**Mini-projet système : développement d'un noyau de système d'exploitation**

Responsable : Christophe Rippert
`Christophe.Rippert@Grenoble-INP.fr`

# Gestion de l'écran en mode CGA

## Introduction

Dans ce mini-projet, on va se limiter à un mode d'affichage très simple géré par toutes les cartes graphiques depuis le début des années 80 (norme CGA, pour *Color Graphics Adapter* : la première carte vidéo gérant les couleurs sur les PC compatibles IBM).

## Spécification de l'affichage à l'écran

L'écran que nous considérons est le mode texte de base des cartes vidéo des PC dans lequel le noyau démarre. L'affichage fait 80 colonnes sur 25 lignes : les lignes et colonnes sont traditionnellement numérotées à partir de 0 (donc de 0 à 24 pour les lignes et de 0 à 79 pour les colonnes). L'affichage s'effectue en écrivant directement dans la mémoire vidéo pour y placer les caractères et leur couleur. Certaines opérations simples d'entrées-sorties sont nécessaires pour déplacer le curseur clignotant qui indique la position actuelle d'affichage.

### Principe

L'écran est couplé à une zone de la mémoire vidéo commençant à une adresse dépendant du mode utilisé (ici l'adresse de début est 0xB8000) : tout ce qui est écrit dans cette zone mémoire est donc immédiatement affiché à l'écran. Dans le mode vidéo utilisé, l'écran peut être vu comme un tableau de $80 \times 25 = 2000$ cases. Chaque case représente un caractère affiché à l'écran, et est composée de 2 octets (on utilise le type `uint8_t` pour représenter un octet en C) :

- le premier octet contient le code ASCII du caractère ;
- le deuxième octet contient le format du caractère, c'est à dire la couleur du texte, la couleur du fond et un bit indiquant si le texte doit clignoter.

Le mot de 16 bits à construire aura donc la forme suivante :

| **bit 15** | **bits 14, 13 et 12** | **bits de 11 à 8** | **bits de 7 à 0** |
|---|---|---|---|
| clignote | couleur du fond | couleur du texte | code ASCII du caractère |

Attention : le clignotement n'est pas géré correctement par l'environnement d'exécution, vous devez forcer le bit 15 à 0.
Les couleurs disponibles sont listées ci-dessous :

| **valeur** | **couleur** | **valeur** | **couleur** | **valeur** | **couleur** | **valeur** | **couleur** |
|---|---|---|---|---|---|---|---|
| 0 | noir | 4 | rouge | 8 | gris foncé | 12 | rouge clair |
| 1 | bleu | 5 | magenta | 9 | bleu clair | 13 | magenta clair |
| 2 | vert | 6 | marron | 10 | vert clair | 14 | jaune |
| 3 | cyan | 7 | gris | 11 | cyan clair | 15 | blanc |

Les 16 couleurs sont possibles pour le texte, par contre seules les 8 premières peuvent être sélectionnées pour le fond.
Pour afficher un caractère à la ligne `lig` et à la colonne `col` de l'écran, on doit donc écrire dans le mot de 2 octets (`uint16_t` en C) dont l'adresse en mémoire peut être calculé à partir de la simple formule suivante : $0xB8000 + 2 \times (lig \times 80 + col)$.

## Gestion du curseur

Lorsqu'on écrit dans un terminal en mode texte, on voit s'afficher un curseur clignotant qui indique la prochaine case dans laquelle on va écrire. Dans le mode vidéo que l'on utilise, ce curseur est géré directement par la carte vidéo : il suffit de lui indiquer à quelles coordonnées elle doit l'afficher.
On communique pour cela via des ports d'entrée-sorties : il s'agit de canaux de communication reliant les périphériques et dont les adresses sont fixées. Il existe deux types de ports :

— les ports de commandes qui servent à indiquer au périphérique l'opération souhaitée ;
— les ports de données qui permettent de communiquer effectivement avec le périphérique, en lisant ou en envoyant des données.

Dans l'architecture x86, il existe 65536 ports : le numéro d'un port est donc une valeur sur 16 bits. Cependant, on ne peut pas accéder aux ports directement via des pointeurs : on doit utiliser des instructions particulières.
Il existe des instructions assembleur dédiées pour la communication via les ports : sur l'architecture x86, il s'agit de l'instruction `in` (pour lire une donnée en provenance d'un port et la stocker dans un registre du processeur) et de l'instruction `out` (pour envoyer une donnée à un port). Ces deux instructions s'écrivent d'une façon particulière :

— `inb %dx, %al` : lit un octet de donnée sur le port dont le numéro est dans `%dx` et le stocke dans `%al` : attention, on doit obligatoirement utiliser les registres `%al` et `%dx`, à l'exclusion de tout autre ;
— `outb %al, %dx` : envoie l'octet contenu dans `%al` sur le port dont le numéro est dans `%dx`.

Bien sûr, il existe des équivalents pour lire des valeurs sur plus de 8 bits (`inw`, `ouw`, ...) mais on ne les utilisera pas dans ce projet.
Pour éviter d'avoir à écrire des bouts de fonction en assembleur, on fournit dans la mini-bibliothèque C (fichier d'en-tête `cpu.h`) des fonctions qui appellent elles-même les instructions `in` et `out` :

— `uint8_t inb(uint16_t num_port)` : renvoie l'octet lu sur le port de numéro `num_port` ;
— `void outb(uint8_t val, uint16_t port)` : envoie la valeur `val` sur le port `num_port`.

Lorsque vous écrivez des fonctions C qui doivent faire des entrée-sorties sur les ports, vous devez utiliser ces fonctions et ne surtout pas essayer d'ajouter de l'assembleur directement dans votre code C (l'assembleur *inline* est très complexe à mettre au point).
Dans les cartes vidéos VGA que l'on utilise dans ce TP, le port de commande gérant la position du curseur est le 0x3D4 et le port de données associé est le 0x3D5. La position du curseur est un entier sur 16 bits calculé via la formule suivante : $pos = col + lig \times 80$. Cette position doit être envoyée en deux temps à la carte vidéo : la succession d'opérations à effecter est donc la suivante :

1. envoyer la commande 0x0F sur le port de commande pour indiquer à la carte que l'on va envoyer la partie basse de la position du curseur ;
2. envoyer cette partie basse sur le port de données ;
3. envoyer la commande 0x0E sur le port de commande pour signaler qu'on envoie maintenant la partie haute ;
4. envoyer la partie haute de la position sur le port de données.

## Les caractères à afficher

On considère dans ce TP les caractères de la table ASCII (`man ascii`), qui sont numérotés de 0 à 127 inclus. Les caractères dont le code est supérieur à 127 (accents, ...) seront ignorés.
Les caractères de code ASCII 32 à 126 doivent être affichés en les plaçant à la position actuelle du curseur clignotant et en déplaçant ce curseur sur la position suivante : à droite, ou au début de la ligne suivante si le curseur était sur la dernière colonne.
Les caractères de 0 à 31, ainsi que le caractère 127 sont des caractères de contrôle. Le tableau ci-dessous décrit ceux devant être gérés. Tous les autres caractères de contrôle doivent être ignorés.

| Code | Mnémonique | Syntaxe C | Effet |
|---|---|---|---|
| 8 | BS | `\b` | Recule le curseur d'une colonne (si colonne $\neq 0$) |
| 9 | HT | `\t` | Avance à la prochaine tabulation (colonnes 0, 8, 16, ..., 72, 79) |
| 10 | LF | `\n` | Déplace le curseur sur la ligne suivante, colonne 0 |
| 12 | FF | `\f` | Efface l'écran et place le curseur sur la colonne 0 de la ligne 0 |
| 13 | CR | `\r` | Déplace le curseur sur la ligne actuelle, colonne 0 |

## Résumé du travail demandé

Le but de cette partie du projet est d'écrire une fonction `console_putbytes` qui affiche une chaine de caractères à la position courante du curseur. Attention, vous devez respecter le nom et la spécification de cette fonction car elle est appelée par d'autres fonctions du noyau, par exemple `printf`.
A part celles pour laquelle c'est noté explicitement, toutes les fonctions doivent être écrites en C. Pour les fonctions à écrire en assembleur, il est recommandé de procéder en deux temps :

1. écrire la fonction en C et la tester ;
2. traduire le code C systématiquement en assembleur x86.

Pour tester en C les fonctions accédant aux ports d'entrée-sortie, vous pouvez utiliser les pseudo-fonctions C : `uint8_t inb(uint16_t port)` et `void outb(uint8_t val, uint16_t port)` (qui ne font en fait qu'appeler les instructions assembleur équivalentes).
Les fonctions en assembleur doivent être écrites dans des fichiers `fct_xxxx.S` (n'essayez pas d'inclure du code assembleur directement dans du code C : écrire du code assembleur *inline* implique de respecter des contraintes complexes et complique grandement la mise au point des fonctions). Notez que les fichiers doivent avoir une extension en majuscules : la différence entre un fichier `.s` et `.S` est que GCC fait passer le pré-processeur sur les fichiers `.S` avant d'appeler l'assembleur, ce qui permet d'inclure des fichiers d'en-tête `.h` et donc d'utiliser des constantes.
Pour arriver au but final vous pouvez par exemple implanter dans cet ordre :

- une fonction `uint16_t *ptr_mem(uint32_t lig, uint32_t col)` qui renvoie un pointeur sur la case mémoire correspondant aux coordonnées fournies : **cette fonction doit être traduite en assembleur** ;
- une fonction `void ecrit_car(uint32_t lig, uint32_t col, char c)` qui écrit le caractère c aux coordonnées spécifiées (vous pouvez aussi ajouter des paramètres pour permettre de préciser la couleur du caractère, celle du fond ou le bit de clignotement) : **cette fonction doit être traduite en assembleur** ;
- une fonction `void efface_ecran(void)` qui doit parcourir les lignes et les colonnes de l'écran pour écrire dans chaque case un espace en blanc sur fond noir (afin d'initialiser les formats dans la mémoire) ;
- une fonction `void place_curseur(uint32_t lig, uint32_t col)` qui place le curseur à la position donnée : **cette fonction doit être traduite en assembleur** ;
- une fonction `void traite_car(char c)` qui traite un caractère donné (c'est à dire qui l'affiche si c'est un caractère normal ou qui implante l'effet voulu si c'est un caractère de contrôle) ;
- une fonction `void defilement(void)` qui fait remonter d'une ligne l'affichage à l'écran (il pourra être judicieux d'utiliser `memmove` définie dans `string.h` pour cela) ;
- la fonction `void console_putbytes(const char *s, int len)` demandée, qui va sûrement utiliser les fonctions précédentes.

Afin de vérifier le bon fonctionnement de vos différentes fonctions, le plus simple est de faire un affichage avec `printf` (définie dans `stdio.h`), car `printf` utilise `console_putbytes` pour l'affichage à l'écran.
**Remarque importante :** l'émulateur QEmu ne permet pas de lire correctement la position courante du curseur depuis la carte vidéo, seulement de l'écrire. Le module de gestion de l'écran gardera donc en interne la position courante du curseur (ainsi que les différents attributs : couleur du texte, du fond, clignotement) dans des variables globales et l'enverra vers la carte quand nécessaire.
Le bout de bibliothèque C fourni comprend de nombreuses fonctions utiles : il faut s'en servir pour ne pas ré-inventer (et perdre du temps à mettre au point) du code redondant ! Vous trouverez la documentation des fonctions C dans les pages man habituelles : par exemple, `man memmove` ou `man sprintf`.

## Conseils d'implantation

On rappelle quelques opérations binaires en C qui seront utiles pour ce projet :

- x décalé de n bits vers la gauche : `x << n` ;
- x décalé de n bits vers la droite : `x >> n` (attention si le type de x est signé, il s'agit d'un décalage arithmétique avec propagation du bit de signe) ;
- forcer les 3 bits de poids faibles de x à 0 : `x &= 0xF8` ;
- forcer le bit de poids fort de x à 1 : `x |= 0x80` ;

— calculer le not binaire de x : `~x` (bien lire « tilda x »).