Gestionnaire d'interruptions

IMPORTANT
Avant de faire cette séance, vous devez avoir lu les documents suivants :

- Cours sur le gestionnaire d'interruption et les threads : *obligatoire*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé, mais déjà lu*
- Documentation sur le mode kernel du MIPS32 : *fortement recommandé*

1. Game over simple

Dans le tp1, vous avez réalisé un petit jeu dans lequel vous deviez deviner un nombre tiré au hasard. Ce jeu avait été mis dans `kinit` parce qu'à ce moment, il n'y avait pas encore d'application utilisateur. Nous vous proposons de mettre le jeu dans l'application user et de limiter le temps pendant lequel vous pouvez jouer. Nous allons vous guider pas-à-pas.

Récupération du code du tp3

- Ouvrez un `terminal`
- Allez dans le répertoire des TPs: `cd ~/kOO`
  ... ou bien le répertoire que vous avez choisi pour faire les TPs.
- Copiez les codes du **tp3**:
  `cp -rp /Infos/lmd/2024/licence/ue/LU3IN029-2024oct/kOO/tp3 .`
- Exécutez la commande : `cd tp3 ; tree -L 1`.
  Vous devriez obtenir ceci :

```
.
├── 1_gameover
└── Makefile
```

Allez dans le répertoire `1_gameover`

Le code de l'application est le suivant (dans `1_gameover/uapp/main.c`)

```c
#include <libc.h>
int main (void)
{
    int guess;
    int random;
    char buf[8];
    char name[16];

    fprintf(0,"Tapez votre nom : ");
    fgets(name, sizeof(name), 0);
    if (name[strlen(name)] == '\n')
        name[strlen(name)] = 0;
    srand(clock()); // start the random generator with a "random" seed.

    random = 1 + rand() % 99;
    fprintf(0,"Donnez un nombre entre 1 et 99: ");
    do {
        fgets(buf, sizeof(buf), 0);
        guess = atoi (buf);
        if (guess < random)
            fprintf(0,"%d est trop petit: ", guess);
        else if (guess > random)
            fprintf(0,"%d est trop grand: ", guess);
    } while (random != guess);

    fprintf(0,"\nGagné %s\n", name);
    return 0;
}
```

1. Pour essayer le jeu (dans le répertoire `tp3/1_gameover`) : tapez `make exec`
   comme vous pouvez le constater, vous avez le temps de jouer.
2. Dans la version précédente du gestionnaire de syscall, nous avions masqué les IRQ en écrivant `0` dans le registre `c0_status` (registre $12 du coprocesseur 0). Cela avait pour conséquence de mettre tous les bits à 0, entre autres le bit `IE`. Il faut modifier ça, parce que sinon, lorsque l'utilisateur demandera à lire le clavier avec l'appel système `fgets()`, l'IRQ venant du timer ne sera jamais prise en compte (voir le commentaire `TODO1` dans le code ci-après), ensuite au retour de la fonction qui réalise l'appel système, il faut masquer les IRQ pour ne pas avoir d'interruption pendant la restauration des registres jusqu'au `eret` qui fait sortir du kernel.

```
syscall_handler:
    addiu   $29,    $29,    -8*4        // context for $31 + EPC + SR + syscall_code + 4 args
    mfc0    $27,    $14                 // $27 <- EPC (addr of syscall instruction)
    mfc0    $26,    $12                 // $26 <- SR (status register)
    addiu   $27,    $27,    4           // $27 <- EPC+4 (return address)
    sw      $31,    7*4($29)            // save $31 because it will be erased
    sw      $27,    6*4($29)            // save EPC+4 (return address of syscall)
    sw      $26,    5*4($29)            // save SR (status register)
    sw      $2,     4*4($29)            // save syscall code (useful for debug message)
// TODO1: remplacez "mtc0 $0, $12" par 2 autres pour mettre 1 dans les bits c0_sr.HWI0 et c0_sr.IE
// vous pouvez utiliser $26
    mtc0    $0,     $12                 // SR <- kernel-mode without INT (UM=0 ERL=0 EXL=0 IE=0)

    la      $26,    syscall_vector      // $26 <- table of syscall functions
    andi    $2,     $2,     SYSCALL_NR-1// apply syscall mask
```

```
    sll     $2,     $2,     2           // compute syscall index (mutiply by 4)
    addu    $2,     $26,    $2          // $2 <- & syscall_vector[$2]
    lw      $2,     ($2)                // at the end: $2 <- syscall_vector[$2]
    jalr    $2                          // call syscall function

// TODO2: Il faut mettre 0 dans SR pour masquer les interruptions
    lw      $26,    5*4($29)            // get old SR
    lw      $27,    6*4($29)            // get return address of syscall
    lw      $31,    7*4($29)            // restore $31 (return address of syscall function)
    mtc0    $26,    $12                 // restore SR
    mtc0    $27,    $14                 // restore EPC
    addiu   $29,    $29,    8*4         // restore stack pointer
    eret                                // return : jr EPC with EXL <- 0
```

```
// TODO1: remplacez "mtc0 $0, $12" par 2 autres pour mettre 1 dans les bits c0_sr.HWI0 et c0_sr.IE
    li      $26,    0x401               // next value of SR
    mtc0    $26,    $12                 // SR <- kernel-mode with INT (HWI0=1 UM=0 ERL=0 EXL=0 IE=1)

// TODO2: Il faut mettre 0 dans SR pour masquer les interruptions
    mtc0    $0,     $12                 // SR <- kernel-mode without INT (UM=0 ERL=0 EXL=0 IE=0)
```

3. Ouvrez le fichier `kernel/kinit.c`. Dans cette fonction, on appelle `archi_init()` avec en paramètre un nombre qui va servir de période d'horloge. Le simulateur de la plateforme sur les machines de la PPTI va environ à 3.5MHz. Combien de secondes demande-t-on dans ce code ?

   arch_init (30*3500000); *about 30 secondes with this simulator (3.5MHz)*

4. Ouvrez le fichier `kernel/harch.c` et vous allez devoir remplir 3 fonctions pour configurer le timer: `arch_init()`, `timer_init()` et `timer_isr()` (pour trouver ces fonctions cherchez le mot `TODO`)

```
void arch_init (int tick)
{
// TODO A remplir avec 4 lignes :
// 1) appel de la fonction timer_init pour le timer 0 avec tick comme période
// 2) mise à 1 du bit 0 du registre ICU_MASK en utilisant la fonction icu_set_mask()
// 3) initialisation de la table irq_vector_isr[] vecteur d'interruption avec timer_isr()
// 4) initialisation de la table irq_vector_dev[] vecteur d'interruption avec 0
}
static void timer_init (int timer, int tick)
{
// TODO A remplir avec 2 lignes :
// 1) initialiser le registre period du timer n°timer avec la période tick (reçus en argument)
// 2) initialiser le registre mode   du timer n°timer avec 3 (démarre le timer avec IRQ demandée) si la
période est non nulle
}
static void timer_isr (int timer)
{
// TODO A remplir avec 3 lignes :
// 1) Acquiter l'interruption du timer en écrivant n'importe quoi dans le registre resetirq
// 2) afficher un message "Game Over" avec kprintf()
// 3) appeler la fonction kernel exit() (c'est une sortie définitive ici)
}
```

```
void arch_init (int tick)
{
    timer_init (0, tick);           // sets period of timer n'0 (thus for CPU n'0) and starts it
    icu_set_mask (0, 0);            // [CPU n'0].IRQ <-- ICU.PIN[0] <- Interrupt signal timer n'0
    irq_vector_isr [0] = timer_isr; // tell the kernel which isr to exec for ICU.PIN n'0
    irq_vector_dev [0] = 0;         // device instance attached to ICU.PIN n'0
}
static void timer_init (int timer, int tick)
{
    __timer_regs_map[timer].period =  tick;     // next period
    __timer_regs_map[timer].mode = (tick)?3:0;  // timer ON with IRQ only if (tick != 0)
}
static void timer_isr (int timer)
{
    __timer_regs_map[timer].resetirq = 1;       // IRQ acknoledgement to lower the interrupt signal
    kprintf ("\nGame Over\n");
    exit(1);
}
```

2. Game over avec décompteur

Dans ce qui précède, l'exécution de l'ISR du Timer est fatale puisqu'elle stoppe l'application après l'affichage de "Game Over!". Nous vous proposons de modifier l'ISR afin d'avoir un comportement un peu plus réaliste. Dans cette nouvelle version, l'ISR du timer décrémente un compteur alloué dans une variable globale du noyau, puis elle revient dans l'application tant que ce compteur est différent de 0. Donc, dans l'ISR du timer si le compteur est différent de 0, elle affiche un message avec la valeur du compteur, sinon elle affiche "game over!" et stoppe l'application, comme dans l'exercice précédent.

Par exemple, au lieu d'afficher:

```
  _     ___    __
 | |__ /'v'\  / /
 | / /(     )/ _ \
 |_\_\ x___x \___/

Tapez votre nom : Moi
Donnez un nombre entre 1 et 99: 45
45 est trop grand: 20
20 est trop grand:
0 est trop petit:
```

```
Game Over
[105002991] EXIT status = 1
```

l'application pourrait afficher:

```
  _     ___    __
 | |__ /'v'\  / /
 | / /(     )/ _ \
 |_\_\ x___x \___/

Tapez votre nom : Moi
Donnez un nombre entre 1 et 99: 45
45 est trop grand: 20
20 est trop grand:
..3 : 12
12 est trop petit: 15
15 est trop petit:
..2 :
..1 :
Game Over
[115002778] EXIT status = 1
```

kernel/harch.c

```
extern void arch_init (int tick, unsigned quantum);
```

kernel/harch.c

```
static unsigned timer_quantum;
static void timer_init (int timer, int tick, unsigned quantum)
{
    __timer_regs_map[timer].resetirq = 0;       // to delete previous untraited IRQ
    __timer_regs_map[timer].period =  tick;     // next period
    __timer_regs_map[timer].mode = (tick)?3:0;  // timer ON with IRQ only if (tick != 0)
    timer_quantum = quantum  % 100;             // %100 to avoid aberrant value
}
static void timer_isr (int timer)
{
    __timer_regs_map[timer].resetirq = 1;
    if (timer_quantum) {
        kprintf ("\n...%d : ", timer_quantum--);
    } else {
        kprintf ("\nGame Over\n");
        exit(1);
    }
}
```

kernel/kinit.c

```
void kinit (void)
{
    [...]
    arch_init (3*3500000, 10); // tick is about 3 seconde, quantum is then about 30 secondes
    [...]
}
```

3. Évaluation de la durée d'une ISR

Dans cet usage du TIMER, les ISR ne sont pas fatales, sauf la dernière. En utilisant le mode debug (make debug) et le fichier `trace0.S`, déterminez la durée en cycles du traitement par le noyau d'une IRQ du timer. Ce n'est pas exactement la même durée pour toutes les IRQ.

Pour cette question, il faut commenter les affichages dans l'ISR, changer la valeur du tick pour voir plus d'IRQ (par exemple 1000) et exécuter en mode débug puis regarder la trace dans `trace0.S`, il faut chercher un `kentry` correspondant à une IRQ et chercher l'instruction `eret` qui marque la fin du traitement.

- La durée mesurée est de l'ordre de 430 cycles pour la première (vers le cycle 3600).