

Programmation Système Avancée

TP2 – Gestion des interruptions, du clavier et du timer

Objectif

Le but de ce TP est de continuer le développement de votre kernel en implémentant les mécanismes nécessaires à la gestion des périphériques grâce au support des interruptions.

A la fin de ce TP, votre kernel sera capable de :

- Gérer les interruptions matérielles en provenance des périphériques.
- Gérer les exceptions processeur afin de détecter des événements indésirables (division par zéro, erreur de protection, etc.).
- Gérer le clavier afin de pouvoir lire les touches pressées et ainsi interagir avec l'utilisateur.
- Gérer le timer matériel afin d'implémenter une base de temps indépendante de la vitesse du processeur.

Partie 1 : mise en place du gestionnaire d'interruptions

Téléchargez l'archive `tp2.tar.gz` sur la page [Cyberlearn](#) du TP et explorez le contenu des différents fichiers fournis.

La première étape du TP est de créer la table des descripteurs d'interruption (IDT) puis de la charger. Ceci permettra au kernel de gérer correctement la réception d'interruptions matérielles et d'exceptions processeur.

Les fichiers `idt.c` et `idt.h` vous sont fournis. Il s'agit de fichiers squelettes, mais ceux-ci contiennent déjà le code pour les fonctionnalités suivantes :

- Fonction `idt_build_entry` : permet de créer une entrée pour l'IDT.
- Structure `idt_ptr_t` : représente un pointeur sur l'IDT ; nécessaire au processeur pour charger l'IDT.
- Structure `idt_entry_t` : structure décrivant un descripteur d'interruption (i.e. une entrée dans la table IDT).
- Fonction `idt_init` : fonction initialisant l'IDT, c'est-à-dire mettant en place les gestionnaires d'interruption pour les exceptions processeur ainsi que les IRQ. Cette fonction devra aussi s'occuper de charger l'IDT.
- Fonction `exception_handler` : squelette de la fonction servant de handler aux exceptions processeur. Vous remarquerez que celle-ci possède un argument : il s'agit du contexte du CPU.

Dans la fonction `idt_init`, vous implémenterez les points ci-dessous :

- Créez une IDT avec 256 entrées ; celle-ci sera remplie initialement avec des zéros.
- Insérez, aux entrées 0 à 20, les descripteurs d'interruption pour les 21 exceptions processeur (cf. cours pour la liste des exceptions). A noter que celles-ci doivent être de type `TYPE_INTERRUPT_GATE` (cf. fichier `x86.h`) et que le code de leur ISR (Interrupt

Service Routine) est localisé dans le segment du kernel (cherchez le mot clé `SELECTOR` dans le fichier `x86.h`).

- Insérez, aux entrées 32 à 47, les descripteurs d'interruption pour les 16 requêtes d'interruptions matérielles (IRQ). Celles-ci doivent aussi être de type `TYPE_INTERRUPT_GATE`.
- Enfin, l'IDT devra être chargée par le CPU. Les slides de théorie expliquent comment faire. Quelques lignes de code assembleur vous seront nécessaires.

Dans le fichier `idt.c`, il est aussi indispensable d'implémenter le gestionnaire (handler) d'interruption pour les exceptions et le gestionnaire d'interruption pour les IRQ. Chacun de ces gestionnaires nécessite un argument de type `regs_t *regs`, qui est le contexte processeur au moment de l'interruption. A noter que le champs `number` de la structure `regs` indique le numéro d'exception/IRQ déposé sur la pile par la routine d'interruption implémentée en assembleur.

Au cas où une exception processeur est déclenchée, on désire que le kernel affiche en rouge le nom de l'exception, puis stoppe le kernel (fonction `halt` du fichier `x86.h`).

Pour le gestionnaire s'occupant des IRQ, on désire gérer deux interruptions matérielles : le timer (IRQ0) et le clavier (IRQ1). Le comportement à implémenter dans le cas de ces deux IRQ est expliqué dans les sections suivantes. Pour l'instant, vous pouvez simplement afficher un message de test. A noter qu'au début ou à la fin de votre gestionnaire d'interruption pour les IRQ, vous devrez impérativement envoyer un message End Of Interrupt (EOI) au contrôleur d'interruption avec la fonction `pic_eoi` (cf. `pic.c`), sinon l'interruption ne sera plus jamais déclenchée.

La partie bas niveau est implémentée en assembleur dans le fichier `idt_asm.s` qui vous est également fourni. Dans ce fichier, vous pouvez trouver :

- `_exception_nocode` : un exemple de routine d'interruption (ISR) sur lequel une entrée de l'IDT peut pointer – dans le cas d'une exception où le CPU ne dépose pas de code d'erreur sur la pile.
- `_exception_code` : un exemple de routine d'interruption (ISR) sur lequel une entrée de l'IDT peut pointer – dans le cas d'une exception où un code d'erreur est automatiquement déposé sur la pile par le CPU.
- `_irq` : un exemple de routine d'interruption (ISR) sur lequel une entrée de l'IDT peut pointer – dans le cas d'une requête d'interruption matérielle (IRQ).
- Fonction `exception_wrapper` : cette fonction implémente les étapes suivantes (dans l'ordre) :
 1. Sauvegarde le contexte du CPU.
 2. Appelle la partie C du handler d'interruption via l'appel à la fonction `exception_handler`.
 3. Restaure le contexte du CPU.
 4. Revient au code interrompu grâce à l'instruction assembleur `iret`.

Les routines d'interruption `_exception_nocode` et `_exception_code` ci-dessus font appel à la fonction `exception_wrapper`. Celle-ci est simplement un wrapper dans le cas d'une exception processeur. Dans le cas d'une requête d'interruption (IRQ), le wrapper `irq_wrapper` est appelé et c'est à vous de l'implémenter. Celui-ci devrait être quasiment identique au wrapper pour les exceptions, à la différence qu'il devra appeler un handler C différent (celui gérant les IRQ).

A noter que pour les handlers d'IRQ (en assembleur), vous empilerez le numéro d'IRQ générée et non l'interruption CPU correspondante générée.

Bonus

Le code assembleur peut s'avérer relativement lourd, étant donné la répétition de code très semblable dans de nombreuses fonctions.

L'assembleur `nasm` possède un mécanisme de macro avec paramètres assez puissant et facile à utiliser. Si le temps le permet, essayez de factoriser au maximum votre code en utilisant les possibilités offertes par les macros de `nasm`.

Partie 2 : gestion du clavier

L'implémentation des fonctions liées au clavier est réalisée dans les fichiers sources : `keyboard.c` et `keyboard.h`. Quatre fonctions sont à implémenter.

La première fonction, `keyboard_init()`, s'occupe de l'initialisation du clavier. Cette fonction peut potentiellement être vide si vous n'avez rien à initialiser.

La deuxième fonction, `keyboard_handler()`, est le gestionnaire (*handler*) d'interruption du clavier, c'est-à-dire la fonction appelée au moment où l'IRQ1 est déclenchée. Chaque touche du clavier pressée ou relâchée génère une requête d'interruption sur l'IRQ1. Relisez les slides du cours sur la partie du clavier afin de comprendre comment déterminer quelles touches ont été pressées ou relâchées. Notez que ce handler de clavier est appelé depuis le code gérant les IRQ dans `idt.c`. Etant donné qu'une routine d'interruption (ISR) doit être la plus courte possible, nous ne voulons pas bloquer dans le code jusqu'à ce qu'une touche soit pressée. De fait, votre handler de clavier doit respecter les consignes suivantes :

- Le code ASCII (Suisse, c'est-à-dire correspondant aux claviers de l'école) de chaque touche pressée doit être stocké dans un buffer interne.
- Vous devez être capable de stocker les caractères standards (alpha-numérique) ainsi que les caractères accentués (et autres) lorsque les touches *shift* (majuscules) sont pressées (majuscules et caractères comme "!?+&%", etc.). A noter que les deux touches shift doivent être correctement gérées.
- Si le buffer interne est plein, alors aucun caractère supplémentaire ne peut être stocké dans le buffer et un message indiquant que le buffer est plein doit être affiché de manière visible à l'écran (en couleur par exemple). Des caractères pourront à nouveau être stockés dans le buffer qu'une fois les caractères du buffer consommés (lus).
- Le handler de clavier doit :
 - seulement stocker les touches pressées.
 - être le plus court et rapide possible ;
 - ne pas être bloquant ;
 - ne réaliser aucun affichage (sauf pour le buffer plein qui est une exception pour le débogage) ;

La troisième fonction à implémenter est une fonction permettant de lire les caractères du buffer du clavier. Cette fonction possède le prototype suivant :

```
int getc();
```

Cette fonction est bloquante, c'est-à-dire que si aucun caractère n'est présent dans le buffer, celle-ci bloque jusqu'au prochain caractère tapé au clavier. Le caractère retourné par `getc` est le caractère ASCII tapé par l'utilisateur. La translation en caractères ASCII doit se faire selon le mapping du clavier Suisse, donc avec accents, symboles, etc. correspondant aux claviers des machines de l'école. Il n'est pas nécessaire de gérer les touches de fonctions (F1-F12).

Enfin, la dernière fonction à implémenter est une fonction permettant de déterminer, de manière non bloquante, si une touche du clavier a été pressée. Elle possède le prototype suivant :

```
int keypressed();
```

Cette fonction sera très utile par la suite, notamment lors de l'implémentation de jeux vidéos !

Partie 3 : gestion du timer

L'implémentation des fonctions liées au timer est réalisée dans les fichiers sources : `timer.c` et `timer.h`. Quatre fonctions sont à implémenter.

La première fonction est l'initialisation du timer à la fréquence souhaitée par l'utilisateur. Cette fonction doit aussi initialiser le nombre de ticks du système à zéro. Le prototype à respecter est :

```
void timer_init(uint32_t freq_hz);
```

La deuxième fonction à implémenter est le gestionnaire (*handler*) d'interruption du timer, c'est-à-dire la fonction appelée au moment où l'IRQ0 est déclenchée. Si la fonction d'initialisation du timer est implémentée correctement, le gestionnaire doit être appelé exactement à la fréquence spécifiée dans l'appel à `timer_init`. Notez que ce handler de timer est appelé depuis le code gérant les IRQ dans `idt.c`. A titre démonstratif, on désire que le handler de timer affiche, à la fréquence du timer, quelque-chose de visuel à l'écran. Il est important que l'affichage change au cours du temps afin qu'il soit visuellement aisé de déterminer que la fréquence du timer est correcte.

La troisième fonction à implémenter doit retourner le nombre de ticks écoulés depuis le démarrage du système. Cette fonction doit respecter le prototype ci-dessous :

```
uint get_ticks();
```

Enfin, la dernière fonction à implémenter permettra d'attendre un certain nombre de millisecondes en utilisant une attente active. La fonction doit respecter le prototype suivant :

```
void sleep(uint ms);
```

Vérifiez bien sûre que votre fonction `sleep` se comporte correctement quelle que soit la fréquence de timer choisie !

Attention à ne pas écrire de code utilisant des nombres flottants (même si cela compile – ne vous y fiez pas) ! En effet, pour utiliser des nombres flottants, il est nécessaire de programmer le coprocesseur mathématique – principalement en assembleur – ce qui sort du cadre de ce cours.

Partie 4 : modification du kernel

Le code réalisant l'initialisation de votre kernel doit être modifié. En effet, le kernel doit maintenant s'occuper de l'initialisation des points suivants :

- Initialisation du clavier.
- Initialisation de l'IDT.
- Initialisation de la GDT.
- Initialisation de l'affichage (l'affichage implémenté lors du TP1).
- Initialisation du timer (PIT) ; la fréquence typique du timer dans les systèmes d'exploitation est de l'ordre de 50 à 100 Hz.
- Initialisation du contrôleur d'interruptions (PIC).

A vous de déterminer quel est l'ordre d'initialisation approprié. Indiquez à l'utilisateur le déroulement des phases d'initialisation présentées ci-dessus à l'aide de messages affichés à l'écran.

N'oubliez pas que le fichier `pic.c` vous est fourni et qu'il contient, la fonction `pic_init()` qui permet d'initialiser les contrôleurs PIC et surtout de remapper les IRQ comme expliqué en cours.

Une fois l'initialisation de ces différents points effectuée, vous activerez (démasquerez) les interruptions matérielles avec la fonction `sti` se trouvant dans le fichier `x86.h`. Exécutez ensuite votre fonction `sleep` afin de vérifier qu'elle fonctionne correctement.

Votre kernel s'occupera ensuite de :

- Lire tout caractère tapé au clavier, puis de l'afficher.
- En cas de frappe du caractère `Q` (majuscule), le kernel affichera un message puis stoppera le système. Le kernel utilisera bien sûr la fonction `getc` pour la gestion du caractère `Q`.

Questions

Veuillez répondre aux questions ci-dessous :

1. Dans quel ordre vous avez initialisé les différents points ci-dessus dans votre kernel ? Justifiez.
2. Pourquoi remappe-t-on les IRQ 0 à 7 aux interruptions 32 à 39 ?
 - Que se passerait-il si on ne le faisait pas ?
3. Comment pouvez-vous vérifier que votre gestionnaire d'interruption pour les exceptions fonctionne correctement ?
4. Quelle taille de buffer clavier avez-vous choisie et pourquoi ?
5. Comment pouvez-vous causer une situation de buffer plein, même pour un buffer de grande capacité ?
6. Est-ce que votre fonction `sleep` possède une limitation ? Si oui, laquelle et pourquoi ?

Conseils

Ce TP est divisé en plusieurs parties. Bien que ces parties soient différentes, elles possèdent des dépendances assez fortes. Dans un premier temps, il est fortement conseillé de lire entièrement l'énoncé afin d'avoir une vue d'ensemble du travail à réaliser.

Travail à rendre

Pour ce TP vous me rendrez sur [Cyberlearn](#) :

- L'arborescence complète de votre projet selon les formalités de rendu décrites dans le document « [Consignes pour les travaux pratiques](#) » se trouvant sur la page du cours.
- Vous n'avez pas de rapport à écrire, par contre vous rendrez vos réponses aux questions posées dans la section « Questions ».
 - Il est important que vous développiez et justifiez chaque réponse.
 - Vous rendrez vos réponses sous forme d'un document PDF ajouté à l'archive.
- Le code devra respecter les consignes décrites dans les documents « [Consignes générales pour l'écriture du code](#) » et « [Consignes pour l'écriture du code C](#) ».

En outre, vous présenterez une petite démo live de votre kernel démontrant que vous avez réalisé les objectifs du TP.