Programmation Système I : processus

Juliusz Chroboczek

3 février 2016

Pour une introduction à la programmation système, voyez le début de mes notes de cours de L3[1]. Pour des références bibliographiques, voyez la page web du cours[2].

1 Programmes et processus

Un *programme* est un fichier contenant du code pouvant être exécuté. Par exemple, le fichier `a.out` généré par le compilateur est un programme, comme le fichier `/bin/ls`. Lorsqu'il est *exécuté*, le programme est chargé en mémoire dans une structure de données qui s'appelle un *processus*. Il y a à tout moment un seul programme qui s'appelle `emacs` sur mon portable, mais il peut y avoir zéro, un ou plusieurs processus qui exécutent Emacs.

1.1 Structure d'un programme

De nos jours, les programmes utilisent des formats de fichier compliqués, comme *ELF* (sous Unix) ou *COFF* (sous Windows). Les Unix plus anciens utilisaient un format appelé *a.out* (pour *Assembler Output*)[3].

Le format *a.out* a changé plusieurs fois au cours de son histoire, dans ce paragraphe je décris la variante utilisée par 2.11BSD (la dernière version d'Unix BSD qui pouvait s'exécuter sur un PDP-11). Le fichier commence par un entête qui a la forme suivante :

```
struct exec {
    int a_magic;
    unsigned int a_text;
    unsigned int a_data;
    unsigned int a_bss;
    unsigned int a_syms;
    unsigned int a_entry;
    unsigned int a_unused;
    unsigned int a_flag;
};
```

1. `http://www.pps.univ-paris-diderot.fr/~jch/enseignement/programmation-systeme.pdf`
2. `http://www.pps.univ-paris-diderot.fr/~jch/enseignement/systeme/`
3. Du coup, un fichier appelé `a.out` est de nos jours un fichier ELF.

Comme le type `int` a une taille de 16 bits sur un PDP-11, l'entête a une taille de 14 octets. Le champ `a_magic` contient un nombre magique qui permet d'identifier un fichier *a.out*. Le champ `a_text` contient la longueur de la section `text` ; le champ `a_data` contient la longueur de la section `data`, et enfin le champ `a_bss` contient la longueur de la section `bss`. Les détails des autres champs ne nous intéressent pas dans ce cours.

L'entête est suivi du contenu de la section `text`, de longueur `a_text`, qui contient le code du programme, et ensuite du contenu de la section `a_data`, qui contient les valeurs des variables globales initialisées. La section `bss`, qui contient les variables globales non-initialisées, n'est pas stockée dans le programme.

1.2 Carte mémoire d'un processus

Lorsqu'un programme est exécuté sous 2.11BSD, le système crée un espace de mémoire virtuelle dans lequel il pourra s'exécuter (voir paragraphe 2 ci-dessous) qu'il peuple de la façon suivante :

- la section `text` est chargée à l'adresse 0, et protégée en écriture – seules les lectures et l'exécution sont autorisées ;
- la section `data` est chargée après la section `text`, à la première adresse multiple de la taille de page (8192 sur un PDP-11, voir paragraphe 2 ci-dessous) ; les écritures et les lectures y sont autorisées ;
- la section `bss` est placée après la section `data`, et son contenu est initialisé à 0 ;
- la section `bss` est suivie d'une section `stack`, qui sert de pile d'appels de fonction et de *tas* pour les allocations effectuées à l'aide de `malloc`.

Quelques améliorations ont été apportées à ce schéma après 2.11BSD. La principale est que la section `text` n'est plus chargée à l'adresse 0 — la page 0 est protégées en lecture et en écriture, ce qui permet de détecter les déréférencements du pointeur nul.

2 La mémoire virtuelle

Les adresses mémoire visibles par le logiciel (les paramètres des instructions de chargement et de stockage du langage machine, ou les valeurs des pointeurs en C) sont des adresses dites *virtuelles* ; elles sont différentes des adresses de mémoire dites *physiques* que voit le matériel. Un composant appelé le MMU *(Memory Management Unit)* s'occuppe de convertir les adresses virtuelles en adresses physiques (figure 1).

La mémoire virtuelle est divisée en unités appelées *pages*, qui ont une taille de 4096 ou 8192 octets selon l'architecture. À chaque processus est associée une *table de pagination* qui spécifie la page de mémoire physique qui correspond à chaque page virtuelle. Plus précisément, une entrée de page contient les données suivantes :

- un bit qui dit si la page est *valide* ; si ce n'est pas le cas, tout accès à cette page invoque un gestionnaire du noyau ;
- si la page est valide, le numéro de la page de mémoire physique correspondant ;
- des bits qui indiquent les permissions de la page (lecture, écriture, exécution) ;
- des informations supplémentaires qui indiquent si la page a été lue ou écrite récemment.

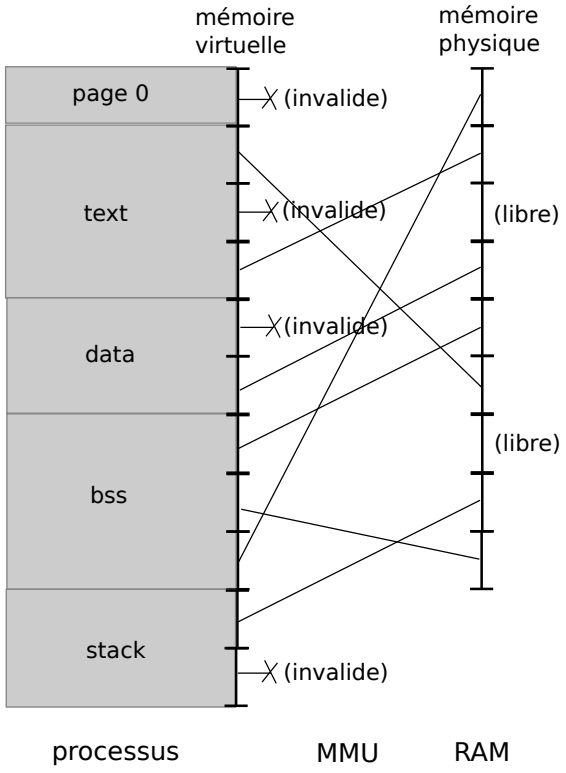

Figure 1 — Mémoire virtuelle d'un processus. (Les pages invalides des sections `text` et `data` seront chargées de manière paresseuse, voir paragraphe 5.)

2.1 Pagination

La présence d'un MMU et l'indirection de l'accès à la mémoire physique à travers la table de pagination est utilisée de plusieurs façons par le système. Tout d'abord, la page 0 de chaque processus ainsi que les pages qui ne correspondent à aucune section du programme sont marquées invalides, ce qui permet au système de capturer les accès à ces pages et d'envoyer un signal `SIGSEGV` (« violation de segmentation[4] ») au processus coupable. Ensuite, lorsqu'un programme est chargé en mémoire, les pages qui correspondent à la section `text` sont partagées entre tous les processus qui exécutent ce même programme, ce qui permet d'économiser de la mémoire — comme la section `text` est protégée en écriture, aucune interférence accidentelle n'est possible.

Une autre optimisation est la *pagination sur disque*. Lorsque le système se trouve à court de mémoire, il identifie des pages auxquelles le code utilisateur n'a pas accédé depuis un moment. Si ces pages sont dans une section texte (pages « *file-backed* »), il les marque invalides et récupère la mémoire physique correspondante ; sinon (pages « *swap-backed* »), il sauvegarde les données qu'elles contiennent dans une zone spécifique du disque, la *swap*[5], avant de les marquer invalides

4. La segmentation est une technique de gestion de la mémoire virtuelle obsolète, qui a été remplacée par la pagination, mais l'ancienne terminologie reste utilisée.
5. Le *swapping* est une technique encore plus obsolète que la segmentation.

et de récupérer la mémoire physique.

Lorsque le programme utilisateur essaie d'accéder à une page marquée invalide, le MMU interrompt son exécution et passe la main au noyau. Le noyau détermine la nature de la page manquante : si elle est *file-backed*, il recharge la page depuis le fichier exécutable, si elle est *swap-backed*, il la recharge depuis la *swap*, puis relance le programme à l'endroit où il a été interrompu. Enfin, si la page n'était ni *file-* ni *swap-backed* (par exemple, c'est la page 0), il envoie le signal `SIGSEGV` au processus.

Une technique plus subtile, le *copy-on-write* (COW) est utilisé pour partager les pages des sections `data` et `bss` ; nous en parlons au paragraphe 4.1 ci-dessous.

3 L'ordonnanceur

Le système manipule plusieurs processus à la fois. Sur un système à un seul processeur (ou « cœur »), il utilise une technique appelée le *temps partagé*.

À tout moment, un processus est en train de s'exécuter. Lorsque le processus fait un appel-système bloquant, ou lorsque sa *tranche de temps (time slice)* est épuisée, le système interrompt ce processus, sauvegarde l'état du processeur et la table de pagination, sélectionne un nouveau processus, charge l'état du processeur et la table de pagination du nouveau processus et relance son exécution.

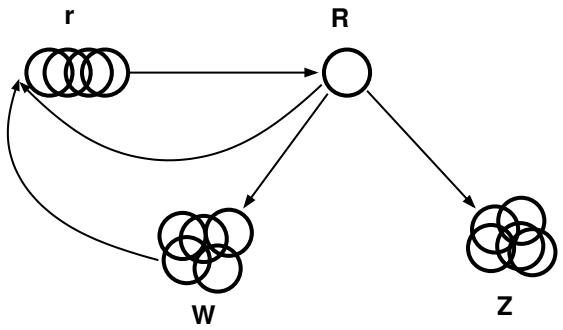

Figure 2 — Structures de données de l'ordonnanceur

Le module qui s'occuppe de choisir les processus à exécuter s'appelle l'*ordonnanceur (scheduler)*. L'ordonnanceur maintient plusieurs files de processus (figure 2) :

- le processus en train de s'exécuter, dans l'état *R* (« *running* ») ;
- la file de processus prêts à s'exécuter, dans l'état *r* (« *runnable* »), parfois appelé *S* (« *sleeping* ») ;
- les processus qui ont exécuté un appel système bloquant, et qui sont donc en attente d'un événement (un bloc de données qui arrive depuis le disque, un paquet qui arrive depuis le réseau, etc.), dans l'état *W* (« *waiting* »).

Un processus dans l'état *Z* (« *zombie* ») n'est pas vraiment un processus — c'est juste une petite structure de données qui contient le numéro et le résultat d'un processus mort.

Dans le cas d'un système à plusieurs processeurs, il y a un ordonnanceur par processeur. Il y a alors une transition supplémentaire : un processus peut migrer d'un processeur vers un autre, par exemple pour égaliser la charge des processeurs.

La commande `ps algx` permet de consulter l'état de tous les processus du système.

4 Vie et mort des processus

4.1 Création d'un processus

Un processus est créé à l'aide de l'appel système `fork`, que vous avez vu durant le cours de L3. Je n'en rappelle donc pas l'utilisation.

Lors d'un appel à `fork`, la structure de données associée au processus est dupliquée : l'état du processeur, et la table de pagination. Les pages physiques correspondant à la section `text` ne sont pas dupliquées — elles sont protégées en lecture, on peut donc les partager entre processus.

Ce qui est plus surprenant, c'est que les pages physiques des autres sections (`data`, `bss` et `stack` ne sont pas dupliquées non plus — elles sont simplement protégées en lecture et partagées. Si un processus lit une telle page, tout se passe bien ; si par contre il essaie de l'écrire, le système prend la main, duplique la page, puis relance le programme — on dit que le système a géré un *défaut de page (page fault)*. Cette technique consistant à dupliquer une page de façon paresseuse lorsqu'elle est écrite s'appelle le *copy-on-write*.

La principale conséquence du *copy-on-write* est que la séquence `fork`/`exec` est relativement efficace : la table de pagination du fils est détruite lors de l'invocation d'`exec` (voir paragraphe 5), ce qui fait que la plupart des pages *copy-on-write* ne seront jamais dupliquées.

4.2 Mort d'un processus

Un processus meurt lorsqu'il invoque l'appel système `_exit`. La fonction `exit`, que vous avez l'habitude d'employer, invoque les fonctions enregistrées à l'aide d'`atexit` puis invoque `_exit`. Elle permet notamment de vider les tampons de la bibliothèque `stdio` avant de terminer[6].

Digression : la fonction `_start` Lorsqu'un processus est exécuté, ce n'est pas la fonction `main` qui est lancée, mais une fonction normalement appelée `_start` qui est contenue dans la bibliothèque standard. La fonction `_start` effectue les actions suivantes :

- elle calcule `argc`, `argv` et `environ` d'une façon dépendant du système ;
- elle appelle `main(argc, argv, environ)` ;
- si `main` retourne, elle invoque `exit` avec le résultat du `main`.

Ce dernier point explique pourquoi retourner depuis `main` est équivalent à une invocation d'`exit`.

4.3 Synchronisation avec la mort d'un processus

Le père d'un processus doit se synchroniser avec la mort de celui-ci. Pour cela, il invoque l'appel système `wait`, qui a l'un des comportement suivants :

6. Vous êtes au point sur les tampons ? Si ce n'est pas le cas, j'en parle dans mes notes de cours de L3.

- si le processus appelant a un ou plusieurs fils dans l'état *Z* (zombie), un zombie est supprimé et l'appel à `wait` retourne immédiatement ;
- sinon, si le processus à un ou plusieurs fils, il est mis en attente de la mort d'un fils : l'appel système *bloque* et le processus est mis dans l'état *W* ;
- sinon, l'appel système `wait` retourne l'erreur `ESEARCH`.

Le double fork Si le père d'un processus meurt avant son fils, le fils est « adopté » par le processus *init*, de *pid* 0, dont la seule tâche est de collecter les zombies. Cette propriété permet de se débarasser d'un fils en effectuant un *double fork* : créer un processus qui lui-même crée un processus et meurt immédiatement ; le petit fils sera adopté par *init*. La plupart des *démons* Unix sont implémentés à l'aide d'un double fork.

5 Exécution d'un programme

Un programme est exécuté en invoquant l'appel système `execve`. Cet appel système détruit la table de pagination du processus courant, et en crée une nouvelle selon la structure décrite au paragraphe 1.2. Les fonctions `execvp`, `execlp`, etc. sont des interfaces plus commodes à l'appel système `execve`.

Optimisation : chargement paresseux Sur un système à mémoire virtuelle, la mémoire physique n'est pas peuplée immédiatement lors d'un appel à `execve`. La plupart des pages sont marquées invalides, et ne seront chargées depuis le disque (dans le cas des sections `text` et `data`) ou initialisées à 0 (dans le cas des sections `bss` et `stack`) que lorsque le programme essaiera d'y accéder pour la première fois. Ce chargement paresseux est analogue à la duplication paresseuse effectuée lors d'un *copy-on-write*.

L'effet de ces optimisations peut être observé à l'aide de la commande `/usr/bin/time` (pas la commande `time` du *shell*), qui indique le nombre de défauts de page majeurs (qui ont requis le chargement d'une page depuis le disque) et le nombre de défauts mineurs (qui ont seulement requis l'initialisation d'une page à 0, sa duplication, ou sa copie depuis le cache du disque).