

Mon premier premier logiciel

Gilles Grimaud

Mars 2021

Cela fait maintenant quelques années que vous écrivez des logiciels, et pourtant, nombreux parmi vous sont ceux qui n'ont encore jamais écrits un “premier logiciel”, c'est-à-dire un logiciel qui s'exécute en premier, lorsque la machine démarre. C'est ce que nous allons faire maintenant.

Vous trouverez en suivant ce lien vers le depot `my-kernel` qui vous propose un logiciel minimal. Le Makefile proposé compile le programme présent dans `src/main.c` et produit l'image iso d'un disque amorçable sur architecture x86. Pour exécuter ce “premier logiciel” il vous faudra soit :

1. flasher l'image iso sur un support persistant (clef usb, disque) ; soit
2. utiliser une machine virtuelle pour démarrera sur l'image iso produite.

Nous privilégions cette seconde solution dans une phase de developpement et de test, car elle est plus confortable que la première (les cycles compilation/exécution sont plus courts et le debugage en est facilité).

pour démarrer une image iso en ligne de commande, nous vous recommandons l'utilisation de qemu.

```
qemu-system-x86_64 -boot d -m 2048 -cdrom mykernel.iso -curses
```

Cette commande est directement disponible dans le Makefile proposé avec :

```
make run
```

Pour produire une image iso a partir d'un binaire exécutable nous utilisons l'utilitaire grub. Vous pouvez voir la recette de cuisine que nous vous proposons dans le Makefile. Pour l'utiliser faites simplement :

```
make
```

Le make file compile d'abord votre fichier source en un .o, puis il link votre .o avec quelques autres éléments et produit un fichier binaire, et enfin, il utilise les utilitaires grub-mkrescue et xorriso pour produire le fichier iso. Notez en faisant de la sorte, techniquement, le premier premier programme exécuté est le bios, qui charge le logiciel grub, puis grub affiche un menu qui permet de charger et de lancer votre logiciel (comme s'il lancé un noyau de système d'exploitation).

Notez donc que pour que cela fonctionne correctement vous aurez besoin des outils suivants :

- gcc
- grub-common
- xorriso
- qemu

Ces outils sont installés sur les machines des salles de tp. Vous pouvez vous y connecter (via le VPN) avec :

```
ssh <login>@a<#salle>p<#poste>.fil.univ-lille1.fr
```

Sinon vous pouvez installer les outils sur votre distribution linux préférée. L'installation sur microsoft windows et macos X n'est pas impossible, mais est plus délicate... Quelques explications sont données dans le README.md du dépôt pour mettre en place un setup macos X.

Comme cela a été expliqué précédemment, c'est un fichier `.c` qui est compilé pour produire l'image iso. Cependant il est important de noter le langage C utilisé ici est du C "bar-metal". Au contraire d'un programme C classique, il ne s'exécute pas "au dessus" d'un système d'exploitation et vous ne disposez donc d'aucune des bibliothèques dont vous avez l'habitude. Ainsi des fonctions tel que `printf`, `malloc`, `fopen`, `fork`, `exit`... qui sont implémentées par la `glibc` et qui utilisent des services de votre système d'exploitation ne sont pas disponibles.

Pour vous aider, le fichier `main.c` que l'on vous propose réalise néanmoins deux fonctions de base. La première est d'initialiser les mécanismes fondamentaux des architectures intel. Il s'agit d'une part de la GDT, l'initialisation mise en place par le `main.c` fourni vous donne accès à toute la mémoire de la machine, sans restriction. D'autre part il s'agit de l>IDT qui gère sur les architectures intel les mécanismes d'interruption au sein du microprocesseur. La seconde fonction qu'assure le fichier `main.c` est de vous proposer une implémentation minimaliste des fonctions `putc()`, `puts()` et `puthex()`. Grâce à ces fonctions vous pouvez envoyer sur l'écran des caractères, et donc afficher des informations. L'écran est configuré pour fonctionner en mode "texte" 80 colonnes, 25 lignes. Dans ce mode video (défini dans les normes VGA par IBM), le contrôleur graphique (la carte graphique) partage un segment de sa mémoire avec le microprocesseur. Du point de vu du processeur, la mémoire du contrôleur graphique est accessible à l'adresse `0xA0000` mais en mode texte, les informations utilisées par la carte graphique pour produire l'image sont accessibles au microprocesseur à partir de l'adresse `0xB8000`. De plus le contrôleur graphique peut gérer le clignotement d'un curseur, via des registres matériels spécifiques. L'implémentation des fonctions `putc()` programme ce registre pour gérer le curseur matériel. Pour programmer les registres des périphériques matériels, les architectures intel proposent les instructions machines `in` et `out`. La base de code que l'on vous propose définit dans `include/ioport.h` une fonction C `unsigned char _inb(int port);` et une fonction C `void _outb(int port, unsigned char val);`, qui sont notamment utilisées pour piloter la position du curseur matériel.

Partie 1. Lire les touches saisies au clavier

Dans un premier temps nous allons nous intéresser au fonctionnement d'un contrôleur de clavier typique des architectures intel, le contrôleur PS2. Nous n'avons pas choisi d'utiliser dans ce sujet le contrôleur de clavier usb car la gestion des communications USB complique inutilement l'exercice proposé.

Vous pourrez trouver la description du fonctionnement du contrôleur clavier de vos machines sur le site osdev.org. On peut notamment y découvrir que le clavier est accessible via deux registres associés aux ports 0x60 et 0x64. On y lit que le premier est appelé *data port*, et qu'il peut être lu ou écrit, alors que le second est appelé *status register* quand on le lit, et *command register* quand on l'écrit.

Par ailleurs, le processeur clavier génère une interruption de niveau 1 quand une touche est pressée.

Question 1.1 : mon premier premier hello world.

Réaliser un premier programme qui affiche simplement "hello world" lorsque votre image iso démarre. Pour afficher Hello World, vous pourrez utiliser la fonction `puts()` proposée dans le code que nous fournissons avec le dépôt.

Question 1.2 : interroger le contrôleur clavier.

Réalisez un premier programme qui affiche simplement le code clavier retourné par le contrôleur clavier lorsqu'on tape une touche. Pour cela, réalisez simplement une boucle infinie qui lit le code clavier produit sur le port 0x60 avec des `_inb()` et écrit le nombre lu sur l'écran en utilisant par exemple `puthex()` et `putc()`.

Question 1.3 : produire des codes ascii.

De toute évidence, les codes produits par le contrôleur clavier ne sont pas des codes ascii. Mais surtout plusieurs codes sortent pour une seule frappe. - expliquez les différents codes que vous lisez et - Proposez une fonction `char keyboard_map(unsigned char);` qui retourne le code ascii associé à une touche clavier lorsque cela a un sens, ou zero sinon.

Partie 2. Mettre en place un mécanisme d'ordonnancement et de sémaphores

L'interruption de niveau 1 est une interruption d'horloge qui, par défaut est programmée pour se produire toutes les 18 milli-secondes. Vous pouvez associer l'exécution d'une fonction à l'occurrence de cette interruption en utilisant la fonction `idt_setup_handler()` proposée par le dépôt (et implémentée dans `src/idt.c`). Nous ne détaillons pas ici toutes les subtilités de la gestion des interruptions sur les architectures intel, mais tout peut être lu, dans `src/idt.c` et `src/idt0.s`.

Grâce à `idt_setup_handler(i,fct)` qui est équivalent à `IRQ_VECTOR[i]=fct;`

lorsque vous utilisez la librairie hardware de simulation du matériel, vous pouvez programmer une interruption, et notamment une interruption timer (qui est donc ici, associé au niveau d'interruption 0).

Par ailleurs, notez que la gestion du verrouillage et du déverrouillage des interruptions est plus compliquée, sur nos PC, que ne le laissait espérer l'interface de la librairie **hardware** utilisée en ASE. Pour bloquer le traitement des interruptions il existe, sur intel une instruction **cli** (clear interrupt) et une instruction opposée **sti** (set interrupt) qui réactive le traitement des interruptions par le microprocesseur. Ceci est suffisant pour vous permettre d'implémenter des sections critiques de code (en l'absence de multicœurs). Cependant, il vous faut noter que lorsqu'une interruption est produite par un matériel, le signal est transmit à un premier circuit qui va *geler* toute autre signal d'interruption jusqu'à ce que vous le déverrouiller. Vous pouvez voir, dans le fichier `idt.c` que la fonction `irq_handler` (ligne 146) qui appelle les fonctions `handler()` associées à une interruption, prend garde de *degeler* la reception de nouvelles interruptions avant de terminer. Elle fait cela en écrivant dans un registre de périphérique matériel associé à ce controleur d'interruption (lignes 153 à 156). Même après avoir exécuté une instruction **sti** vous ne recevrez pas d'autre signal d'interruption tant que vous n'aurez pas effectué cette opération sur le controleur d'interruption.

Question 2.1 : Un ordonnancement préemptif.

Reutilisez la fonction de changement de contexte vue en cours d'ASE pour implémenter un ordonnanceur de contexte préemptif minimaliste. La principale difficulté ici est d'adapter votre logiciel pour qu'il n'est plus recours aux fonctions `malloc` et `free`, ni aux `_mask` et `IRQ_VECTOR`. De plus le code démarré est en mode protégé 32bit et non 64, il faut donc utiliser les registres `esp,ebp` et non `rsp` et `rbp`.

Question 2.2 : Un premier démonstrateur.

A l'aide de votre mécanisme de sémaphore, réalisez un premier démonstrateur qui lance deux contextes. Le premier affiche en boucle les caractères saisi au clavier, alors que le second affiche, un compteur qui s'incrémente, en haut à droite de l'écran.

Question 2.3 : Les fonctions `sem_up()` et `sem_down()`.

Importez les fonctions de gestions des sémaphores vues en cours d'ASE dans votre système. Pensez (1) à proscrire toute utilisation du tas (pas de `malloc` ni de `free`) et (2) à utiliser les instructions intel `cli` et `sti` pour garder vos sections critiques de code plutot que les `_mask` proposés par la librairie C.

Partie 3. Implémenter `getc()` entre interruptions et sémaphores

Le programme de saisie de touche au clavier que vous avez réalisé dans la question 2.3 à l'inconvénient d'effectuer une "attente active" des saisies au clavier. D'une part, cela implique que le microprocesseur passe tout le temps que l'ordonnanceur consacre au programme de saisie à scruter la saisie d'une touche, ce qui n'est pas très utile. D'autre part, cela implique que si un trop grand nombre de touche est tapé au clavier, alors qu'un autre programme est en train de s'exécuter, le buffer du contrôleur de clavier risque de se remplir, et des frappes risquent d'être perdues.

Pour éviter cela, il est plus pertinent d'organiser l'architecture de votre petit système de telle sorte que la réception d'une touche du clavier soit gérée par le système, et non, directement par les programmes qui s'exécutent dans des contextes.

Question 3.1 : L'interruption clavier

Développez une fonction associée à l'interruption clavier (interruption de niveau 0). Cette fonction lit le code clavier, et écrit, si cela a du sens, le code ASCII associé à la touche dans une file.

Question 3.2 : Suspendre les contextes qui attendent une saisie au clavier

Développez une fonction `char getc()`. Cette fonction retourne un caractère lu au clavier. Notez que cette fonction doit être bloquante. Le contexte qui l'appelle sera suspendu (et donc ne sera plus élu par l'ordonnanceur) jusqu'à ce qu'une nouvelle touche soit pressée.

Pour réaliser cela, proposez une solution qui utilise les sémaphores implémentées précédemment. Lorsqu'une interruption clavier a lieu, elle peut "débloquer" un contexte qui attendrait ou bien éviter au prochain appel d'être bloqué, puisque un caractère est disponible.

Question 3.3 : Gérer une file d'entrées au clavier

La solution proposée pour la question 3.2 ne gère qu'un contexte appelant la fonction `getc()`. Si plusieurs contextes appellent la fonction `getc()` que peut-il se passer d'incorrect ? Proposez une solution pour traiter ce problème.