Architecture des Systèmes d'Exploitation

Devoir surveillé final

4 janvier 2017

N.B.
1. Tous les documents sont autorisés, ainsi que les calculatrices.
2. Les trois parties peuvent être traitées indépendamment.
3. Le sujet comporte un total de 24 points, mais je m'arrête de compter à 20.

Mémoire virtuelle - allocation et partage (10 points)

Dans cet exercice il s'agit d'implémenter un mécanisme d'allocation et de partage simple de pages de mémoire entre processus. Pour soliciter les fonctions du système, les processus utiliseront les interruptions logicielles.

deux d'interruptions logicielles sont définit :

- `SYS_ALLOC_NEXT_PAGE` et ;
- `SYS_FREE_LAST_PAGE`.

Considérez un processus qui dispose déjà de `n` pages de mémoire dans son espace, les pages de `0` à `n-1`.

Lorsque ce processus génère l'interruption `SYS_ALLOC_NEXT_PAGE` (via un `_int(SYS_ALLOC_NEXT_PAGE);`) il demande au noyau d'allouer une page de mémoire physique et de la mapper dans son espace d'adressage virtuel, aprés la dernière page (`n-1` donc en `n`). Si l'appel abouti (retourne -1) le processus en question dispose donc, à la suite de cet appel de `n+1` page de mémoire virtuelle. S'il echoue (retourne 0) le processus n'a pas de nouvelle page.

Lorsque le processus qui dispose de `n` pages de mémoire génère une interruption `SYS_FREE_LAST_PAGE`, il demande au noyau de libérer la dernière page de mémoire qu'il possède (la `n-1`). Ce faisant le système libère la page physique associée, et il retire le droit d'accès à la page `n-1` qui ne fait plus part des pages accessibles du processus.

Interruptions logicielles

Question 1. (1 point)

Les fonctions `SYS_ALLOC_NEXT_PAGE` et `SYS_FREE_LAST_PAGE` retournent un entier. Cependant l'instruction intel `int` (équvalente à la fonction _int) ne prend ni ne retourne de valeur *a priori*. Il est décidé que le registre `eax` sera utilisé pour comme valeur de retour aprés l'appel à l'interruption via une instruction `int`. Pour mémoire, on peut lire un registre (par exemple `eax`) du microprocesseur est le "ranger" dans une variable C comme suit :

```
asm("mov %%mon_registre,%0" : "=r" (ma_variable_c));
```

de même on peut "transférer" une variable C dans un registre (ici `eax`) :

```
asm("mov %0,%%mon_registre" : : "r" (ma_variable_c));
```

Proposez une implémentation de la fonction C suivante :

```
int _syscall(int irq, int arg);
```

Cette fonction déclanche l'interruption irq (via la fonction `void _int(int irq)`), mais elle passe à l'interruption le parametre arg, via le registre `eax`. une fois l'interruption exécuté elle assume que la valeur présente dans `eax` est la valeur "retournée" par l'interruption logicielle et c'est cette valeur qu'elle retourne (à l'appelant).

Gestion de la mémoire virtuelle

Pour pouvoir gérer la mémoire virtuelle de ses processus le système d'exploitation considéré utilise des structures de données internes. Ces structures sont au nombre de deux :

1. La première permet de connaitre la liste des pages physique libres afin de pouvoir gérer les mécansimes d'allocation et de libération des pages physiques ;
2. La seconde permet de connaitre les pages de mémoire physiques associées aux pages de mémoires virtuelles pour chaque espage d'adressage du mode "utilisateur".

Question 2.1. (0,5 point)

proposez une structure de donnée qui permettre de lister l'ensemble des pages de mémoire physiques libre, d'en retirer une (allocation) ou d'en ajouter une (libération). Pour cela nous retenons simplement l'idée d'une liste chainée dont le chainage est "stocké" dans les pages physiques, à la manière des blocs de mémoire libre, sur le disque, dont le chainage est stocké dans chaque bloc libre... Pour cela : Définissez la structure de donnée `struct free_page {...}` qui est stockée dans chaque page libre et la variable globale `first_free_page` qui permet de trouver la première page libre dans cette liste.

Question 2.2. (0,5 point)

Chaque espace d'adressage utilisateur, c'est à dire chaque configuration de mémoire virtuelle définit par le noyau, est caractérisé par :

1. une adresse de base, toujours la même : `virtual_memory` ;
2. un nombre `nb_pages` de pages virtuelles disponibles, aprés cette base, ce nombre ne pouvant exceder `MAX_VIRTUAL_PAGES` ;
3. une table `page_set` listant les `nb_pages` physiques que le noyau a associé aux `nb_pages` virtuelles de l'espace d'adressage utilisateur.

Proposez une structure de donnée

```
struct user_space_mapping_s {
  ...
} ;
```

qui puisse être utilisée par le noyau pour déterminer, pour chaque espace d'adressage utilisateur, quel est l'adresse de la page physique `ppage` associée a la page virtuelle `vpage`, lorsqu'il y en a une.

puis proposez une variable globale `address_space` qui permette de connaitre l'espace d'adressage (via la structure `user_space_mapping_s`) de `NB_MAX_PROCESS` espaces d'adressages différents au maximum.

Expliquez simplement comment s'utilise votre structure de donnée et la variable `user_space_mapping`.

Question 3. (1 point)

Donnez une implémentation de la fonction c `void init_address_space()` qui initialise l'espace d'adressage des pages de mémoire physique disponible afin d'en permettre l'allocation. Il s'agit donc d'initialiser non seulement `first_free_page` mais aussi l'ensemble des pages physiques qui pourront être allouées par la suite. Vous considérerez que les pages physiques sont au nombres de `NB_PHYSICAL_PAGES` et que la première commence à l'adresse `physical_memory` mais, comme nous l'avons vu en TD/TP cette première page (la page 0) est réservé au noyau puisqu'elle contient `IRQVECTOR`, en conséquence de quoi elle ne peut pas être allouée. De plus cette fonction initialisera

la variable `address_space` de tel sorte que l'ensemble des `NB_MAX_PROCESS` soient initialisé avec un mapping vide (aucune page de mémoire physique n'est associée à une page de mémoire virtuelle).

Question 4. (1 point)

Définissez la fonction `void *alloc_page()` qui alloue une page de mémoire physique en utilisant `first_free_page`. Cette fonction retourne l'adresse de la page de mémoire physique allouée ou NULL s'il n'y a plus aucune page de disponible.

Question 5. (1 point)

Définissez la fonction `void free_page(void *page)` qui libere une page de mémoire physique en utilisant `first_free_page`. Cette fonction prend en paramêtre l'adresse de la page de mémoire physique à libérer.

Question 6. (1 point)

Donnez une implémentation de la fonction

```
int alloc_next_page(int process_number);
```

Cette fonction est appelée avec le numero de processus en cours d'exécution, (notament) lorsque le programme utilisateur déclanche l'interruption `SYS_ALLOC_NEXT_PAGE`. Elle retourne `0` si tout c'est bien passé, et `-1` sinon.

Question 7. (1 point)

Donnez une implémentation de la fonction

```
void free_last_page(int process_number);
```

Cette fonction est appelée, avecc le numero du processus en cours d'exécution, (notament) lorsque le programme utilisateur déclanche l'interruption `SYS_FREE_LAST_PAGE`.

Partage de mémoire virtuelle

Pour partager une page de mémoire, on souhaite enrichir le fonctionnement de `SYS_ALLOC_NEXT_PAGE` et de `SYS_FREE_LAST_PAGE`.

L'interruption `SYS_ALLOC_NEXT_PAGE` accepte maintenant un parametre `shared_page_id`. Si ce parametre est égal à 0, l'allocation d'une page de mémoire virtuelle fonctionne comme auparavant. Par contre si ce nombre est différent de 0 (et inférieur à `NB_MAX_SHARED_PAGE_ID`) alors la page ajoutée par `SYS_ALLOC_LAST_PAGE` est une page de mémoire partagée associé à l'identifiant `shared_page_id`.

Lorsqu'un processus appelle `SYS_ALLOC_NEXT_PAGE` avec un identifiant déjà utilisé par un autre processus, alors la page qui lui est ajoutée est **la même page physique** que celle qui a été ajoutée au premier processus. Ainsi les deux processus partagent maintenant une page de mémoire.

Lorsqu'un processus appelle `SYS_FREE_LAST_PAGE` sur une page partagée, alors la page est retirée de son espage d'adressage. Cependant, la page de mémoire physique qui lui était associée ne doit être libérée qui s'il n'existe plus aucun espace d'adressage virtuel qui ne l'utilise. Pour cela, un simple "compteur de référence" associé à chaque page partagée sera incrémenté lorsqu'un processus ajoute cette page partagée dans son espace, et décrémenté lorsqu'il la retire. lorsque ce compteur atteint `0` c'est qu'il n'y a plus aucun espace d'adressage qui utilise cette page de mémoire partagée, et c'est à ce moment la seulement qu'elle est vraiment libérée.

Question 8. (1 point)

Complétez le code C suivant (en remplacant les marqueurs `<ICI>` par le code manquant) :

```
...
#define NB_MAX_SHARED_PAGE_ID 1024
...
struct shared_page_s {
```

```
<ICI>
} ;
...
struct shared_page_s shared_page[<ICI>];
...
```

afin de déclarer un tableau qui permettre d'associer l'adresse en mémoire physique d'une page à un `shared_page_id` (compris entre 0 - pas de partage- et 1023). Ce tableau vous permettra aussi de faire le comptage de référence à l'entrée.

Proposez encore une modification de la structure `struct user_space_mapping` afin de pouvoir retrouver rapidement l'entrée du tableau `shared_page` lorsque la page de mémoire virtuelle est mappée sur une page de mémoire physique partagée.

Question 9. (1 point)

Proposez une implémentation de la fonction

```
void alloc_next_page_shared(int process, int shared_page_id) ;
```

Cette fonction est appelée lorsqu'un processus demande à allouer une nouvelle page de mémoire virtuelle en l'associant à la page de mémoire physique `shared_page_id` (qui est donc différent de `0`).

Question 10. (1 point)

Proposez une nouvelle implémentation de la fonction

```
void free_last_page(int process);
```

afin qu'elle gére, le cas écheant, le fait que la page de mémoire virtuelle est associée à une page de mémoire physique. Pour mémoire, dans ce cas, elle ne doit être libérée que s'il n'existe plus aucun autre mapping de cette page dans aucun autre espace d'adressage virtuel.

Ordonnancement — Gestion de signaux (6 points)

Dans cet exercice il vous est demandé d'enrichir le fonctionnement de l'ordonnanceur vu en TDs/TPs avec un mécanisme de gestion des signaux. Un signal a le même comportement apparant qu'une interruption qui serait émise par un contexte A (la source) à destination d'un contexte B (le programme interrompu).

Vous baserez vos implémentations sur la version "ordonnancement sous interruptions" des travaux pratiques d'ASE. Vous ne vous préoccuperez pas des sémaphores.

Table des contextes

La première nécessité est de pouvoir désigner un contexte depuis un autre contexte, depuis l'espace utilisateur. Un contexte sera désigné par un identifiant `ctx_id` qui est un index dans la table des contextes gérée par le système :

```
typedef ctx_id int;
struct ctx_s tctx [MAX_CTX];
```

En conséquence la primitive de creation des contextes `create_context(...)` sera modifiée comme suit :

```
ctx_id create_context(...);
```

Elle ne retourne plus l'adresse d'un contexte, mais un identifiant de contexte.

Question 11. (1 point) Dans les systèmes d'exploitation utilisant la mémoire virtuelle pour séparer les processus, Pourquoi préférer désigner depuis l'espace utilisateur un contexte par un entier (`ctx_id`) plutôt que directement par l'adresse d'une structure `struct ctx_s` ?

Vous n'avez pas à décrire les modifications de la fonction `create_context()` engendrées par la gestion de cette table `tctx`.

Signaux utilisateurs

On distingue deux signaux pouvant être émis par un contexte à destination d'un autre contexte :

```
enum sgnl_type_e {SGNL_USR1,SGNL_USR2};
```

On introduit une primitive `c void signl_kill(ctx_id cid, sgnl_type_e sgnl);` qui envoie le signal `sgnl` au contexte désigné par l'identifiant `cid`.

Action à la réception d'un signal

La primitive

```
typedef void (*sgnl_handler_t) (int);
void signl_action(enum sgnl_type_e sgnl, sgnl_handler_t sgnl_handler);
```

sert à modifier l'action effectuée par le contexte courant à la réception d'un signal spécifique. Si la valeur `sgnl_handler_t` est `NULL`, le signal sera ignoré, sinon la fonction `sgnl_handler_t` sera exécutée. Cette exécution se fera dans la pile du contexte "interrompu".

Signaux en attente

Entre le moment où il est envoyé et celui où il est délivré, un signal est dit *en attente*. Si un signal est envoyé à un contexte alors qu'il est déjà en attente, le nouvel envoi n'est pas mémorisé. Autrement dit, si l'on envoi deux fois (ou plus) un signal donné à un processus avant alors que le premier signal est toujours *en attente*, il ne sera executé qu'une fois.

Structure de données

Pour chaque contexte, il est nécessaire de mémoriser pour chaque signal :

- la fonction à exécuter à la réception du signal (ou `NULL`) ;
- si le signal est en attente ou non.

Question 12. (1 point)

Proposez les modifications nécessaires des strutures de données pour mettre en œuvre cette gestion des signaux.

Kill et action

Question 13. (1 point)

Proposez une implémentation de la fonction `sgnl_kill()`.

Question 14. (1 point)

Proposez une implémentation de la fonction `sgnl_action` dont le prototype est :

```
void signl_action(enum sgnl_type_e sgnl, sgnl_handler_t sgnl_handler);
```

Exécution

Question 15. (2 points)

Afin que déclencher l'exécution de la fonction de type `sgnl_handler_t` lorsque le signal est *reçu* par le contexte destinataire, il convient de modifier `switch_to_ctx(...)`. Proposez une nouvelle implémentation de cette fonction.

Disque dur - adressage CHS, LBA28 et LBA48 (8 points)

Pour cet exercice vous vous baserez sur le système de fichiers implémenté en TDs/TPs, et plus précisément sur la première couche "driver". Le controleur de disque considéré reprend le même système de "port commande" et "port donnée". Simplement il implémente quelques commandes supplémentaires définies ci-aprés.

Pour les implémentations en C vous pourrez utilisez les fonctions `_in()`, `_out()` et `_sleep()`.

De plus vous pourrez considérez que :

- la macro `PORT_HDA_CMD` définit le port de commande du controleur de disque ;
- la macro `PORT_HDA_DATA` définit le port du premier octets de données (les 16 ports suivants correspondants aux octets de données consécutifs) ;
- la macro `HDA_IRQ` définit le niveau d'interruption associé au controleur de disque.

adressage des disques

adressage vu en TDs/TPs (CHS)

On considère la taille d'un secteur étant de 512 octets, avec 1024 secteurs par piste.

Question 16. (1 point)

En considérant que le numéro de cylindre est codé sur un entier 16 bits, quelle est la capacité maximale adressable théoriquement ?

adressage LBA 28

L'adressage par cylindre + secteur est appelé "CHS". Il a été détrôné au tournant des années 2000 (aprés 20 ans de bons et loyaux services) au profit de l'adressage dit "LBA".

Cet adressage identifie les secteurs de l'ensemble du disque avec un numéro unique de "bloc logique `b` dans le disque" et non de "secteur `s` de la piste `p` dans le disque"). Le numero de bloc logique est en fait le numéro de secteur depuis le début du disque au lieu d'être le numero de secteur de la piste `p`. Ainsi le bloc 1 correspond au secteur 1 piste 0 et le bloc 1025 correspond au secteur 1 piste 1 (si une piste contient 1024 secteurs).

Question 17. (1 point)

Le mode "LBA28" code un numéro de bloc sur 28 bits. Quelle est alors la capacité maximale d'un disque adressable théoriquement ?

adressage LBA 48

L'adressage LBA 28 s'est avéré insuffisant, et il est aujourd'hui de plus en plus emplacé par un adressage LBA 48. Dans ce cas le numéro du secteur est codé sur un nombre de 48bits.

Question 18. (1 point)

Quel est la capacité maximale de données adressables pour un disque supportant l'adressage "LBA48" qui code le numéro de secteur sur 48 bits ?

Conversion d'adressage

Question 19. (1 point)

Ecrivez le code de la fonction

```
uint32_t chs_to_lba(uint16_t cyl, uint16_t sec) ;
```

qui renvoit le numéro de secteur LBA28 correspondant au secteur CHS passé en paramètre.

Utilisation des adressages LBA

Vous considèrerez dans la suite de l'énnoncé que les disques supportant l'adressage LBA28 répondent à une commande supplémentaire `LBARDY` de code `0x18`. Aprés exécution de cette commande (qui ne génère pas d'interruption) il est possible de lire une valeur sur le premier port de donné. Cette valeur vaut `0` le disque ne connait que CHS. Si la valeur vaut `1` le disque supporte un adressage LBA28. Si elle vaut `2` il supporte LBA28 et LBA48.

Si LBA28 est supporté, c'est que le controleur de disque supporte une commande `SEEK28` de code `0x0A` qui prend sur 4 octets, avec 4 ports de données "l'adresse LBA28".

Si LBA48 est supporté c'est que le controleur de disque supporte en plus de la commande `SEEK28` précédente une commande `SEEK48` de code `0x0C` qui prend sur 6 octets, avec 6 ports de données, "l'adresse LBA48".

Question 20. (1 point)

Ecrivez le code de la fonction

```
int check_lba();
```

qui retourne `1` le disque dur passé en paramètre supporte l'adressage LBA28, `2` s'il support LBA28 et LBA48 et `0` s'il ne supporte que CHS.

Question 21. (2 points)

Proposez une nouvelle implémentation des fonctions

```
void seek_lba28(uint32_t lba28);
```

et

```
void seek_lba48(uint64_t lba48);
```

Question 22. (1 point)

Proposez enfin une nouvelle implémentation de

```
void read_sector_lba28(uint32_t lba28, unsigned char *buffer);

void write_sector_lba28(uint32_t lba28, unsigned char *buffer);
```

et

```
void write_sector_lba48(uint64_t lba48, unsigned char *buffer);

void format_sector_lba48(uint64_t lba48, unsigned int key);
```

Un `assert` assurera que ces fonctions ne sont jamais appelées lorsque le disque ne supporte pas l'adressage solicité.