Architecture des **S**ystèmes d'**E**xploitation
Maîtrise d'Informatique

Juin 2013

Devoir Surveillé

– tous documents autorisés –

Le langage C est requis pour les implémentations

Les différentes parties peuvent être traitées indépendamment, tous les documents sont autorisés, mais ne perdez pas trop de temps à chercher l'information utile. Notez encore que l'énoncé contient 24 points, je m'arrêterai à 20/20 ;-)

Exercice I. mémoire virtuelle (8 points)

Nous nous intéressons ici à la primitive posix `mmap` *utilisée pour manipuler un fichier (nommé «* `bigdata.bin` *») contenant une structure de donnée de 512Mo. Le programme qui utilise* `mmap` *est exécuté sur un serveur Intel utilisant un système Unix en mode protégé 32 bits. Selon les spécifications du processeur Intel considéré, en mode « protégé » 32 bits, l'adresse virtuelle est décomposée en 3 partis. Les 10 bits de poids forts forment l'index «* `i1` *» dans la table de premier niveau d'indirection de la MMU. Les 10 bits suivant forment l'index «* `i2` *» dans la table de second niveau dans l'adresse est indiquée à l'entrée «* `i1` *» de la table de premier niveau. Les 12 bits de points faibles forme l'offset dans la page de mémoire physique, l'adresse de la page de mémoire physique étant indiquée à l'entrée «* `i2` *» de la table de second niveau.*

Question I.1 (2 points)

Etant donnée la spécification du fonctionnement de la mémoire virtuelle d'un processeur Intel en mode « protégé » 32 bits, quel est la taille d'une page de mémoire ?

Question I.2 (2 points)

Si ce mécanisme de mémoire virtuelle n'utilisez pas de cache (TLB), combien d'accès à la mémoire physique serait fait à chaque fois qu'un programme accéderait à sa mémoire virtuelle ?

Question I.3 (2 points)

Selon vous, quel est le principal intérêt d'utiliser `mmap` plutôt que `fopen` pour ouvrir ce fichier « `bigdata.bin` » ?

Question I.4 (2 points)

Dans un serveur qui dispose de beaucoup de mémoire de travail (RAM), et peut donc y stocker entièrement le fichier, calculez :

1. Le nombre de page de mémoire physique nécessaire pour cela.
2. Le nombre de tables de second niveau qu'il faudra renseigner pour que le processeur puisse associer une adresse virtuelle à chaque adresse physique, selon les spécifications données en début d'énoncé.

N.B. Détaillez le raisonnement de vos calculs !

Exercice II. Ordonnancement EDF (8 points)

L'algorithme d'ordonnancement EDF (i.e. « Earliest Deadline First ») est un ordonnanceur utilisé dans les systèmes dit temps réel. Selon cet algorithme, chaque tache à une échéance (deadline). L'ordonnancement consiste à toujours choisir d'élire la tâche avec l'échéance la plus courte (il traite le plus urgent, en premier). Il ne passe au suivant, que lorsque la tâche est terminée ou que son échéance est dépassée. Pour l'exercice cette échéance sera simplement exprimée avec un entier (long) qui correspond au nombre de millisecondes écoulées depuis le début de l'ordonnancement.
Pour mémoire, voici ci-dessous un exemple fichier prototype de l'ordonnanceur (qui utilisait un algorithme d'ordonnancement « Round Robin ») réalisé en TP :

```
1.  #define CTX_MAGIC 0xCAD0AB0B
2.
3.  typedef void (t_fct) (void *);
4.
5.  enum ctx_state_e { /* enum des etats d'un contexte     */
6.    CTX_READY,       /*  . Contexte activable           */
7.    CTX_ACTIVABLE,   /*  . Contexte re-activable        */
8.    CTX_TERMINATED   /*  . Contexte termine             */
9.  };
10.
11. struct s_ctx {
12.   unsigned int     magic ;    /* detrompeur            */
13.   void            *savedESP ; /* valeur courante d'ESP */
14.   void            *savedEBP ; /* valeur courante d'EBP */
15.   t_fct           *startfct ; /* fonction de depart    */
16.   void            *arg ;      /* argument de depart    */
17.   char            *stack ;    /* pile d'execution      */
18.   enum ctx_state_e state ;    /* etat courant          */
19.   struct s_ctx    *nextCtx ;  /* chaînage pour "ring"  */
20. } ;
21.
22. /* anneau des contextes ordonnancables                 */
23. struct ctx_s * ring      = NULL ;
24. /* contexte d'exécution actuellement actif             */
25. struct s_ctx *currentCtx = NULL ;
26.
27. /* Creation d'un nouveau contexte à ordonnancer        */
28. int initCtx(struct s_ctx *aCtx, int stackSize, t_fct f,
       void *arg) ;
29. /* Implementation de la strategie d'ordonnancement     */
30. void yield() ;
31. /* Implem. du mecanisme de commutation de contexte     */
32. void switchToCtx(struct s_ctx *aCtx) ;
33. /* demarrage de la préemption de tâche                 */
34. void setupCtx() ;
```

listing Ctx.h

```
1.  [...]
2.  void yield() { /* implémentation du "round-robin" */
3.    switchToCtx(ring->next);
4.  }
5.  [...]
```

Implémentation de yield() vue en TP.

Question II.1 (2 point)

Pour que les tâches puissent fonctionner selon l'algorithme EDF, le programmeur doit pouvoir les créer en précisant une échéance, puis l'ordonnancement doit pouvoir tenir compte de cette échéance. Proposez un nouveau fichier de prototype qui tienne compte de ces éléments pour assurer un ordonnanceur utilisant l'algorithme EDF. Vous pourrez simplement indiquer ce qui change dans le fichier « `ctx.h` » donné précédemment.

Question II.2 (2 points)

Le fonction `setupCtx()` démarre l'ordonnancement préemptif en programmant une interruption qui déclenchera un appel à `yield()` chaque fois qu'une milliseconde s'est écoulée, quoi que le programme soit en train de faire. C'est pourquoi on parle d'ordonnancement « préemptif ». Avec l'algorithme EDF, l'ordonnanceur ne change pas de tâche à chaque périodiquement, comme le fait l'algorithme « *round-robin* ».
Quel sont alors, selon vous, les deux principales raisons qui justifient que l'ordonnancement reste néanmoins préemptif ?

Question II.3 (2 points)

En vous appuyant sur les déclarations que vous avez modifiées dans la question précédentes, donner une implémentation de la stratégie d'ordonnancement qui corresponde à l'algorithme EDF.

Question II.4 (2 points)

L'usage de sémaphore pose des problèmes particuliers lorsque l'ordonnancement est de type EDF. Aussi une stratégie d'« inversion de priorité » doit être mise en œuvre pour éviter ces problèmes. Proposez un programme qui utilise votre ordonnanceur pour créer trois tâches avec trois échéances différentes, et une seule sémaphore partagée, de tel sorte que le programme affiche « L'inversion de priorité est fonctionnelle. » si l'algorithme d'ordonnancement gère l'inversion de priorité, et qu'il n'affiche rien sinon.

N.B. : Détaillez le fonctionnement de votre programme.

Exercice III. Gestion asynchrone des accès au disque (8 points)

Nous nous intéressons ici à la gestion asynchrone d'un disque. Les registres d'accès au contrôleur de disque sont décrits dans l'extrait ci-dessous :
Extrait du fichier `Hardware.ini` qui décrit les ports associés au disque dur maître.

```
#
# Configuration des disques durs
#

# > Disque dur IDE Maitre
ENABLE_HDA      = 1     # ENABLE_HD=0 =>
                        #             simulation du disque désactivée
HDA_CMDREG      = 0x3F6 # registre de commande du disque maître
HDA_DATAREGS    = 0x110 # base des registres de données
                        # (r,r+1,r+2,...r+7)
HDA_IRQ         = 14    # Interruption du disque
```

L'envoi de commandes se fait également en écrivant sur le port désigné comme port de commande dans le fichier de configuration. Cela permet au microprocesseur de solliciter une opération du disque magnétique. Une fois que le microprocesseur envoie une commande, l'exécution de celle-ci débute immédiatement. Si la commande nécessite des données, il faut obligatoirement les avoir fournies (via les registres de données) avant de déclencher la commande. De la liste des commandes ATA-2 nous avons retenu le sous-ensemble décrit dans la table 1.

Nom	Code	Port de données (P0; P1; ...; P15)	objet
SEEK	0x02	numCyl (int16); numSec (int16)	déplace la tête de lecture
READ	0x04	nbSec (int16)	Lit nbSec secteurs
WRITE	0x06	nbSec (int16)	écrit nbSec secteurs
FORMAT	0x08	nbSec (int16); val (int32)	initialise nbSec secteurs avec val
STATUS	0x12	IRQFlags (int8)	Drapeau d'activation des interruptions.
DMASET	0x14	R.F.U.	R.F.U.
DSKNFO	0x16	nbCyl (int16); nbSec (int16); tailleSec (int16)	retourne la géométrie d'un disque
MANUF	0xA2	Id du fabricant du disque (16 octets)	Identifie le disque
DIAG	0xA4	Status	Diagnostic du disque : 0 = KO / 1 = OK

Table 1 : Commandes ATA-2

L'objectif de cet exercice est de proposer une gestion du disque permettant à d'autres tâches de travailler pendant que la tâche qui demande l'opération « disque » attend le résultat de sa commande. Pour cela il vous est demandé de proposer une gestion asynchrone du disque. Le principe est simple : une fois que la fonction d'accès au disque (`read_sector`, `write_sector` ou `format_sector`) a programmée le contrôleur de disque, plutôt que d'en attendre le résultat avec un `_sleep`, la fonction se bloque en utilisant un sémaphore, afin de céder le microprocesseur à une autre tâche. Lorsque l'interruption matérielle associée au disque est déclenchée (c'est-à-dire lorsque le disque a terminé l'opération) il débloque la tâche avait programmé la commande. Bien sûr, en pratique, plusieurs tâches peuvent « simultané-

ment » solliciter le disque. Aussi un autre sémaphore doit être utilisé pour assurer l'exclusion mutuelle de l'accès au contrôleur de disque.

Les procédures `int _in(int port)`, `void _out(int port, int value)`, `void _sleep(int irqlvl)`, *et* `void _mask(int irqlvl)` *fonctionnent de la même manière que dans le premier énoncé.*

Pour mémoire voici les procédures ci-dessous :

```
void cmd_seek
(int cyl, int sec) {
   /* set cylinder */
   _out(0x110,(cyl>>8)&0xFF);
   _out(0x111,cyl&0xFF);
   /* set cylinder */
   _out(0x112,(sec>>8)&0xFF);
   _out(0x113,sec&0xFF);
   /* set command */
   _out(0x3F6,0x02);
   /* wait seek */
   _sleep(14);
}
```

```
void format_sector
(int cyl, int sec, int value) {
   /*seek to the right place */
   cmd_seek(cyl,sec);
   /*setup the sector and val*/
   _out(0x110,0);
   _out(0x111,1);
   _out(0x112,(value>>24)&255);
   _out(0x113,(value>>16)&255);
   _out(0x114,(value>>8)&255);
   _out(0x115,(value)&255);
   /* set command format */
   _out(0x3F6,0x08);
   /* wait format */
   _sleep(14);
}
```

```
void write_sector
(int cyl,int sec,char *buffer){
   /* seek to the right place */
   cmd_seek(cyl,sec);
   /* fill the output buffer */
   _memcpy
   (buffer, MASTERBUFFER, 4096);
   /* set the nb of sector */
   _out(0x110,0);
   _out(0x111,1);
   /* set the command write */
   _out(0x3F6,0x04);
   /* wait writting */
   _sleep(14);
}
```

```
void read_sector
(int cyl,int sec,char *buffer){
  /* seek to the right place */
  cmd_seek(cyl,sec);
  /* set the nb of sector */
  _out(0x110,0);
  _out(0x111,1);
  /* set command read */
  _out(0x3F6,0x06);
  /* wait format */
  _sleep(14);
  /* fill the input buffer */
  _memcpy
  (MASTERBUFFER, buffer, 4096);
}
```

Ci-dessous, le prototype typique des fonctions associées aux sémaphores :

```
1.   [...]
2.   struct semaphore_s {
3.     int value ;       /*current count of the semaphore */
4.     struct ctx_s *lockList ; /*ctx suspended by the sem*/
5.   } ;
6.   /* initialisation primitive */
7.   void init_semaphore(struct semaphore_s, int count);
8.   void P(struct semaphore_s, int value); /* request a sem */
9.   void V(struct semaphore_s, int value); /* release a sem */
10.  [...]
```

Question III.1 (2 point)

Selon vous, la procédure `P()` peut-elle être appelée dans le corps d'une interruption ? et la procédure `V()` ? Expliquez vos réponses.

Question III.2 (2 point)

Expliquez comment un sémaphore peut être utilisé pour garantir que deux tâches ne tentent pas de programmer « simultanément » le contrôleur de disque.

Question III.3 (2 point)

Puis expliquez comment une autre sémaphore peut être utilisée pour suspendre la tâche en cours lorsque les `format_sector`, `write_sector` et `read_sector` ont programmées une commande disque et comment la même tâche pourra être réactivée lorsque l'interruption correspondant à l'exécution de la commande sera déclenchée.

Question III.4 (2 point)

Donnez le corps de la fonction `void init_disk() ;` qui initialise les sémaphores dont vous avez besoin, puis donnez votre implémentation (qui utilisera ces sémaphores) pour réaliser `void cmd_seek`, `format_sector`, `write_sector` et `read_sector`.