers qui sache faire de même : la taille figée du tableau de blocs d'inodes utilisé par xv6 ne fonctionnerait pas bien dans de tels environnements. Séparer la gestion des disques du système de fichiers peut être la conception la plus propre, mais l'interface complexe entre les deux a conduit certains systèmes, comme ZFS de Sun, à les combiner ensemble.

Beaucoup d'autres caractéristiques des systèmes de fichiers modernes sont absentes de xv6 ; par exemple, il n'y a pas de support pour les instantanés [6] et les sauvegardes incrémentales.

Les systèmes Unix modernes permettent d'accéder à de nombreux types de ressources avec les mêmes appels système que pour le stockage sur disque : les tubes nommés, les connexions réseau, les systèmes de fichiers distants à travers le réseau, et les interfaces de surveillance et de contrôle telles que `/proc`. À la place des `if` de xv6 dans `fileread` et `filewrite`, ces systèmes attribuent typiquement à chaque ouverture de fichier une table de pointeurs sur des fonctions, une par opération, et utilisent le pointeur pour appeler l'implémentation de l'appel pour l'inode concerné. Les systèmes de fichiers distants ainsi que les systèmes de fichiers en mode utilisateur fournissent des fonctions qui vont transformer ces appels en appels à distance à travers le réseau et attendent la réponse avant de la renvoyer.

6.16 Exercices

1. Pourquoi paniquer dans `balloc` ? Est-ce que xv6 peut se sortir de la situation autrement ?
2. Pourquoi paniquer dans `ialloc` ? Est-ce que xv6 peut se sortir de la situation autrement ?
3. Pourquoi `filealloc` ne panique pas lorsqu'elle est à court d'emplacements ? Pourquoi ce cas est-il plus fréquent et donc est-il nécessaire de le gérer ?
4. Supposons que le fichier correspondant à `ip` soit supprimé par un autre processus entre les appels à `iunlock(ip)` et `dirlink` dans `sys_link`. Le lien sera-t-il créé correctement ? Expliquez pourquoi.

6. NdT : Le terme original est *snapshot*.

5. La fonction `create` fait quatre appels (un à `ialloc` et trois à `dirlink`) qui doivent tous réussir. Si l'un d'eux échoue, l'appel à `create` appelle `panic`. Pourquoi est-ce acceptable ? Pourquoi aucun de ces quatre appels ne doit échouer ?
6. La fonction `sys_chdir` appelle `iunlock(ip)` avant `iput(cp->cwd)`, qui pourrait tenter de verrouiller `cp->cwd`. Pourtant, reculer l'appel à `iunlock(ip)` après l'appel à `iput` ne risquerait pas de conduire à un interblocage. Pourquoi ?
7. Implémentez l'appel système `lseek`. Supporter `lseek` nécessitera que vous modifiez `filewrite` pour remplir les trous dans un fichier si `lseek` positionne `off` au-delà de `f->ip->size`.
8. Ajoutez les flags `O_TRUNC` et `O_APPEND` à `open`, de façon que les opérateurs `>` et » fonctionnent en Shell.
9. Modifiez le système de fichiers pour supporter les liens symboliques.

Chapitre 7

Résumé

Ce texte a introduit les principales idées des systèmes d'exploitation par l'étude d'un système, xv6, ligne par ligne. Certaines lignes concrétisent l'essence même de ces principales idées (par exemple la commutation de contexte, la frontière entre les modes utilisateur et noyau, les verrous, etc.) et chaque ligne est importante ; les autres lignes de code illustrent comment implémenter une idée particulière dans un système d'exploitation et pourraient facilement être réalisées différemment (par exemple un meilleur algorithme d'ordonnancement, de meilleures structures de données sur le disque pour représenter les fichiers, une meilleure journalisation pour permettre les transactions concurrentes, etc.). Toutes les idées ont été illustrées dans le contexte d'une interface d'appels système particulière et très réussie, l'interface Unix, mais elles sont également reprises dans la conception d'autres systèmes d'exploitation.

Annexe A

Le matériel du PC

Cette annexe décrit le matériel des ordinateurs personnels (PC), plateforme sur laquelle tourne xv6.

Un PC est un ordinateur conforme à plusieurs normes industrielles, avec pour objectif qu'un logiciel donné puisse s'exécuter sur des machines vendues par différents fournisseurs. Ces normes évoluent avec le temps et un PC des années 90 ne ressemble pas aux PC modernes. La plupart des normes actuelles sont publiques et leur documentation est disponible en ligne.

Vu de l'extérieur, un PC est une boîte avec un clavier, un écran et différents périphériques (par exemple les CD-ROM, etc.). À l'intérieur de cette boîte se trouve un circuit imprimé (la « carte mère ») avec des puces de CPU, de mémoire, graphiques, de controlleur d'entrées/sorties et des bus grâce auxquels les puces communiquent. Les bus respectent des protocols standards (comme par exemple PCI et USB) de sorte que les périphériques fonctionnent avec des PC de différents vendeurs.

De notre point de vue, nous pouvons abstraire les PC en trois composants : le processeur, la mémoire et les périphériques d'entrées/sorties. Le processeur effectue des calculs, la mémoire contient des instructions et des données pour ces calculs et les périphériques permettent au processeur d'intéragir avec le matériel pour le stockage, la communication et d'autres fonctions.

On peut représenter la mémoire principale comme étant connectée au processeur par un ensemble de fils ou de liens, certains pour les bits d'adresse, certains pour les bits de données et d'autres pour les indicateurs [1] de contrôle. Pour lire une valeur depuis la mémoire principale, le processeur envoie des tensions hautes ou basses représentant des bits à 1 ou à 0 sur le lien des adresses et un 1 sur le lien « lecture » pendant une durée déterminée, puis lit les valeurs renvoyées en interprétant les tensions sur le lien des données. Pour écrire une valeur dans la mémoire principale, le processeur envoie les bits appropriés sur les liens adresse et données et un 1 sur le lien « écriture » durant une durée déterminée. Les interfaces mémoires sont en réalité plus complexes, mais les détails de leur fonctionnement ne sont importants que si l'on doit atteindre de hautes performances.

A.1 Processeur et mémoire

Un CPU (pour *Central Processing Unit*) ou processeur d'ordinateur exécute une boucle conceptuellement simple : il consulte une adresse dans un registre appelé « compteur ordinal », lit une instruction machine depuis cette adresse en mémoire, avance le compteur ordinal après l'ins-

1. NdT : Originallement *flags* dans le texte, ce mot sera remplaçé par indicateur ou bits.

truction et exécute l'instruction. Répéter. Si l'exécution de l'instruction ne modifie pas le compteur ordinal, cette boucle interprétera la mémoire pointée par le compteur ordinal comme une séquence d'instructions à exécuter les unes à la suite des autres. Les instructions qui modifient le compteur ordinal incluent les conditionnelles et les apels de fonctions.

Le mécanisme d'exécution est inutile sans la capacité de stocker et de modifier les données du programme. Le stockage le plus rapide pour les données est fourni par l'ensemble les registres processeurs. Un registre est une cellule de stickage à l'intérieur du processeur lui-même, capable de contenir une valeur de la taille d'un mot machine (généralement 16, 32 ou 64 bits). Les données conservées dans ces registres peuvent être lues ou écrites très rapidement, en un seul cycle processeur.

Les PC ont un processeur qui implémente le jeu d'instructions x86, qui a été défini à l'origine par Intel et est devenu un standard. Plusieurs fabriquants produisent des processeurs qui implémentent ce jeu d'instruction. Comme tout les autres standards des PC il évolue, mais les nouveaux standards sont toujours rétrocompatibles. Le chargeur d'amorçage [2] doit prendre en compte cette évolution puisque chaque processeur de PC commence par simuler un Intel 8088, la puce de processeur du IBM PC produit en 1981. Toutefois, pour comprendre le fonctionnement de xv6, il vous faudra être familier avec le jeu d'instructions x86 moderne.

Les x86 modernes fournissent huit registres 32 bits à usage général — `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp` et `%esp` — et un compteur ordinal `%eip` (le « pointeur d'instructions »). Le préfixe commun `e` correspond à étendu, car il s'agit d'extensions 32 bits des registres 16 bits `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, `%sp` et `%ip`. Les deux ensembles de registres sont en correspondance de sorte que par exemple, `%ax` soit la moitié inférieure de `%eax` : écrire dans `%ax` modifie la valeur stockée dans `%eax` et vice versa. Les quatre premiers registres ont aussi un nom pour leur 8 bits de poids faible : `%al` et `%ah` désignent les 8 bits de poids respectivement forts et faible de `%ax`; `%bl`, `%bh`, `%cl`, `%ch`, `%dl` et `%dh` suivent ce modèle. En plus de ces registres, x86 possède huit registres à virgule flottante de 80 bits ainsi qu'une poignée de registres spéciaux comme les « registres de contrôle » `%cr0`, `%cr2` [3] , `%cr3` et `%cr4`; les registres de débogage `%dr0`, `%dr1`, `%dr2`, et `%dr3`; les « registres de segments » `%cs`, `%ds`, `%es`, `%fs`, `%gs` et `%ss`; et les pseudo-registres de la table des descripteurs locale et globale `%ldtr` et `%gdtr`. Les registres de contôle et les registres de segments sont importants pour tout système d'exploitation. Les registres à virgule flottante et les registres de débogage sont moins intéressants et ne sont pas utilisés par xv6.

Les registres sont rapides mais coûtent cher. La plupart des processeurs fournissent au plus quelques dizaine de registres à usage général. Le prochain niveau conceptuel de stockage est la mémoire vive principale (ou RAM pour *Random-Access Memory*). La mémoire principale est 10 à 100 fois plus lente qu'un registre, mais elle est beaucoup moins coûteuse, on peut donc en avoir plus. Une des raisons qui rend la mémoire principale relativement lente est qu'elle est physiquement séparé de la puce de processeur. Un processeur x86 possède une petite douzaine de registres, mais un PC actuel comprend généralement des gigabytes de mémoire principale. En raison de cette énorme différence à la fois en temps d'accès et en taille entre les registres et la mémoire principale, la plupart des processeurs, dont le x86, conservent des copies de sections de la mémoire principale récemment accédées dans une mémoire cache. La mémoire cache sert d'intermédiaire entre les registres et la mémoire à la fois en temps d'accès et en taille. Aujourd'hui, les processeurs x86 possèdent générallement trois niveaux de caches. Chaque cœur a un petit cache de premier niveau avec des temps d'accès relativement proches de la fréquence d'horloge du processeur ainsi qu'un cache de second niveau plus grand. Plusieurs cœurs se partagent un même cache de niveau 3.

2. NdT : Le chargeur d'amorçage ou *boot loader* est un logiciel permettant le démarrage d'un ou de plusieurs systèmes d'exploitation.

3. NdT : Le registre `%cr1` est réservé au processeur.

Mémoire	Temps d'accès	Taille
registres	1 cycle	64 octets
Cache L1	~4 cycles	64 kilooctets
ache L2	~10 cycles	4 megaoctets
Cache L3	~40-75 cycles	8 megaoctets
Cache L3 distant	~100-300 cycles	
DRAM locale	~60 nsec	
DRAM distante	~100 nsec	

FIGURE A.1 – Temps de latence pour un système Intel Xeon, basés sur http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

La majorité des processeurs x86 masquent les caches aux systèmes d'exploitation, on peut donc se représenter un processeur comme ayant simplement deux sortes de stockage — les registres et la mémoire — sans se soucier des distinctions entre les différents niveaux de la hiérarchie de la mémoire.

A.2 Entrées/Sorties

Les processeurs doivent communiquer aussi bien avec les périphériques qu'avec la mémoire. Le processeur de x86 fournit les instruction spéciales `in` et `out` qui lisent et écrivent des valeurs depuis les adresses du périphérique appelées « ports d'entrées/sorties ». L'implémentation matérielle de ces instructions est essentiellement la même que lire ou écrire dans la mémoire. Les premiers processeurs x86 possédaient un lien d'adresse supplémentaire : 0 signifiait lire/écrire depuis un port d'entrées/sorties et 1 signifiait lire/écrire depuis la mémoire principale. Chaque périphérique matériel surveille ces liens pour les lectures et les écritures dans la plage de ports d'entrées/sorties qui lui est attribuée. Les ports d'un périphérique permettent au logiciel de configurer ce périphérique, d'examiner son statut et de lui demander d'effectuer des actions ; par exemple, les logiciels peuvent utiliser les lectures et écritures des ports d'entrées/sorties pour amener l'interface matériel du disque à lire et écrire des secteurs sur le disque.

De nombreuses architectures d'ordinateurs ne possèdent pas d'instructions d'accès aux périphériques spécifiques. À la place, les périphériques possèdent des adresses mémoires fixes et le processeur communique avec un périphérique (sous ordre du système d'exploitation) en lisant et écrivant des valeurs à ces adresses. Dans les faits, les architectures x86 modernes utilisent cette technique appelée « *adressage des entrées/sorties en mémoire* » pour la majorité des périphériques à haut débit tels que les contrôleurs réseaux, disques et graphiques. Pour des raisons de rétrocompatibilité, les anciennes instructions `in` et `out` persistent, tout comme les périphériques qui les utilisent, comme les controlleur de disque IDE, qu'utilise xv6.

Annexe B

Le chargeur d'amorçage

Lorsqu'un PC x86 démarre, il commence par exécuter un programme appelé le BIOS (*Basic Input/Output System*), qui est stocké dans une mémoire non volatile sur la carte mère. Le rôle du BIOS est de préparer le matériel après quoi le chargeur d'amorçage va transférer le contrôle au système d'exploitation. Plus précisemment, il transfère le contrôle au code chargé depuis le secteur de démarrage, le premier secteur de 512 octets du disque de démarrage. Le secteur de démarrage contient le chargeur d'amorçage : une suite d'instructions qui charge le noyau en mémoire. Le BIOS charge le secteur de démarrage depuis l'adresse `0x7c00` puis saute (place le registre processeur `%ip`) à cette adresse. Lorsque le chargeur d'amorçage commence son exécution, le processeur simule un processeur Intel 8088 et le rôle du chargeur d'amorçage est de mettre le processeur dans un mode de fonctionnement plus moderne pour charger le noyau xv6 du disque vers la mémoire puis transférer le contrôle au noyau. Le chargeur d'amorçage de xv6 comprend deux fichiers sources, l'un écrit dans une combinaison d'assembleur 16 et 32 bits (`bootasm.S`; [9100]) et l'autre écrit en C (`bootmain.c`; [9200])).

B.1 Code : Amorçage (en assembleur)

La première instruction du chargeur d'amorçage est `cli` [9112], fonction qui désactive les interruptions processeur. Les interruptions sont un moyen pour le matériel de solliciter une fonction du système d'exploitation nommée gestionnaire d'interruptions. Le BIOS est un petit système d'exploitation, et il peut avoir instauré son propre gestionnaire d'interruptions lorsqu'il initialisait le matériel. Mais le BIOS n'est plus en cours d'exécution — c'est le chargeur d'amorçage qui s'exécute — et il n'est donc plus approprié ou sûre de gérer les interruptions des périphériques matériels. Lorsque xv6 sera prêt (dans le chapitre 3), il va ré-activer les interruptions.

Le processeur est en « mode réel », mode dans lequel il simule un Intel 8088. En mode réel, il y a huit registres 16 bits à usage général, or le processeur envoie des adresses 20 bits à la mémoire. Les registres de segment `%cs`, `%ds`, `%es` et `%ss` fournissent les bits supplémentaires nécessaires pour générer des adresses mémoires 20 bits à partir de registres 16 bits. Lorsqu'un programme fait référence à une adresse mémoire, le processeur ajoute automatiquement 16 fois la valeur de l'un des registres de segment ; ces registres ont une taille de 16 bits. Le registre de segment à utiliser est génerallement implicite selon le type de mémoire référencé : la recherche d'instructions utilise `%cs`, la lecture et l'ecriture de données utilisent `%ds` et la lecture et l'ecriture de la pile utilisent `%ss`.

Xv6 suppose qu'une instruction x86 utilise une adresse virtuelle pour ses opérandes mémoires, or une instruction x86 utilise actuellement une « adresse logique » (voir figure B.1). Une adresse logique est constituée d'un selecteur de segment et d'un offset, et est parfois notée `segment:offset`.

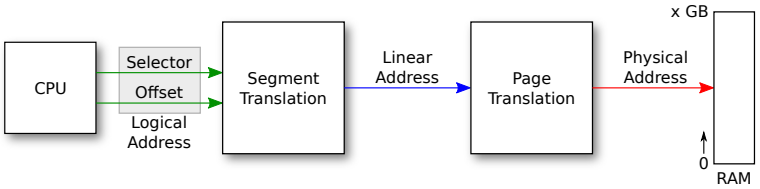

FIGURE B.1 – Relations entre adresses logiques, linéaires et physiques.

La plupart du temps, le segment est implicite et le programme ne manipule directement que l'offset. Le processeur [1] effectue les traductions décrite ci-dessus pour générer une « adresse linéaire ». Si la MMU [2] est activée (voir chapitre 2), elle traduit les adresses linéaires en adresses physiques ; dans le cas contraire, le processeur utilise les adresses linéaire comme des adresses physiques.

Le chargeur d'amorçage n'active pas la pagination ; les adresses logiques utilisées sont traduites en adresses linéaires par le processeur, puis utilisées directement comme adresses physiques. Xv6 configure le processeur pour traduire les adresses logiques en adresses linéaires sans changements de sorte qu'elles soient toujours égales. Pour des raisons hitoriques, nous avons utilisé le terme « adresse virtuelle » pour faire référence aux adresses manipulées par les programmes ; une adresse virtuelle xv6 est identique à une adresse logique x86 et est égale à l'adresse linéaire qui lui est associée par le processeur. Lorsque la pagination sera activée, la seule traduction d'adresse intéréssante dans le système sera entre linéaire et physique.

Le BIOS ne fournit aucune garantie quant au contenu de `%ds`, `%es` et `%ss`, ainsi la priorité après avoir désactivé les interruptions est de mettre la valeur du registre `%ax` à zéro puis de copier ce zéro dans les registres `%ds`, `%es` et `%ss` [9115-9118].

Un `segment:offset` virtuel peut produire une adresse physique de 21 bits, or le Intel 8088ne peut adresser que des adresses mémoire de 20 bits, ainsi il écarte le bit de poids fort : `0xffff0+0xffff` = `0x10ffef`, mais l'adresse virtuelle `0xffff:0xffff` sur le 8088 fait référence à l'adresse physique `0x0ffef`. Certains des premiers logiciels reposant sur le matériel ignoraient le 21ème bit d'adresse, ainsi lorsqu'Intel a introduit des processeur avec plus de 20 bits d'adresse physique, IBM a fourni un hack de compatibilité qui est indispensable pour tout matériel compatible PC. Si le second bit du port de sortie contrôlleur de clavier vaut zéro, le 21ème bit de l'adresse physique est toujours effacé ; s'il vaut 1, le 21ème bit se comporte normallement. Le chargeur d'amorçage doit activer le 21ème bir d'adresse en utilisant les entrées/sorties au contrôlleur de clavier sur les ports 0x64 et 0x60 [9120-9136].

Les registres 16 bits d'usages généraux et de segment du mode réel rend l'utilisation par un programme de plus de 65 536 octets de mémoire difficile et il est impossible d'utiliser plus d'un megaoctet. Les processeurs x86 possèdent, depuis le 80286, un « mode protégé », qui permet aux adresses physiques d'avoir beaucoup plus de bits et, depuis le 80386, un « mode 32 bits » qui permet aux registres, aux adresses virtuelles et à la plupart des calculs arithmétiques d'être effectués avec 32 bits plutôt que 16. La séquence de démarrage de xv6 active le mode protégé et le mode 32 bits de la façon suivante.

En mode protégé, un registre de segment est un indexe vers une « table des descripteurs de segments » (cf. figure B.2). Chaque entrée de la table spécifie une adresse physique de base,

1. NdT : Dans le texte original, le terme *segmentation hardware* est employé, mais par soucis de simplicité, nous utiliserons « processeur ».
2. NdT : Dans le texte original, le terme *paging hardware* est employé. La pagination étant décrite dans le chapitre 2, nous utiliserons directement le terme de MMU (*Memory Managment Unit*), que nous accorderons au féminin, la MMU représentant une unité.

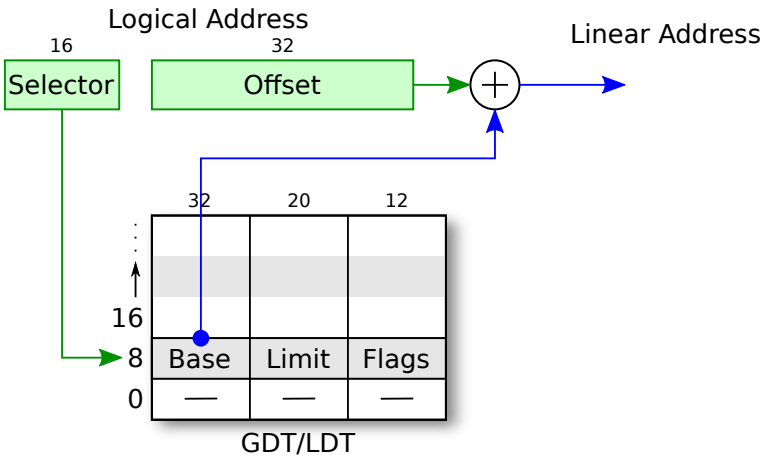

FIGURE B.2 – Segments en mode protégé.

une adresse virtuelle maximale appelée la limite et les bits d'authorisation pour le segment. Ces permissions représentent les protections en mode protégé : le noyau peut les utiliser pour s'assurer qu'un programme n'utilise que son propre espace mémoire.

Xv6 n'utilise pratiquement pas les segments ; il utilise à la place la pagination, comme le décrit le chapitre 2. Le chargeur d'amorçage instaure la table des descripteurs de segments `gdt` 9182-9185 de sorte que tous les segments aient une adresse de base de zéro et la limite maximale possible (quatre gigaoctets). La table possède une entrée nulle, une entrée pour le code exécutable et une entrée pour les données. Le descripteur de segment du code a un indicateur permettant de savoir si le code doit s'exécuter en mode 32 bits 0660. Avec cette configuration, lorsque le chargeur d'amorçage entre dans le mode protégé, les adresses logiques correspondent unes à unes aux adresse physiques.

Le chargeur d'amorçage exécute une instruction `lgdt` [9141] pour charger le registre GDT (*Global Descriptor Table*) avec la valeur `gdtdesc` [9187-9189], qui pointe sur la table `gdt`.

Une fois le registre GDT chargé, le chargeur d'amorçage active le mode protégé en mettant la valeur 1 bit (`CR0_PE`) dans le registre `%cr0` [9142-9144]. Activer le mode protégé ne change pas immédiatement la façon dont le processeur traduit les adresses logiques en adresses physiques ; ce n'est que lorsque l'on charge une nouvelle valeur dans un registre de segment que le processeur lit le registre GDT et modifie ses paramètres de segmentation internes. On ne peut pas modifier directement `%cs`, ainsi à la place, on exécute une instruction `ljump` (*far jump*) [9153], qui permet de spécifier un selecteur de segment code. Le jump continue son exécution à la ligne suivante [9156], et ce faisant, définit `%cs` comme faisant référence à l'entrée descripteur de code dans `gdt`. Ce descripteur décrit un segment code 32 bits, ainsi le processeur commute en mode 32 bits. Le chargeur d'amorçage a permit au processeur d'évoluer depuis le 8088 vers le 80286 puis le 80386.

La première action du chargeur d'amorçage en mode 32 bits est d'initialiser les registres de segments données avec la valeur `SEG_KDATA` [9158-9161]. Les adresses logiques correspondent maintenant directement aux adresses physiques. La seule étape à effectuer avant d'exécuter du code C est d'instaurer une pile dans une région inutilisée de la mémoire. La mémoire entre de l'adresse `0xa0000` jusqu'à `0x100000` est généralement parsemée de zones de mémoire de périphériques, et le noyau de xv6 s'attend à être situé à l'adresse `0x100000`. Le chargeur d'amorçage lui est situé entre les adresses `0x7c00` et `0x7e00` (512 octets). N'importe quelle autre section de la mémoire seraient fondamentalement un bon emplacement pour la pile. Le chargeur d'amorçage choisit l'adresse `0x7c00` (connu dans ce fichier comme `$start`) comme

sommet de la pile; la pile décroit à partir d'ici jusqu'à atteindre l'adresse `0x0000`, loin du chargeur d'amorçage.

Enfin, le chargeur d'amorçage appelle la fonction C `bootmain` [9168]. Le rôle de bootmain est de charger et d'exécuter le noyau. Cette fonction ne retourne que si quelque chose s'est mal passé. Dans ce cas, le code envoie quelques mots en sortie sur le port `0x8a00` [9170-9176]. Sur du vrai matériel, il n'y a pas de périphériques connecté à ce port, ainsi ce code ne fait rien. Si le chargeur d'amorçage s'exécute à l'intérieur d'un simulateur de PC, le port `0x8a00` est connecté au simulateur lui-même et peut lui rendre le contrôle. Simulateur ou non, le code exécute ensuite une boucle infinie [9177-9178]. Un vrai chargeur d'amorçage pourrait tenter d'afficher un message d'erreur avant.

B.2 Code : Amorçage (en C)

La partie en C du chargeur d'amorçage, `bootmain.c` [9200], s'attend à trouver une copie de l'exécutable du noyau sur le disque, à partir du deuxième secteur. Le noyau est un binaire au format ELF, comme vu dans le chapitre 2. Pour avoir accès à l'en-tête ELF, `bootmain` charge les premiers 4096 octets du binaire ELF [9227]. Il place la copie en mémoire à l'adresse `0x10000`.

> NdT : Ce paragraphe a été supprimé de la version originale du livret, je le replace ici à titre informatif.
>
> `Bootmain` effectue beaucoup de casts entre pointeurs et entiers, et entre différents types de pointeurs [9224, 9227, and so on]. Ces casts n'ont de sens que si le compilateur et le processeur représentent les entiers et tous les pointeurs essentiellement de la même manière, ce qui n'est pas vrai sur toutes les combinaisons de machines/compilateurs. Cela est en revanche vrai pour le x86 en mode 32 bits : les entiers sont encodés en 32 bits et tous les pointeurs sont des adresses également sur 32 bits.

La prochaine étape est une vérification rapide qu'il s'agisse probablement bien d'un binaire ELF, et non d'un disque non initialisé. `Bootmain` lit le contenu de la section commençant à l'emplacement disque `off` octets après le début de l'en-tête ELF, et écrit à l'adresse `paddr`. `Bootmain` appelle `readseg` pour charger depuis le disque [9238] et appelle `stosb` pour mettre à zéro le reste du segment [9240]. `Stosb` [0492] utilise l'instruction x86 `rep stosb` pour initialiser chaque octet d'un bloc de la mémoire.

> NdT : Cette partie a été supprimée de la version originale du livret, je la replace ici à titre informatif.
>
> `Readseg` [9279] lit au minimum `count` octets depuis la position `offset` du disque, vers la mémoire à l'adresse `pa`. L'interface de disque IDE du x86 fonctionne sur des secteurs de 512 octets, ainsi `readseg` peut lire non seulement la section mémoire désirée, mais aussi quelques octets avant et après, dépendant de l'alignement. Pour le segment de programme de l'example ci-dessus, le chargeur d'amorçage va appeler `readseg((uchar*)0x100000, 0xb57e, 0x1000)`. `Readseg` commence par évaluer l'adresse physique de fin, la première adresse supérieur à `paddr` qui ne nécesssite pas d'être chargée depuis le disque [9283], et arrondit à l'inférieur `pa` à un décalage aligné sur les secteur du disque. Puis il convertit le décalage d'un décalage exprimé en octets à un décalage exprimé en secteurs; il ajoute 1 car le noyau commence au secteur disque 1 (le secteur disque 0 correspond au chargeur d'amorçage). Enfin, il appelle `readsect` pour lire chaque secteur vers la mémoire.
>
> `Readsect` [9260] lit un unique secteur disque. `Readsect` commence par appeler `waitdisk` pour attendre jusqu'à ce que le disque signale qu'il est prêt à accepter des commandes. Le disque se signale en mettant les deux bits de poids forts de son octet d'état (connecté au port d'entré `0x1F7`) à `01`. `Waitdisk` [9251] lit l'octet d'état jusqu'à ce que les bits aient les bonnes valeurs. Le chapitre 3 utilise une manière efficace d'attendre les changements de status du matériel, mais le *polling*, tel qu'effetué ici, est convenable pour le chargeur d'amorçage.

Une fois le disque prêt, `readsect` effectue une commande de lecture. Il commence par écrire les arguments de la commande — le nombre de secteurs et le numéro de secteur (offset) — sur les registres du disque sur les ports de sortie `0x1F2-0x1F6` [9264-9268]. Le bit `0xe0` écrit sur le port `0x1f6` signale au disque que `0x1f3-0x1f6` contient un numéro de secteur (aussi appelé adresse de bloc linéaire) en opposition avec une adresse de cylindre/-tête/secteur plus compliquée, utilisée dans les disques PC plus anciens. Après avoir écrit les arguments, `readsect` écrit au registre de commande pour déclencher la lecture [9254]. La commande `0x20` correspond à « lecture de secteurs ». À partir d'ici, le disque va lire les données stoquées dans les secteurs spécifiés et les rendre accessibles par morceaux de 32 bits sur le port d'entré `0x1f0`. `Waitdisk` [9251] attend jusqu'à ce que le disque signale que les données sont prêtes, puis l'appel à `insl` lit les 128 (`SECTSIZE/4`) morceaux 32 bits vers la mémoire commençant à l'adresse `dst` [9273].

`Inb`, `outb` et `insl` ne sont pas des fonctions C ordinaires. Il s'agit de fonctions inline dont le corps comprend des morceaux en langage assembleur [0453, 0471, 0462]. Lorsque `gcc` (le compilateur C qu'utilise xv6) voit les appels à `inb` [9254], l'assembleur intégré provoque l'émission d'une seule instruction `inb`. Ce style permet d'utiliser des instruction bas niveaux comme `inb` et `outb` alors que l'on écrit toujours la logique en C plutôt qu'en assembleur.

L'implémentation de `insl` [0462] mérite de s'attarder dessus. `Rep insl` est en réalité une petite boucle déguisée en une simple instruction. Le préfixe `rep` exécute les instructions suivantes `%ecx` fois, décrémentant `%ecx` après chaque itérations. L'instruction `insl` lit des valeurs 32 bits depuis le port `%dx` vers la mémoire à l'adresse `%edi` puis incrémente `%edi` de 4. Ainsi, `rep insl` copie 4×`%ecx` octets, dans des morceaux 32 bits, depuis le port `%dx` vers la mémoire commençant à l'adresse `%edi`. Les annotations du registre indiquent à `gcc` de se préparer à la séquence assembleur en stockant `dst` dans le registre `%edi`, `cnt` dans `%ecx` et `port` dans `%dx`. Ainsi, la fonction `insl` copie les 4×`cnt` octets depuis le port 32 bits `port` vers la mémoire commençant à `dst`. L'instruction `cld` nettoie l'indicateur de direction du processeur, de sorte que l'instruction `insl` incrémente `%edi` ; lorsque l'indicateur est activé, `insl` décrémente `%edi` à la place. Les conventions d'appels x86 ne définissent pas l'état de l'indicateur de direction à l'entrée d'une fonction, ainsi chaque utilisation d'une instruction `insl` doit l'initialiser à la valeur souhaitée.

Le chargeur d'amorçage est presque terminé. `Bootmain` boucle en appelant `readseg` qui boucle en appelant `readsect` [9236-9241]. À la fin de la boucle, `bootmain` a chargé le noyau dans la mémoire.

La compilation et l'édition de lien du noyau sont tels que le noyau s'attend à se trouver à l'adresse virtuelle commençant à `0x80100000`. Ainsi, les instructions d'appel de fonctions doivent mentionner des adresses de destination du type `0x801xxxxx` ; des exemples se trouvent dans `kernel.asm`. Cette adresse est configurée dans `kernel.ld` [9311]. `0x80100000` est une adresse relativement haute, située vers la fin de l'espace d'adressage de 32 bits ; le chapitre 2 explique les raisons de ce choix. Il pourrait ne pas y avoir de mémoire physique à une si haute adresse. Une fois que le noyau commence son exécution, il va initialiser la MMU pour quelle lie les adresses virtuelles comma,çant à `0x80100000` aux adresses physiques commançant à `0x00100000` ; le noyau suppose qu'il existe une adresse physique à une si petite adresse. À ce moment du processus d'amorçage cependant, la MMU n'est pas activée. À la place, `kernel.ld` spécifie que `paddr` de l'ELF commence à l'adresse `0x00100000`, ce qui fait que le chargeur d'amorçage copie le noyau à une adresse basse, à laquelle la MMU va éventuellement faire référence.

La dernière étape du chargeur d'amorçage est d'appeler le point d'entrée noyau, qui est l'instruction par laquelle le noyau s'attend à commencer son exécution. Pour xv6, l'adresse du point d'entrée est `0x10000c` :

```
# objdump -f kernel

kernel:     file format elf32-i386
```

```
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

Par convention, le symbole `_start` spécifie le point d'entré ELF, qui est définit dans `entry.S` [1040]. Puisque xv6 n'a pas initialiser de mémoire virtuelle pour l'instant, le point d'entrée de xv6 est l'adresse physique de `entry` [1044].

B.3 Dans la réalité

Le chargeur d'amorçage décrit dans cette annexe compile autour de 470 octets de code machine, dépendant des optimisations utilisées durant la compilation du code C. Afin de tenir dans cette petite quantité de mémoire, le chargeur d'amorçage de xv6 effectue l'hyptohèse de simplification majeure que le noyau a été écrit sur le disque de démarrage, de manière contigüe, et qu'il commence au secteur 1. De façon plus commune, les noyaux sont stoqués dans des systèmes de fichiers ordinaires où ils peuvent ne pas être contigüe, ou sont chargés depuis le réseau. Ces complications nécessitent que le chargeur d'amorçage soit capable de gérer toute une palette de contrôleur de disque et réseau et de comprendre plusieurs systèmes de fichiers et protocols réseaux. En d'autres termes, le chargeur d'amorçage lui-même doit être un système d'exploitation miniature. Puisqu'un chargeur d'amorçage aussi complexe ne tiendrait sûrement pas en 512 octets, la plupart des systèmes d'exploitation des PC utilisent un processus de démarrage en deux temps. Premièrement, un chargeur d'amorçage simple, comme celui montré dans cette annexe, charge un chargeur d'amorçage plus complet depuis un emplacement connu du disque, s'appuyant souvent sur le BIOS pour accéder au disque plutôt que d'essayer d'y accéder[3] lui même. Puis, le chargeur d'amorçage complet, libéré de la limite des 512 octets, peut implémenter la compléxité nécessaire pour trouver, charger et exécuter le noyau désiré. Les PC modernes évitent beaucoup des difficultés citées car ils supportent le format UEFI (*Unified Extensible Firmware Interface*), qui permet aux PC de lire un chargeur d'amorçage plus lourd depuis le disque (et de l'exécuter en mode protégé ou en mode 32 bits).

Cette annexe est écrite comme si la seule chose qui se produisait entre l'allumage et l'exécution du chargeur d'amorçage était que le BIOS chargait le secteur de démarrage. En réalité, le BIOS effectue un nombre considérable d'initialisations afin que le matériel complexe d'un ordinateur moderne ressemble à un PC standard traditionnel. Le BIOS est vraiment un système d'exploitation miniature embarqué dans le matériel, qui s'exécute après le démarrage de l'ordinateur.

B.4 Exercices

1. En raison de la granularité, l'appel à `readseg` dans le code est équivalent à `readseg((uchar*)0x100 0xb500, 0x1000)`. En pratique, ce mauvais comportement ne pose pas de problème. Pourquoi le mauvais `readsect` ne pose-t-il pas de problème ?
2. Supposons que nous voulions que `bootmain` charge le noyau à l'adresse `0x200000` au lieu de `0x100000`, et que nous ayons réalisé ceci en modifiant `bootmain` pour ajouter `0x100000` aux adresses virtuelles de chaque section ELF. Quelque chose ne fonctionnerait pas. Quoi ?
3. Il paraît potentiellement dangereux pour le chargeur d'amorçage de copier l'en-tête ELF en mémoire, à l'emplacement arbitraire `0x100000`. Pourquoi ne pas appeler `malloc` pour obtenir l'espace mémoire necéssaire ?

3. NdT : Et pour ce faire, devoir savoir utiliser des pilotes.