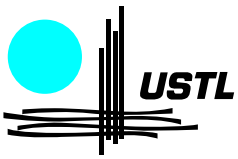

Maîtrise d'informatique/IUP GMI3
Module de SE

Examen – SE – 3 heures

Gilles GRIMAUD

Janvier 2004

Les documents de cours et TD/TP sont autorisés.
Les differents exercices proposés dans le sujet sont indépendants. On introduira tout code C par une brève description de son comportement ; on attachera un commentaire à toute portion non triviale de code.

Problème d'ordonnancement

Dans le cadre d'un petit système d'exploitation embarqué, vous êtes amené à concevoir un ordonnanceur préemptif. Ce mécanisme doit permettre à plusieurs contextes d'exécution de s'exécuter de manière concurrente. Les programmes qui s'exécutent alors dans le système seront ordonnancés de manière préemptives (sans que les applications ne sollicitent explicitement l'ordonnanceur).

Vous disposez initialement des structures de données et des procédures qui ont déjà été réalisées. Aussi, dans la suite de l'énoncé, vous pourrez considérer que les éléments déclarés par le code C ci-dessous sont disponibles. Dans ses déclarations autant que dans ses réalisations la gestion des contextes d'exécution par le microsystème d'exploitation considéré est donc comparable à celui mis en œuvre dans les TD/TPs du second sujet.

Fichier `ctx.h` :

```
#define CTX_MAGIC 0xCAD0AB0B

typedef void (t_fct) (void *); /* Type fonctions "point de depart" */

enum ctx_state_e { /* enumeration des etats d'un contexte */
  CTX_READY,       /*  . Contexte activable */
  CTX_ACTIVABLE,   /*  . Contexte re-activable */
  CTX_TERMINATED   /*  . Contexte termine */
};

struct s_ctx {
  unsigned int     magic ;    /* detrompeur */
  void            *savedESP ; /* valeur courante du registre ESP */
  void            *savedEBP ; /* valeur courante du registre EBP */
  t_fct           *startfct ; /* fonction pt de depart du contexte */
  void            *arg ;      /* argument de depart */
  char            *stack ;    /* pile d'execution */
  enum ctx_state_e state ;    /* etat courant du contexte */
  struct s_ctx    *nextCtx ;  /* point de chaînage pour "ring" */
} ;

/* anneau des contextes ordonnancables */
struct ctx_s * ring      = NULL ;
/* contexte d'exécution actuellement actif */
```

```
struct s_ctx *currentCtx = NULL ;

/* Creation */
int initCtx(struct s_ctx *aCtx, int stackSize, t_fct f, void *arg) ;
/* Implementation de la strategie d'ordonnancement */
void yield() ;
/* Implementation du mecanisme de commutation de contexte */
void switchToCtx(struct s_ctx *aCtx) ;
```

Le gestionnaire de contexte spécifié par le fichier `ctx.h` ne fournit pas un mécanisme d'ordonnancement préemptif puisque seul l'appel (explicite) à la procédure `yield()` déclanche un ordonnancement.

Exercice 1 (Gestion d'un matériel - *10 points*)
Pour pouvoir partager le temps d'accès au microprocesseur le microsystème d'exploitation a à sa disposition un circuit matériel appelé *TPU*. Ce circuit est une horloge de base (un simple registre 32 bits incrémenté périodiquement) auquel sont adjoint des horloges qui servent d'alarmes programmables. La fiche technique des registres du circuit *TPU* est donnée ci-dessous.

Fiche récapitulative des registres du circuit *TPU* :

NOM	ADRESSE	DESCRIPTION
TPU_STATUS	0xC4	bit 0 : lecture seule : 1 $\Rightarrow$ signal alarme 1 bit 1 : lecture seule : 1 $\Rightarrow$ signal alarme 2 bit 2 : lecture seule : 1 $\Rightarrow$ signal alarme 3 bit 3,4,5 : lecture & écriture : configuration alarme 1 bit 6,7,8 : lecture & écriture : configuration alarme 2 bit 9,10,11 : lecture & écriture : configuration alarme 3
TPU_BASE	0xC8	32 bits en lecture/écriture : compteur.
TPU_ALARM1	0xF4	32 bits en lecture/écriture : compteur.
TPU_ALARM2	0xF8	32 bits en lecture/écriture : compteur.
TPU_ALARM3	0xFC	32 bits en lecture/écriture : compteur.

Détails des bits de configuration des alarmes :

BIT	DESCRIPTION
bit 1	0 $\Rightarrow$ alarme OFF 1 $\Rightarrow$ alarme ON
bit 2,3	demultiplieur de fréquence. 0,0 $\Rightarrow$ fréquence alarme $\Leftrightarrow$ fréquence horloge 0,1 $\Rightarrow$ fréquence alarme $\Leftrightarrow$ 1/8 fréquence horloge 1,0 $\Rightarrow$ fréquence alarme $\Leftrightarrow$ 1/64 fréquence horloge 1,1 $\Rightarrow$ fréquence alarme $\Leftrightarrow$ 1/512 fréquence horloge

L'horloge est consultable à chaque instant par l'intermédiaire du registre `TPU_BASE`. Ce registre contient un mot de 32 bits. Le signal d'horloge et de 4,77 MHz. Cela signifie que le registre est incrémenté 4 770 000 fois chaque secondes, soit a peu prés une fois toutes les 0,21 microsecondes. La programmation d'une à trois interruptions générées après un certain délai est possible grâce aux registres `TPU_ALRM1`, `TPU_ALRM2` et `TPU_ALRM3` ainsi qu'avec l'aide du registre `TPU_STATUS`. Lorsque l'un (ou plusieurs) des registres `TPU_ALRM1`, `TPU_ALRM2` ou `TPU_ALRM3`, eux aussi incrémentés automatiquement, atteint la valeur `0xFFFFFFFF` une interruption est déclanché sur le microprocesseur. Pour programmer une interruption il suffit donc d'initialiser l'un des trois registres avec une valeur choisie. Lorsqu'il atteint `0xFFFFFFFF` l'interruption est déclanchée. La fréquence d'incrémentation des registres d'alarme est programmable via le registre de `TPU_STATUS`.

Ainsi, conformément à la Fiche récapitulative des registres, l'alarme n°1 ne sera incrémentée que si le bit 3 est à 1. Dans ce cas, elle sera incrémentée toute les 0,21 microsecondes si les bits 4 et 5 sont à 0 et 0, toutes les $8 \times 0,21$ microsecondes si les bits 4 et 5 sont à 0 et 1, toutes les $64 \times 0,21$ microsecondes si les

bits 4 et 5 sont à 1 et 0 ou encore toutes les $512 \times 0,21$ microsecondes si les bits 4 et 5 sont positionnés à 1 et 1. Il en va de même pour l'alarme 2 avec les bits 6, 7 et 8 ou encore avec l'alarme 3 et les bits 9, 10 et 11.

Question 1.1 Quels registres du *TPU* doivent être écrit pour programmer l'alarme numero 3 ? Donnez les valeurs (en hexadécimal) à placer dans les registres du *TPU* pour que son alarme 3 génère une interruption une milliseconde après initialisation des registres.

Pour pouvoir écrire la valeur 32 bits `value` dans le registre matériel numéroté `addr`, vous disposez de la procédure C suivante :

```
void _out(int addr, unsigned int value);.
```

Ainsi pour pouvoir écrire 0 dans le compteur de base (`TPU_BASE`) du *TPU* il suffit de faire :

```
_out(0xC8,0);
```

Pour pouvoir lire :

```
unsigned int _in(int addr);
```

Aussi pour pouvoir lire le registre de statut (`TPU_STATUS`) du *TPU* il suffit de faire :

```
unsigned int status = _in(0xC4);
```

Question 1.2 Proposer une implémentation de la procédure C `void setTPUALRM3(int ms)` qui programme la troisième alarme du *TPU* afin qu'il génère une interruption après ms millisecondes.

Question 1.3 La déclaration de la fonction `void setTPUALRM3(int ms)` prend un int en paramètre de fonction. Donc ms peut être une valeur entre `-2147483648` et `+2147483647` (selon le compilateur C utilisé). Dans la pratique néanmoins votre procédure ne rend pas le service attendu pour n'importe quelle valeur de l'argument. Pour quel intervalle de valeur de l'argument `ms` l'implantation de votre fonction `setTPUALARM3` fonctionne correctement ?

Question 1.4 La procédure C `void ITTPU()` est appelée lorsque le *TPU* génère une interruption. Néanmoins, comme indiquée dans la documentation, cette interruption peut survenir soit à cause de l'alarme 1, soit à cause de l'alarme 2 soit à cause de l'alarme 3. De plus, lorsque deux alarmes surviennent simultanément, le *TPU* ne déclanche qu'un seul appel à `void ITTPU()`.

En utilisant le registre `TPU_STATUS`, proposez une implantation de `void ITTPU()` qui appelle les fonctions `void ITALRM1()`, `void ITALRM2()` et `void ITALRM3()` lorsque l'interruption est déclanché par rapport à l'alarme 1, 2 ou 3.

Question 1.5 Nous supposons maintenant que la procédure C `void ITALRM3()` est appelée lorsque l'alarme 3 est déclanchée. Utilisez le gestionnaire de contexte, tel qu'il est spécifier dans `ctx.h` et la fonction `void setTPUALRM3(int ms)` pour réaliser un ordonnancement préemptif des tâches (avec une granularité de 20ms entre chaque préemption).

Exercice 2 (-Abstraire- Enrichissement de l'ordonnanceur *10 points*)

On souhaite maintenant enrichir le mécanisme d'ordonnancement des processus avec un mécanisme de signal. Le mécanisme de signal que vous avez à mettre en œvre fonctionne comme suit :

Un contexte d'exécution (par exemple `ctx2`) peut (eventuellement) associer une fonction `void sigfct(void *sigArg);` à la réception d'un signal en appelant la procédure `void setSignalHandler(t_fct sigfct);`.

Un contexte d'exécution (par exemple `ctx1` ou même `ctx2`) peut demander la transmission d'un signal à un autre contexte d'exécution. Il appelle alors la procédure `void sendSignal(ctx2, sigParam)`.

Comme le contexte `ctx2` a demandé à capturer le signal lors du prochain ordonnancement de `ctx2`, la fonction sigfct est appelée avec le paramètre `sigParam`. Une fois cette procédure exécutée le contexte est « normalement » réactivé. Si le contexte `ctx2` n'avait enregistré aucune fonction d'interception du signal, lors de son ordonnancement le contexte ctx2 n'aurait appelé aucune fonction et le signal aurait été « perdu ».

Pour mettre en œuvre cette nouvelle fonctionnalité il vous est demandé de proposer une extension du module de gestion des contextes.

Question 2.1 Les contextes d'exécution doivent connaître un nouvel état : `CATCH_SIGNAL`. Cet état est positionné lorsqu'on demande le déclanchement d'un signal sur le contexte. De plus la structure de donnée `s_ctx` doit être modifiée afin qu'elle puisse contenir l'adresse de la fonction à exécuter (l'adresse de `sigfct` dans notre exemple) ainsi que l'argument passé en paramètre au signal (`sigParam` dans notre exemple). Proposer une modification du fichier `ctx.h` pour integrer ces nouvelles informations.

Question 2.2 Donnez une implantation de la procédure `void setSignalHandler(t_fct f);` qui attache l'exécution de la fonction `f` au contexte courant. Cette procédure « écrase » éventuellement l'appel à une fonction précédement enregistrée.

Question 2.3 Donnez une implantation de la procédure `void sendSignal(s_ctx *aCtx, void *arg);` Cette procédure programme le déclanchement d'un signal exécuter avec l'argument `arg`, lors du prochain ordonnancement du contexte `aCtx`.

Question 2.4 Expliquez en quelques mots, pourquoi il n'est pas possible d'exécuter la fonction de capture du signal après avoir restauré les registres ESP et EBP de ce contexte.

Question 2.5 Donnez une nouvelle implantation de `void switchTo(s_ctx *aCtx);` afin qu'elle exécute, le cas échant, un éventuel signal associé au contexte `aCtx`.

Exercice 3 (Sécuriser Isolation des debordements de la pile d'appel *10 points*)
Nous souhaitons maintenant détecter le débordement des piles d'exécution. Lorsqu'une tâche utilise des algorithmes récursifs par exemple, il arrive que la pile d'exécution qui lui est associée soit trop petite. Le programme continu alors à s'exécuter hors de la mémoire associée à sa pile, ce qui engendre des défaillances complètes du système. Pour éviter cela, il vous est demandé d'exploiter une MMU dédiée qui permet de détecter des accès jusqu'à 16 zones mémoires.

Une zone mémoire MMU est définit par une adresse de base, une longueur et une paire « droit d'accès en lecture » / « droit d'accès en écriture ». Le circuit MMU « espionne » les accès mémoires effectués par le microprocesseur. Lorsqu'une adresse mémoire accédée appartient à une zone mémoire définie, la MMU vérifie que la mémoire est consultée conformément aux droits d'accès qui sont associés à cette zone. Si un accès interdit est demandé par le microprocesseur la MMU génère une interruption.

Fiche récapitulative des registres du circuit MMU :

NOM	ADRESSE	DESCRIPTION
`MMU_STATUS`	0x1F8	32 bits accèssible en lecture & écriture bit 0 : 0 ⇒ lecture autorisée pour la zone 0 bit 1 : 0 ⇒ écriture autorisée pour la zone 0 bit 2 : 0 ⇒ lecture autorisée pour la zone 1 bit 3 : 0 ⇒ écriture autorisée pour la zone 1 ... bit 30 : 0 ⇒ lecture autorisée pour la zone 15 bit 31 : 0 ⇒ écriture autorisée pour la zone 15
`MMU_AREA0ADDR` `MMU_AREA0LN`	0x200 0x204	32 bits en lecture/écriture : adresse de base zone 0. 32 bits en lecture/écriture : longueur zone 0.
`MMU_AREA1ADDR` `MMU_AREA1LN`	0x208 0x20C	32 bits en lecture/écriture : adresse de base zone 0. 32 bits en lecture/écriture : longueur zone 0.
...	...	...
`MMU_AREAFADDR` `MMU_AREAFLN`	0x278 0x27C	32 bits en lecture/écriture : adresse de base zone 15. 32 bits en lecture/écriture : longueur zone 15.

Une zone de longueur 0 est une zone que la MMU ne considère pas.

Question III.1 Quels registres faut-il modifier et quelle valeur faut-il y placer pour que la zone de mémoire 1 de la MMU débute à l'adresse 0xBFF000, qu'elle soit composée de 64 octets contiguë et qu'elle soit interdite d'accès en écriture.

Question III.2 Pour ce prémunir des débordements de pile des contextes d'exécution le système d'exploitation ce propose d'utiliser la MMU présentée ci-dessus. Pour cela la procédure initCtx a été modifiée. Si l'initialisation du contexte est correcte la procédure int stackProtect(struct s_ctx *aCtx) est appelée. Cette procédure programme la MMU afin d'interdit l'accès, aussi bien en lecture qu'en écriture dans les 4 octets formant le sommet de la pile (dont l'adresse est indiqué par le membre stack de la struct s_ctx) alloué.

N.B. Conformément aux exemples donnés en cours, dans l'exemple que nous considérons le sommet de la pile est l'adresse la plus petite. Il est donc égal à l'adresse donnée par le membre stack.

Donnez le corps de la procédure int stackProtect(struct s_ctx *aCtx) qui retourne 1 si la MMU à pu être programmer normalement ou 0 si toutes les zones de la MMU sont déjà utilisées...

Question III.3 Lorsqu'une zone de mémoire est accédée alors qu'elle est interdite d'accès, une interruption est déclanchée. Cette interruption exécute la procédure void ITMMU() ; Le traitement le plus simple, lorsqu'un contexte d'exécution déborde consiste à arrêter l'exécution de la tâche courante et de demander à la politique d'ordonnancement (fonction yield() de ctx.h) d'élire un nouveau contexte d'exécution. Proposer une implantation de ITMMU qui réalise cette opération. N.B. Nous considérons que la procédure ITMMU s'exécute dans une pile d'exécution " système " dissociée de la pile d'exécution du contexte qui a déclanché l'erreur.

On veut maintenant permettre qu'un contexte d'exécution puisse attacher l'exécution d'une fonction au débordement de sa pile d'exécution. L'idée est simplement la suivante. Un contexte d'exécution peut appeler la procédure void setStackOverflowHandler(t_fct f) ;. Si un débordement de pile à lieu, la pile est réinitialisée, la fonction f est alors exécutée dans la pile du contexte et lorsqu'elle est fini, le contexte est terminé.

Question III.4 Proposez une modification des structures de données présentées dans ctx.h. et une implantation de la fonction void setStackOverflowHandler(t_fct f) ;.

Question III.5 Proposez une nouvelle implantation de la fonction ITMMU() qui traite la gestion des " stackOverflowHandler ".

Fin du sujet