

Exercices supplémentaires autour de xv6

2022 – 2023

©Pierre David

Disponible sur <https://gitlab.com/pdagog/ens>.

Ce texte est placé sous licence « Creative Commons Attribution – Pas d'Utilisation Commerciale 4.0 International »

Pour accéder à une copie de cette licence, merci de vous rendre à l'adresse suivante
<https://creativecommons.org/licenses/by-nc/4.0/>



Table des matières

| | |
|--|----|
| Chapitre 1 - Organisation du système | 5 |
| Chapitre 2 - Tables de pages | 9 |
| Chapitre 3 - Trappes, interruptions et périphériques | 11 |
| Chapitre 4 - Verrouillage | 13 |
| Chapitre 5 - Ordonnancement | 15 |
| Chapitre 6 - Système de fichiers | 21 |

Chapitre 1 - Organisation du système

Exercice 1.1

Cet exercice est un préalable à tous les exercices où vous aurez à modifier du code de xv6. L'utilisation de git et d'un dépôt privé vous permet de facilement repérer vos modifications et d'y accéder depuis n'importe quel ordinateur.

1. Commencez par *cloner* le dépôt original du MIT, <https://github.com/mit-pdos/xv6-public>
Attention : ne vous contentez pas de télécharger les sources, mais utilisez le bouton `Code`, recopiez l'URL de clonage et utilisez-la avec `git clone`.
Astuce : vous pouvez utiliser l'utilitaire graphique gitk (paquet Ubuntu gitk) pour naviguer dans l'historique du dépôt. Si vous n'avez pas cloné correctement le dépôt, gitk vous l'indiquera.
2. Si vous ne l'avez pas déjà fait, il est temps d'installer votre clef SSH sur votre compte <https://gitlab.unistra.fr> sinon vous devrez saisir votre mot de passe à chaque fois que vous accédez à votre dépôt.
3. Créez ensuite un dépôt privé (vide) sur <https://gitlab.unistra.fr> (nouveau « projet » dans la terminologie gitlab)
4. Vous pouvez ensuite suivre les instructions affichées pour configurer votre identité et pousser le dépôt git existant (que vous venez de récupérer à l'étape 1) sur gitlab.unistra.fr
Vérifiez avec `git remote -v` que l'origine est bien votre dépôt sur gitlab.unistra.fr
5. Pour conserver un point de référence stable, il vous est suggéré de faire vos modifications uniquement dans des branches dédiées. Pour créer et utiliser une branche, il faut utiliser les commandes suivantes :

```
git branch toto    pour créer une branche toto locale
git checkout toto  pour basculer sur la branche toto locale
git push --all     pour propager la branche locale vers gitlab.unistra.fr
```

Modifiez le fichier README, puis commitez votre modification et poussez-la vers le serveur.

Astuce 1 : vérifiez sur le serveur que votre branche a bien été créée et contient votre commit.

Astuce 2 : en utilisant `gitk --all`, vous pouvez voir graphiquement toutes vos branches.

6. Utilisez la commande `git diff master` pour afficher les différences par rapport à la branche master
7. Pour revenir à la branche principale, utilisez `git checkout master`
8. Si vous n'êtes pas à l'aise avec git et les branches, précipitez-vous sur le site de git (<https://git-scm.com>) et le livre en ligne <https://git-scm.com/book/en/v2>

Pour toutes vos modifications, prenez bien soin de repartir de la branche master, c'est-à-dire de la version originale de xv6, et de créer une nouvelle branche pour isoler vos modifications.

Exercice 1.2

Démarrez xv6 sans débogueur : tapez `make qemu`. Une fenêtre qemu s'ouvre, qui émule l'écran et le clavier d'un PC standard. Xv6 démarre un shell simplifié. Dans cette fenêtre, vous pouvez utiliser la commande `ls` de xv6, simplifiée elle aussi : les 3 nombres associés à chaque fichier sont :

- le type du fichier (1 pour répertoire, 2 pour fichier régulier, 3 pour périphérique)
- le numéro d'inode du fichier
- la taille du fichier en octets

Toujours dans la fenêtre qemu, utilisez la commande `cat README`. Vous constatez que xv6 suppose que le clavier matériel émulé par qemu est un clavier QWERTY. Vous verrez ultérieurement que la lecture des touches du clavier US est déjà bien complexe...

En revanche, la fenêtre terminal dans laquelle vous avez lancé `make qemu` affiche les mêmes informations que l'écran émulé par qemu. Vous pouvez également taper les commandes dans cette fenêtre, que qemu considère comme un terminal connecté par une liaison série, qui envoie donc les codes ASCII correspondant aux caractères saisis. Ceci permet d'utiliser le clavier normal que votre système natif reconnaît correctement. Xv6

diffère d'un système standard en ceci qu'il attend des informations indifféremment du matériel « clavier » ou du matériel « contrôleur de périphérique de liaison série ».

Vous pouvez utiliser `make qemu-nox` pour démarrer xv6 sans fenêtre parasite d'émulation du clavier et de l'écran.

Si vous avez un ordinateur portable, vous constaterez vraisemblablement pendant cet exercice qu'il chauffe et que le ventilateur se met en route. Vous pouvez utiliser `top` pour constater que `qemu` consomme énormément de temps CPU.

Laissez votre processeur refroidir un peu en fermant la fenêtre `qemu` ou en utilisant `kill` sur le processus `qemu`.

Exercice 1.3

Avant d'utiliser un débogueur pour interagir avec le noyau, il faut ajouter la ligne suivante (cf cours) dans le fichier `$HOME/.gdbinit` :

```
add-auto-load-safe-path /le/chemin/où/vous/avez/placé/xv6-public/.gdbinit
```

Pour faciliter l'utilisation du débogueur, il est préférable de modifier le fichier `Makefile` pour :

- changer le niveau d'optimisation `-O2` en `-Og` lors de la compilation (n'oubliez pas de forcer la recompilation de tous les fichiers en faisant `make clean`);
- émuler un seul processeur, en mettant la variable `CPUS` à 1.

Vous pouvez ensuite lancer `make qemu-gdb` ou `make qemu-nox-gdb`. Dans une autre fenêtre terminal, depuis le même répertoire, démarrez `gdb` (sans argument), puis tapez la commande `c` (ou `continue`) pour que `gdb` donne à `qemu` l'ordre de démarrer. Vous pouvez taper `Ctrl C` dans `gdb` pour suspendre l'exécution à tout moment et interagir avec `gdb` et le noyau `xv6`.

La table ci-après liste quelques commandes de `gdb` (cf <https://www.gnu.org/software/gdb/documentation/>) :

| Commande | Raccourci | Explication |
|---------------------------------|------------------|---|
| <code>window</code> | <code>win</code> | scinde la fenêtre en deux (source en haut / commandes <code>gdb</code> en bas) |
| <code>break mafct</code> | <code>b</code> | pose un point d'arrêt à l'entrée de <code>mafct</code> |
| <code>continue</code> | <code>c</code> | lance l'exécution jusqu'au prochain point d'arrêt |
| <code>advance 123</code> | | continue l'exécution jusqu'à la ligne 123 |
| <code>finish</code> | <code>fin</code> | termine l'exécution de la fonction courante |
| <code>where</code> | <code>bt</code> | affiche la pile |
| <code>print *toto[10]</code> | <code>p</code> | affiche l'élément pointé par le 11e élément du tableau <code>toto</code> |
| <code>print /x *toto[10]</code> | <code>p</code> | idem, mais affichage en hexadécimal |
| <code>step</code> | <code>s</code> | exécute une instruction C |
| <code>next</code> | <code>n</code> | exécute une instruction C sans s'arrêter si appel de fonction |
| <code>stepi</code> | <code>si</code> | exécute une instruction assembleur |
| <code>info registers</code> | <code>i r</code> | affiche le contenu des registres du processeur |
| <code>x/10wd toto</code> | <code>x</code> | affiche 10 mots (<i>word</i> , 32 bits) en décimal à partir de l'adresse <code>toto</code> |
| <code>kill</code> | <code>k</code> | termine <code>xv6</code> |
| <code>quit</code> | <code>q</code> | quitte le débogueur |

Pour vous familiariser avec `gdb`, placez un point d'arrêt dans la fonction `sys_chdir`, puis lancez l'exécution. Tapez la commande « `cd .` ». Lorsque `gdb` s'arrête, avancez juste après l'initialisation de la variable `curproc`, puis affichez le contenu du descripteur de processus : quels sont le numéro et l'état du processus ? Quelle est l'adresse de la pile noyau du processus ? Quel est le numéro du processus père ?

Lors de l'appel d'une primitive système, le noyau sauvegarde les registres du processeur manipulés par le processus dans la structure `trapframe` (voir `x86.h`), elle-même située dans le descripteur de processus (champ `tf`, voir `proc.h`). Quelle est l'adresse (en hexadécimal) de la prochaine instruction que le processus exécutera ? Quelle est l'adresse de la pile utilisateur ?

Pourquoi les adresses des deux piles (utilisateur et noyau) sont-elles si différentes ?

Avancez jusqu'à l'appel de `ilock`. Quelle est la valeur de l'argument de la primitive `chdir`, placé dans la variable `path` par la fonction `argstr` ?

Avant de quitter la session gdb avec la commande `q`, n'oubliez pas d'arrêter le programme en cours avec `k`.

Exercice 1.4

Réalisez votre première modification dans le noyau de xv6 : dans le fichier `syscall.c`, fonction `syscall`, juste après la ligne « `num = curproc->tf->eax` », affichez la valeur de `num` (numéro de la primitive système appelée) avec la fonction `cprintf` qui s'utilise à peu près comme la fonction `printf` de la bibliothèque standard.

Recompilez xv6 et démarrez-le (sans débogueur, c'est plus simple). Contemplez le résultat de votre modification. Vous pouvez faire `git checkout -f syscall.c` pour restaurer le fichier dans son état d'origine (il n'est sans doute pas très utile de sauvegarder cette modification).

Pourquoi faut-il utiliser dans le noyau xv6 une fonction `cprintf` et non la vraie fonction `printf` ?

Exercice 1.5

Au démarrage de l'ordinateur, le BIOS est le programme chargé de lire le noyau depuis le disque dur et de le placer en mémoire à partir de l'adresse `0x10 0000`. Une fois le noyau en mémoire, le BIOS lui transfère l'exécution à partir de l'étiquette `entry` (fichier `entry.S`).

Une des premières missions du code de xv6 à cette adresse est d'activer la traduction mémoire via l'unité de gestion mémoire (MMU). Pour ce faire, xv6 utilise le tableau constant `entrypgdir` (fichier `main.c`) comme table des pages initiale et temporaire, avec la caractéristique particulière qu'il s'agit d'une table simple de 1024 pages de 4 Mo (et non de 4 Ko), telle que décrite dans les figures 4.3 et 4.4 du « *Intel 64 and IA-32 Architectures Software Developer's Manual* » <https://software.intel.com/en-us/articles/intel-sdm>.

En vous aidant du source (fichier `main.c` et fichiers inclus comme `mmu.h`) et des figures ci-dessus mentionnées, donnez le contenu de cette table (indices, contenu précis de chaque entrée). Quelle est la signification du contenu (les bits) de chaque entrée ?

Une fois l'adresse de cette table configurée dans la MMU (instruction « `movl %eax,%cr3` », fichier `entry.S`) comment le noyau perçoit-il la mémoire ? Représentez graphiquement (avec les adresses numériques) les plages d'adresses accessibles par le noyau et les adresses virtuelles et physiques correspondantes.

Exercice 1.6

Xv6 dispose d'un nombre limité de commandes. Le but de l'exercice est d'ajouter la commande `mv`. Vous vous limiterez ici à la forme simple permettant de renommer un fichier :

```
mv nom-actuel nouveau-nom
```

Rédigez cette commande, en prenant soin de vérifier le nombre d'arguments. Terminez l'exécution en appelant la primitive `exit`, sans argument dans xv6, sinon vous aurez un message d'erreur. On rappelle que le renommage d'un fichier est une succession d'opérations sur les liens physiques, grâce aux primitives `link` et `unlink`.

Où faut-il modifier le fichier `Makefile` pour ajouter votre commande ? Vous noterez une particularité dans le nom de l'exécutable pour votre commande : pouvez-vous en trouver la raison ?

Chapitre 2 - Tables de pages

Exercice 2.1

Xv6 ne dispose pas de commande `ps` pour afficher les processus, mais reconnaît pour ce faire la touche `Ctrl P` à la console, qui provoque l'appel de la fonction `procdump` dans `proc.c`.

L'objet de l'exercice est de remplacer cet affichage par le nombre de pages libres. Commencez par renommer dans `proc.c` la fonction `procdump` (en `xprocdump` ou ce que vous voulez). Puis, ajoutez une nouvelle fonction `procdump` dans `kalloc.c` pour compter et afficher (avec `cprintf`) le nombre d'entrées dans la liste des pages libres (variable `kmem`). Vous pouvez ignorer les questions de verrouillage pour le moment.

Est-ce que vous retrouvez la taille (approximative) de la mémoire gérée par xv6 ?

Exercice 2.2

La fonction `setupkvm` est utilisée par chaque processus pour initialiser la partie supérieure (au-dessus de `KERNBASE`) de l'espace d'adressage. Elle utilise le tableau `kmap`. Dessinez la table de pages résultant de l'appel à `setupkvm`, avec les valeurs numériques associées (il ne doit pas y avoir de constantes symboliques sur votre schéma). Vous pouvez utiliser `gdb` en complément de la lecture du code de `setupkvm`, `mappages` et `walkpgdir`.

Exercice 2.3

La primitive `exec` construit un nouvel espace d'adressage en allouant une nouvelle table de pages. On souhaite ici afficher cet espace à la fin de la primitive en explorant la table de pages nouvellement créée et en affichant les caractéristiques des pages trouvées :

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
[0] -> 0xdf7a000 w u
[1] -> 0xdf78000 w s
[2] -> 0xdf77000 w u
init: starting sh
[0] -> 0xdf32000 w u
[1] -> 0xdf30000 w u
[2] -> 0xdf2f000 w s
[3] -> 0xdf2e000 w u
$ ls
[0] -> 0xdee2000 w u
[1] -> 0xdee0000 w s
[2] -> 0xdedf000 w u
```

Pour cela, on propose de rédiger la fonction `void printpgdir (pde_t *pgdir)` dans le fichier `vm.c`, en utilisant les macros du fichier `mmu.h` et `memlayout.h` (`NPENTRIES`, `NPTENTRIES`, `PTE_*`, `PTE_ADDR` et `P2V`) pour faciliter l'accès aux adresses et aux flags de la table de page. Vous appellerez cette fonction à la fin de `exec`. On notera sur l'exemple qu'on restreint l'affichage aux adresses virtuelles inférieures à 2 Go (`KERNBASE`), puisque les adresses au-delà correspondent aux adresses utilisées par le noyau.

Exercice 2.4

L'espace d'adressage d'un processus avec xv6 commence à l'adresse 0, qui est de ce fait valide et empêche de détecter les problèmes de pointeurs non initialisés grâce au fameux « *segmentation fault* ».

Modifiez xv6 afin que le déréférencement de l'adresse virtuelle nulle se traduise par un arrêt du programme. Concrètement, il faut que la première page du processus soit simplement absente.

Attention : les programmes utilisateurs fournis avec xv6 sont générés pour démarrer à l'adresse nulle, qui est maintenant invalide. Il faut donc modifier l'appel de l'éditeur de liens, pour la cible « `_%: %o $(ULIB)` », afin d'indiquer que la section `text` du fichier ELF généré démarre à l'adresse 4096.

Chapitre 3 - Trappes, interruptions et périphériques

Exercice 3.1

Cet exercice et les suivants nécessitent, pour la simplicité, que le nombre de processeurs de la machine virtuelle émulée par qemu soit fixé à 1 (voir exercices du chapitre 1).

1. Modifiez la fonction `trap` pour afficher le numéro de l'interruption ou de l'exception détectée.
2. Vous aurez noté que certains numéros reviennent très souvent : faites en sorte, pour les deux qui reviennent très souvent, de n'afficher que le nombre d'occurrences. Par exemple :

```
INT 12: 5 occurrence(s)
INT 34: 2 occurrence(s)
INT 56
INT 12: 8 occurrence(s)
INT 78
...
```

3. À quoi correspondent les deux interruptions qui reviennent très souvent ?
4. Où pouvez-vous obtenir le niveau de privilège (anneau 0 ou 3 en terminologie x86) du processus au moment où l'interruption a été prise en compte ? Affichez ce niveau de privilège en même temps que le numéro de l'interruption.
5. Vous noterez que le niveau de privilège affiché est souvent le même. Pourquoi ? Trouvez un moyen pour que l'autre niveau de privilège soit affiché de temps à autre.
6. À l'aide de gdb, tracez le cheminement entre la prise en compte de l'interruption par le processeur et l'entrée dans la fonction `trap`. Astuce : démarrez en posant un point d'arrêt sur `vector32`, par exemple.
7. Où et comment sont sauvegardés les registres ?

Exercice 3.2

Avec gdb, lancez xv6. Lorsque celui-ci est démarré, interrompez gdb avec `Ctrl C`, placez un point d'arrêt sur la fonction `sys_open`, puis reprenez l'exécution. Dans le terminal, affichez le fichier `README` avec la commande `cat`.

1. Tracez les appels de fonctions nécessaires pour récupérer les arguments de la primitive `open`.
2. Quels sont les tests effectués par les différentes fonctions ?

Exercice 3.3

À partir du code des fonctions `idestart`, `idewait` et `ideintr`, expliquez comment le logiciel utilise le contrôleur IDE : rôle et adresse de chaque port, et signification des bits qui composent la valeur le cas échéant.

On rappelle que l'instruction `in` (resp. `out`) du x86 sert à lire (resp. écrire) une valeur (de 8, 16 ou 32 bits) depuis (resp. vers) un port d'entrée/sortie, c'est-à-dire un registre du contrôleur. L'instruction `ins` (resp. `outs`), quant à elle, lit (resp. écrit) une succession de valeurs (de 8, 16 ou 32 bits) depuis (resp. vers) un port d'entrée/sortie.

Exercice 3.4

Lorsque le pilote IDE (fonction `iderw`) reçoit l'ordre de lire ou d'écrire un bloc sur le disque, il ajoute le buffer correspondant au bloc dans une liste référencée par `idequeue` puis, si la liste était vide, démarre la requête. Par la suite, si d'autres ordres sont donnés au pilote avant que la requête ne soit terminée, les buffers correspondants sont simplement ajoutés à la liste.

1. Dans la fonction `iderw`, entre l'ajout du buffer dans la liste et le démarrage de la requête, ajoutez un appel à une fonction `void afficher_compte(void)`. Rédigez cette fonction pour compter et afficher le nombre de buffers actuellement dans la liste référencée par `idequeue`.

2. À l'exécution, vous devez vraisemblablement voir beaucoup d'affichages indiquant la valeur 1. Vous observerez également que des lectures successives (par exemple `cat README`) ne génèrent pas d'affichage, ce qui signifie que les données se trouvent déjà dans le *buffer cache* et qu'il n'y a donc pas besoin d'appeler le pilote pour relire ces données.
3. Comme ces affichages sont nombreux et polluent le terminal, modifiez la fonction pour n'afficher que les valeurs supérieures à 1.
4. Livrez-vous maintenant à un petit jeu : cherchez un moyen de provoquer l'affichage, c'est-à-dire de générer plusieurs requêtes de lecture ou d'écriture (quasi-)simultanées.

Exercice 3.5

(exercice du livret xv6)

Ajoutez une nouvelle primitive système `int getdate(struct rtcdate *)` pour obtenir l'heure UTC actuelle et la renvoyer à un programme utilisateur. Ajoutez un nouveau programme utilisateur pour appeler cette primitive et afficher son résultat.

Vous souhaitez sans doute utiliser la fonction `cmostime` (fichier `lapic.c`) et la `struct rtcdate` (fichier `date.h`) pour lire le périphérique « horloge temps réel ». Décrivez la façon dont le logiciel interagit avec le matériel.

Chapitre 4 - Verrouillage

Exercice 4.1

(exercice du livret xv6)

Dans la fonction `iderw`, déplacez l'appel à `acquire` juste après la boucle d'ajout du buffer dans la liste.

1. Y a-t-il un problème de concurrence ? Si oui, expliquez entre quelles entités.
2. Analysez le fonctionnement du programme `stressfs.c` fourni avec xv6. Selon la rapidité de votre ordinateur, vous pouvez ignorer le commentaire en début de programme.
3. Exécutez la commande `stressfs` avec xv6. Que constatez-vous ? Selon la rapidité de votre ordinateur, vous devrez l'exécuter plusieurs fois pour constater un problème.

Exercice 4.2

L'objectif de cet exercice est d'ajouter à xv6 une fonctionnalité de compteur partagé entre processus. Chaque processus peut avoir un (unique) compteur qu'il manipule avec une nouvelle primitive `int shc(int)` :

- si l'argument est strictement négatif, un compteur est alloué pour le processus et remplace le compteur actuel. La valeur de retour est nulle.
- si l'argument est positif ou nul, le compteur est incrémenté avec la valeur fournie, et la nouvelle valeur du compteur est retournée

Bien sûr, en cas d'erreur, la valeur -1 est retournée. On notera que le compteur n'est pas *nommé*, comme peut l'être par exemple un fichier dans l'arborescence, ce qui signifie que le compteur ne peut être partagé que par des processus qui l'ont hérité, via `fork`, depuis un ancêtre unique.

Pour représenter l'ensemble des compteurs, on propose la structure de données suivante :

```
#define NSHC      10
struct {
    struct spinlock lock;           // verrouillage
    int shc [NSHC];                // valeur de chaque compteur, -1 si pas alloué
} shctable ;
```

Pour référencer le compteur utilisable par un processus, on propose d'ajouter le champ `int shc` dans le descripteur de processus (structure `proc`) : s'il contient -1, le processus n'a accès à aucun compteur, sinon il s'agit d'un indice dans la table `shctable.shc`.

Dans cet exercice, on s'intéresse uniquement à l'allocation et l'utilisation du compteur. La désallocation n'est pas prise en compte dans cet exercice, mais vous pouvez l'implémenter, ce qui nécessitera une modification de la structure de données.

1. Dans `proc.h`, ajoutez le champ `shc` à la structure `proc`.
2. Dans `proc.c`, ajoutez la structure de données ci-dessus, ainsi que la fonction `void shcinit(void)` pour l'initialiser. Appelez cette fonction dans `main.c`, juste avant l'appel de `startothers`.
3. Toujours dans `proc.c`, modifiez `userinit` pour initialiser à -1 le compteur du premier processus, ainsi que `fork` pour que le fils hérite du compteur du père.
4. Ajoutez maintenant la primitive `shc` à xv6. Vous la décomposerez en deux : fonction `sys_shc` dans `sysproc.c` et `shc` dans `proc.c`. Dans un premier temps, n'utilisez pas le verrou dans votre implémentation afin d'exhiber les problèmes de concurrence.
5. Énumérez tous les problèmes de concurrence posés par cette implémentation sans verrouillage.
6. Ajoutez un programme utilisateur `shc` à xv6 : ce programme doit générer 4 processus fils chargés d'incrémenter (de 1) 10 000 fois un compteur partagé alloué par le processus père. Une fois les fils terminés, le processus père doit afficher la valeur du compteur.
7. Exécutez ce programme sur une machine virtuelle à 1 processeur. Que se passe-t-il ? Exécutez-le sur une machine virtuelle à 4 processeurs. Que se passe-t-il ?

8. Implémentez maintenant le verrouillage pour réduire tous les problèmes de concurrence posés par l'implémentation précédente.
9. Avec la structure de données proposée, deux compteurs indépendants ne peuvent être incrémentés simultanément, ce qui limite le parallélisme. Modifiez l'implémentation pour permettre l'incrément en parallèle.

Chapitre 5 - Ordonnancement

Exercice 5.1

(exercice du livret xv6)

La fonction `sleep` doit tester `lk != &ptable.lock` pour éviter un interblocage.

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 ?

Exercice 5.2

(exercice donné lors du TP noté 2019/2020).

Les primitives `exit` et `wait` de xv6 diffèrent de leur équivalent POSIX car elles ne prennent pas d'argument. L'objectif de cette question est de combler cette lacune. On rappelle que le code transmis à `exit` est « remonté » au père lorsque celui-ci appelle `wait`.

Dans cette question, l'argument fourni à `exit` doit être un entier (sans restriction). L'argument fourni à `wait` est quant à lui soit 0, soit un pointeur sur un entier qui contiendra en sortie¹ l'argument de `exit`. Si un processus doit s'arrêter pour une raison « anormale » (défaut de page, etc.), l'entier doit valoir -1.

1. Les primitives `exit` et `wait` devant recevoir un argument, tous les programmes « utilisateurs » de xv6 doivent être modifiés pour passer un argument à chaque appel de ces primitives, sinon vous ne pourrez pas compiler votre version de xv6. Pour vous aider dans cette tâche fastidieuse, vous trouverez ci-après le script shell `param-exit-wait.sh` que vous pouvez utiliser avant de commencer.
2. Faites ensuite un `git commit -a` : les programmes utilisateurs n'étant plus modifiés par la suite, les commandes `git status` et `git diff` ne mentionneront plus ces programmes.
3. Faites en sorte que xv6 compile et démarre avec la nouvelle syntaxe de `exit` et `wait`.
4. Ajoutez le programme `q1` (source `q1.c` ci-après) aux programmes utilisateurs afin de tester facilement l'argument de `exit` et `wait`.
5. Implémentez enfin l'argument de `exit` et `wait`. N'oubliez pas que le pointeur passé à `wait` peut être nul.

Fichier `param-exit-wait.sh`

```
#!/bin/sh

#
# Script shell pour ajouter l'argument 0 à tous les appels de exit
# et wait dans les programmes utilisateurs de xv6.
#
# Note : ce script pourrait être plus court si tous les systèmes
# disposaient de GNU-sed (option -inplace).
#
```

1. La ressemblance avec POSIX s'arrête là puisque l'argument de `exit` est une valeur comprise entre 0 et 255, et l'entier renvoyé par `wait` combine plusieurs informations, notamment la raison pour laquelle le processus s'est arrêté, le numéro de signal, etc. que l'on ne demande pas ici.

```

LISTE="
        forktest.c\
        cat.c\
        echo.c\
        forktest.c\
        grep.c\
        init.c\
        kill.c\
        ln.c\
        ls.c\
        mkdir.c\
        rm.c\
        sh.c\
        stressfs.c\
        usertests.c\
        wc.c\
        zombie.c\
"

for f in $LISTE
do
    sed -e 's/exit();/exit(0);/' \
        -e 's/\([^a-z]\)wait()/\1wait(0)/' \
        $f > $f.bak && mv $f.bak $f
done

exit 0

```

Fichier q1.c

```

#include "types.h"
#include "user.h"

// Programme pour tester le fonctionnement de exit/wait avec un
// argument.
//
// Prend un argument entier sur la ligne de commande, la valeur
// que exit doit renvoyer a wait. Si tout se passe bien :
// q1 123
// doit afficher :
// exit(123) -> wait retourne 123

int
main(int argc, char *argv[])
{
    int n, status;

    if(argc != 2){
        printf(2, "usage: %s\n", argv[0]);
        exit(0);
    }

    n = atoi(argv[1]);

    switch (fork())
    {
        case -1:
            printf(2, "cannot fork\n");
            exit(0);
        case 0:
            exit(n);
        default:
            if(wait(&status)==-1) {
                printf(2, "cannot wait\n");
            }
    }
}

```



```

    printf(1, "exit(%d)\n->\nwait\retourne\nd\n", n, status);
}
exit(0);
}

```

Exercice 5.3

(exercice donné lors du TP noté 2019/2020).

La primitive POSIX `getrusage` permet à un processus père de connaître les ressources systèmes consommées par l'ensemble des processus fils (et leur descendance) terminés et dont le père a enregistré la terminaison par `wait`. L'objectif est d'en implémenter une version très simplifiée.

Cette nouvelle primitive `int getrusage(struct rusage *)` reçoit un pointeur vers une structure que vous placerez dans le nouveau fichier `resource.h` :

```

struct rusage {
    int ru_utime;    // Time spent in user mode
    int ru_stime;    // Time spent in kernel mode
    int ru_maxrss;   // Maximum resident set size
};

```

On rappelle que ces valeurs contiennent le cumul des ressources consommées par l'ensemble des processus fils (et leur descendance) terminés. Les deux premiers champs sont comptés en nombre d'interruptions d'horloge survenus lorsque les processus étaient sur le processeur. Le dernier champ doit contenir le nombre de pages (y compris PDE et PTE) utilisées par les processus (mais sans compter les pages de la partie utilisée par le noyau, au delà de `0x8000 0000`).

1. Téléchargez le fichier `resource.h`, disponible sur Moodle, contenant la définition de la structure ci-dessus.
2. Ajoutez une première version de `getrusage` dans `sysproc.c`. Pour le moment, contentez-vous de renvoyer la valeur 0.
3. Ajoutez le programme `q2` (source `q2.c` ci-après) aux programmes utilisateurs afin de tester facilement votre primitive. Testez la compilation de l'ensemble.
4. Ajoutez ensuite la mesure des compteurs de temps processeur : vous devrez déterminer le mode d'exécution du processeur lorsque l'interruption d'horloge survient.
5. Ajoutez enfin le calcul de la consommation mémoire : vous pourrez simplifier le problème en parcourant la table des pages avant que la mémoire du processus ne soit libérée.

Fichier q2.c

```

#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "resource.h"

#define MILLION      (1000*1000)

// Programme pour tester le fonctionnement de getrusage.
//
// Prend 3 arguments entiers :
// - nombre d'itérations pour consommer du temps en mode utilisateur
// - nombre d'itérations pour consommer du temps en mode système
// - nombre de pages à allouer
//
// Exemple :
// > q2 1 1 0 (test cpu user + cpu sys)
// user=65, sys=123, maxrss=5
// > q2 0 0 3 (test maxrss)
// user=0, sys=0, maxrss=8
// > q2 10 5 17 (la totale)

```

```

// user=752, sys=610, maxrss=22

int
cpuuser(void)                                // passe du temps en mode utilisateur
{
    int s = 0;
    for (int i = 0 ; i < 100*MILLION ; i++)
        s += i;
    return s;
}

void
cpusys(void)                                // passe du temps en mode systeme
{
    int fd;
    char c;

    fd = open("/README", O_RDONLY);
    if (fd == -1) {
        printf(2, "cannot open README\n");
        exit(1);
    }
    while(read (fd, &c, 1) > 0)
        ;
    close (fd);
}

void
consomme(int npages, int nuser, int nsys)
{
    int i, s ;

    if (sbrk (npages * 4096) == 0) {
        printf (2, "cannot sbrk %d pages\n", npages);
        return;
    }

    s = 0;
    for (i = 0 ; i < nuser ; i++)
        s += cpuuser();

    for (i = 0 ; i < nsys ; i++)
        cpusys();

    exit(s);
}

int
main(int argc, char *argv[])
{
    int npages, nuser, nsys;
    int status;
    struct rusage u;

    if(argc != 4){
        printf(2, "usage: %s nuser nsys npages\n", argv[0]);
        exit(0);
    }

    nuser = atoi(argv[1]);                    // nb d'itérations (* 100 million) en mode U
    nsys = atoi(argv[2]);                     // nb d'itérations (* 10) en mode S
    npages = atoi(argv[3]);                   // nb de pages a allouer

    switch (fork())
    {
        case -1:

```

```
    printf(2, "cannot_fork\n");
    exit(1);
case 0:
    consomme (npages, nuser, nsys);
    exit(0);
default:
    if(wait(&status)==-1) {
        printf(2, "cannot_wait\n");
        exit(1);
    }
    // printf(1, "exit(%d)\n", status);
    if(getrusage(&u) == -1)
        printf(2, "cannot_rusage\n");
    printf(1, "user=%d, sys=%d, maxrss=%d\n",
            u.ru_utime, u.ru_stime, u.ru_maxrss);
}
exit(0);
}
```


Chapitre 6 - Système de fichiers

Exercice 6.1

(exercice du livret xv6)

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?

Exercice 6.2

(exercice du livret xv6)

Dans `sys_link`, supposons que le fichier correspondant à `ip` soit supprimé par un autre processus entre les appels à `iunlock(ip)` et `dirlink`. Le lien sera-t-il créé correctement? Expliquez pourquoi.

Exercice 6.3

(exercice du livret xv6)

Ajoutez les flags `O_TRUNC` et `O_APPEND` à `open`, de façon que les opérateurs `>` et `>>` fonctionnent en Shell.

On rappelle que le flag `O_TRUNC` tronque le fichier s'il existait lors de l'ouverture et le flag `O_APPEND` fait que `write` écrit systématiquement à la fin du fichier (même si le fichier est agrandi par ailleurs, par exemple par un autre processus).

Vous devrez modifier le Shell, et plus spécifiquement la fonction `parseredirs`.