Programmer un noyau de système d'exploitation

Philippe Marquet, Gilles Grimaud, Samuel Hym
Florian Vanhems, Clément Ballabriga, Giuseppe Lipari

11 janvier 2022

Table des matières

1 Introduction à l'environnement de programmation **2**
1.1 Un noyau minimal . . . 2
1.2 Prérequis . . . 2
1.3 Librairie minimale d'entrée/sortie . . . 3

2 Retour à un contexte **3**
2.1 La bibliothèque Unix standard . . . 3
2.2 Exercice : Illustration du mécanisme de `setjmp()=/=longjmp()` . . . 3
2.3 Exercice : Lisez la documentation . . . 4
2.4 Exercice : Utilisation d'un retour dans la pile d'exécution . . . 5

3 Contexte d'exécution **5**
3.1 Code assembleur . . . 5
3.2 Première réalisation pratique . . . 6
3.3 Exercice : try / throw . . . 6

4 Création d'un contexte d'exécution **7**
4.1 Segment mémoire . . . 7
4.2 Exercice : structure . . . 7
4.3 Exercice : initialisation . . . 7

5 Changement de contexte **7**
5.1 Coroutines . . . 7
5.2 Exercice : switch . . . 8

6 Ordonnancement **8**
6.1 La liste chaînés dans le noyau Linux . . . 9
6.2 Exercice : création d'un contexte . . . 9
6.3 Exercice : yield() . . . 9

7 Ordonnancement sur interruptions **9**
7.1 Exercice : activer / désactiver les interruptions . . . 9
7.2 Exercice : ordonnanceur round-robin . . . 10

8 Synchronisation **10**
8.1 Producteur / consommateur . . . 10
8.2 Exercice : implantation des sémaphores . . . 11

9 Un pilote de péripherique **11**
9.1 Exercice : Lire les touches saisies au clavier . . . 11
9.2 Exercice : pilote du clavier . . . 12

1 Introduction à l'environnement de programmation

Cela fait maintenant quelques années que vous écrivez des logiciels, et pourtant, nombreux parmi vous sont ceux qui n'ont encore jamais écrit 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.

1.1 Un noyau minimal

Vous trouverez en suivant ce lien vers le dépôt 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 démarrer l'image iso produite.

Nous privilégions cette seconde solution dans une phase de développement et de test, car elle est plus confortable que la première (les cycles compilation/exécution sont plus courts et le débuggage 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 à 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 Makefile compile d'abord votre fichier source en un `.o`, puis il *link*e 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 permier 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 lançait un noyau de système d'exploitation).

1.2 Prérequis

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.

1.3 Librairie minimale d'entrée/sortie

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 que le langage C utilisé ici est du C « bare-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 librairies dont vous avez l'habitude. Ainsi des fonctions telles 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 vidéo (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 vue 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 de la fonction `putc()` programme ce registre pour gérer le curseur matériel. Pour programmer les registres des périphériques matériels, les architectures intel proposent les instructions machine `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.

2 Retour à un contexte

Pour certaines applications, l'exécution du programme doit être reprise en un point particulier. Un point d'exécution est caractérisé par l'état courant de la pile d'appel et des registres du processeur ; on parle de **contexte**.

2.1 La bibliothèque Unix standard

La fonction `setjmp` de la bibliothèque standard mémorise le contexte courant dans une variable de type `jmp_buf` et retourne 0.

La fonction `longjump` permet de réactiver un contexte précédemment sauvegardé. À l'issue de cette réactivation, `setjmp` « retourne » une valeur non nulle.

```c
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

2.2 Exercice : Illustration du mécanisme de `setjmp()=/=longjmp()`

1. Il s'agit d'appréhender le comportement du programme suivant :

```c
#include <setjmp.h>
#include <stdio.h>

static int i = 0;
static jmp_buf buf;

int main()
{
    int j;

    if (setjmp(buf))
        for (j=0; j<5; j++)
            i++;
    else {
        for (j=0; j<5; j++)
            i--;
        longjmp(buf,~0);
    }
   printf("%d\n", i );
}
```

2. et de sa modification :

```c
#include <setjmp.h>
#include <stdio.h>

static int i = 0;
static jmp_buf buf;

int main()
{
    int j = 0;

    if (setjmp(buf))
        for (; j<5; j++)
            i++;
    else {
        for (; j<5; j++)
            i--;
        longjmp(buf,~0);
    }
   printf("%d\n", i );
}
```

2.3 Exercice : Lisez la documentation

Expliquez en quoi le programme suivant est erroné :

```c
#include <setjmp.h>
#include <stdio.h>

static jmp_buf buf;
static int i = 0;

static int cpt()
{
    int j = 0;

    if (setjmp(buf)) {
        for (j=0; j<5; j++)
            i++;
    } else {
        for (j=0; j<5; j++)
            i--;
    }
}

int main()
{
    int np = 0 ;

    cpt();

    if (! np++)
        longjmp(buf,~0);

    printf("i = %d\n", i );
}
```

Vous pouvez ainsi apprécier l'extrait suivant de la page de manuel de `setjump(3)` :

> setjmp() and sigsetjmp make programs hard to understand and maintain. If possible an alternative should be used.

2.4 Exercice : Utilisation d'un retour dans la pile d'exécution

Modifiez le programme suivant qui calcule le produit d'un ensemble d'entiers lus sur l'entrée standard pour retourner dans le contexte de la première invocation de `mul()` si on rencontre une valeur nulle : le produit de termes dont l'un est nul est nul.

```
static int mul(int depth)
{
    int i;

    switch (scanf("%d", &i)) {
        case EOF :
            return 1; /* neutral element */
        case 0 :
            return mul(depth+1); /* erroneous read */
        case 1 :
            if (i)
                return i * mul(depth+1);
            else
                return 0;
    }
}

int main()
{
    int product;

    printf("A list of int, please\n");
    product = mul(0);
    printf("product = %d\n", product);
}
```

3 Contexte d'exécution

Pour s'exécuter, les procédures d'un programme en langage C sont compilées en code machine. Ce code machine exploite lors de son exécution des registres et une pile d'exécution.

Dans le cas d'un programme compilé pour les microprocesseurs Intel x86, le sommet de la pile d'exécution est pointé par le registre 32 bits `esp` (*stack pointer*). Par ailleurs le microprocesseur Intel définit un registre désignant la base de la pile, le registre `ebp` (*base pointer*).

Ces deux registres définissent deux adresses, à l'intérieur de la zone réservée pour la pile d'exécution, l'espace qui les sépare est la fenêtre associée à l'exécution d'une fonction (*frame*) : `esp` pointe le sommet de cette zone et `ebp` la base.

Grossièrement, lorsque une procédure est appelée, les registres du microprocesseur (excepté `esp`) sont sauvés au sommet de la pile, puis les arguments sont empilés et enfin le pointeur de programme, avant qu'il branche au code de la fonction appelée. Notez encore que la pile Intel est organisée selon un ordre d'adresse décroissant (empiler un mot de 32 bits en sommet de pile décrémente de 4 l'adresse pointée par `esp`).

Sauvegarder les valeurs des deux registres `esp` et `ebp` suffit à mémoriser un contexte dans la pile d'exécution.

Restaurer les valeurs de ces registres permet de se retrouver dans le contexte sauvegardé. Une fois ces registres restaurés au sein d'une fonction, les accès aux variables automatiques (les variables locales allouées dans la pile d'exécution) ne sont plus possibles, ces accès étant réalisés par indirection à partir de la valeur des registres.

L'accès aux registres du processeur peut se faire par l'inclusion de code assembleur au sein du code C.

3.1 Code assembleur

Le compilateur GCC autorise l'inclusion de code assembleur au sein du code C via la construction `asm()`. De plus, les opérandes des instructions assembleur peuvent être exprimées en C.

Le code C suivant permet de copier le contenu de la variable `x` dans le registre `eax` puis de le transférer dans la variable `y` ; on précise à GCC que la valeur du registre `eax` est modifiée par le code assembleur :

```c
int main()
{
    int x = 10, y;
    asm("movl %1, %%eax" "\n\t" "movl %%eax, %0"
        : "=r"(y)  /* y is the output operand */
        : "r"(x)   /* x is the input operand */
        : "%%eax"; /* eax is a clobbered register */
}
```

Attention, cette construction est hautement non portable et n'est pas standard ISO C ; on ne peut donc utiliser l'option `-ansi` de `gcc`. Le schéma général est le suivant :

```
asm ( "code assembleur"
      : output operands       (optional)
      : input operands        (optional)
      : list of clobbered registers (optional));
```

Le code assembleur peut référencer les opérandes par leur index (à compter de 0 !) dans la liste des opérandes sous la forme `%i`.

Instruction assembleur. L'instruction x86 la plus couramment utilisée dans ce cadre est l'instruction `movl` qui copie le second opérande dans le premier.

Contraintes. Chacun des opérandes peut être associé à des contraintes. Par exemple on peut spécifier que la valeur doit être dans un registre (`r`). GCC % fera le nécessaire pour ces contraintes soient respectées. Le signe = indique que l'opérande est en sortie.

On se référera au tutoriel d'utilisation d'assembleur x86 en ligne disponible à `http://www-106.ibm.com/developerworks/linux/library/l-ia.html` ou à la documentation de GCC disponible à `http://gcc.gnu.org/onlinedocs/`.

3.2 Première réalisation pratique

Avant de développer les exercices qui suivent, dans un premier temps fournissez un moyen d'afficher la valeur des registres `esp` et `ebp` de la fonction courante.

— Observez ces valeurs sur un programme simple composé d'appels imbriqués puis successifs à des fonctions.
— Comparez ces valeurs avec les adresses des première et dernière variables automatiques locales déclarées dans ces fonctions.
— Comparez ces valeurs avec les adresses des premier et dernier paramètres de ces fonctions.
— Expliquez.

3.3 Exercice : try / throw

On définit un jeu de primitives pour retourner à un contexte préalablement mémorisé dans une valeur de type `struct ctx_s`.

```c
/* A function that returns an int from an int */
typedef int (func_t)(int);

int try(struct ctx_s *pctx, func_t *f, int arg);
```

Cette première primitive va exécuter la fonction `f()` avec le paramètre `arg`. Au besoin le programmeur pourra retourner au contexte d'appel de la fonction `f` mémorisé dans `pctx`. La valeur retournée par `try()` est la valeur retournée par la fonction `f()`.

```c
int throw(struct ctx_s *pctx, int r);
```

Cette primitive va retourner dans un contexte d'appel d'une fonction préalablement mémorisé dans le contexte `pctx` par `try()`. La valeur `r` sera alors celle « retournée » par l'invocation de la fonction au travers `try()`.

— Définissez la structure de données `struct ctx_s`.
— La fonction `try()` sauvegarde un contexte et appelle la fonction passée en paramètre. Donnez une implantation de `try()` C'est recommendé d'imprimer les valeurs des registres `esp` et `ebp` pour debug.

— Proposez une implantation de la fonction `throw()`. La fonction `throw()` restaure un contexte. On se retrouve alors dans un contexte qui était celui de l'exécution de la fonction `try()`. Cette fonction `try()` se devait de retourner une valeur. La valeur que nous allons retourner est celle passée en paramètre `throw()`.

Implantez votre bibliothèque de retour dans la pile d'exécution et testez son comportement sur une variante du programme de l'exercice 2.4.

Observez l'exécution de ce programme sous le débermineur ; en particulier positionnez un point d'arrêt dans `throw()` et continuez l'exécution en pas à pas une fois ce point d'arrêt atteint.

4 Création d'un contexte d'exécution

Un contexte d'exécution est donc principalement une pile d'exécution, celle qui est manipulée par le programme compilé via les registres `esp` et `ebp`.

Cette pile n'est, en elle-même qu'une zone mémoire dont on connaît l'adresse de base. Pour pouvoir restaurer un contexte d'exécution il faut aussi connaître la valeur des registres `esp` et `ebp` qui identifient une frame dans cette pile.

De plus lorsqu'un programme se termine, son contexte d'exécution ne doit plus pouvoir être utilisé.

Enfin, un contexte doit pouvoir être initialisé avec un pointeur de fonction et un pointeur pour les arguments de la fonction. Cette fonction sera celle qui sera appelée lors de la première activation du contexte. On suppose que le pointeur d'arguments est du type `void *`. La fonction appelée aura tout loisir pour effectuer une coercition de la structure pointée dans le type attendu.

4.1 Segment mémoire

La gestion de la pile d'exécution est associée à un segment de mémoire virtuelle pointé par le registre `ss` (*stack segment*). Les noyaux Linux utilisent le même segment pour la pile d'appel et pour le stockage des données usuelles (variables globales et tas d'allocation du C). C'est pourquoi il nous est possible d'allouer un espace mémoire et de l'utiliser pour y placer une pile d'exécution… Sur d'autres systèmes d'exploitation, dissociant ces différents segments, les programmes que nous proposons seraient incorrects.

4.2 Exercice : structure

— Déclarez un nouveau type `func_t`, utilisé par le contexte pour connaître le point d'entrée de la fonction (elle ne retourne rien et prend en paramètre le pointeur d'arguments).
— Étendez la structure de donnée `struct ctx_s` qui décrit un tel contexte.

4.3 Exercice : initialisation

Proposez une procédure

```
int init_ctx(struct ctx_s *ctx, func_t f, void *args);
```

qui initialise le contexte `ctx` avec une pile d'exécution de `STACK_SIZE` octets, avec `STACK_SIZE` une constante déclarée globalement (par exemple, égale à 4096 octets). Lors de sa première activation ce contexte appellera la fonction `f` avec le paramètre `args`.

5 Changement de contexte

5.1 Coroutines

Nous allons implanter un mécanisme de coroutines. Les coroutines sont des procédures qui s'exécutent dans des contextes séparés. Ainsi une procédure `ping` peut « rendre la main » à une procédure `pong` sans terminer, et la procédure `pong` peut faire de même avec la procédure `ping` ensuite, `ping` reprendra son exécution dans le contexte dans lequel elle était avant de passer la main. Notre ping-pong peut aussi se jouer à plus de deux …

```c
struct ctx_s ctx_ping;
struct ctx_s ctx_pong;

void f_ping(void *arg);
void f_pong(void *arg);

int main(int argc, char *argv[])
{
    init_ctx(&ctx_ping, f_ping, NULL);
    init_ctx(&ctx_pong, f_pong, NULL);
    switch_to_ctx(&ctx_ping);

    exit(EXIT_SUCCESS);
}

void f_ping(void *args)
{
    while(1) {
        putc('A') ;
        switch_to_ctx(&ctx_pong);
        putc('B') ;
        switch_to_ctx(&ctx_pong);
        putc('C') ;
        switch_to_ctx(&ctx_pong);
    }
}

void f_pong(void *args)
{
    while(1) {
        putc('1') ;
        switch_to_ctx(&ctx_ping);
        putc('2') ;
        switch_to_ctx(&ctx_ping);
    }
}
```

L'exécution de ce programme produit sans fin :

```
A1B2C1A2B1C2A1...
```

Cet exemple illustre la procédure

```c
void switch_to_ctx(struct ctx_s *ctx) ;
```

qui sauvegarde simplement les pointeurs de pile dans le contexte courant, puis définit le contexte dont l'adresse est passée en paramètre comme nouveau contexte courant, et en restaure les registres de pile. Ainsi lorsque cette procédure exécute `return;` elle « revient » dans le contexte d'exécution passé en paramètre.

Si le contexte est activé pour la première fois, au lieu de revenir avec un `return;` la fonction appelle `f(args)` pour « lancer » la première exécution... Attention, après que les registres de piles aient été initialisés sur une nouvelle pile d'exécution pour la première fois, les variables locales et les arguments de la fonction `switch_to_ctx()` sont inutilisables (ils n'ont pas été enregistrés sur la pile d'exécution).

5.2 Exercice : switch

Proposez une implantation de la procédure `switch_to_ctx()`.

6 Ordonnancement

La primitive `switch_to_ctx()` du mécanisme de coroutines impose au programmeur d'expliciter le nouveau contexte à activer. Par ailleurs, une fois l'exécution de la fonction associée à un contexte terminée, il n'est pas possible à la primitive `swicth_to_ctx()` d'activer un autre contexte ; aucun autre contexte ne lui étant connu.

Un des objectifs de l'ordonnancement est de choisir, lors d'un changement de contexte, le nouveau contexte à activer. Pour cela il est nécessaire de mémoriser l'ensemble des contextes connus ; par exemple sous forme d'une structure chaînée circulaire des contextes.

On propose une nouvelle interface avec laquelle les contextes ne sont plus directement manipulé dans « l'espace utilisateur » :

```
int create_ctx(func_t f, void *args);
void yield();
```

La primitive `create_ctx()` crée un nouveau contexte pour la fonction `f`, et l'insère dans la liste des contextes « actives ». La primitive `yield()` permet au contexte courant de passer la main à un autre contexte ; ce dernier étant déterminé par l'ordonnancement.

6.1 La liste chaînés dans le noyau Linux

Le noyau Linux utilise une bibliothèque de structures de donnés et de fonctions associées pour implanter les listes chaînées, qui a été optimisé par rapidité et taille en mémoire. Une version réduite de cette bibliothèque est disponible dans le fichier `examples/list.h`, et un exemple d'utilisation dans le fichier `examples/list-example.c`.

Des ressources en ligne sont disponible pour comprendre le fonctionnement de cette bibliothèque, par exemple :

— The Linux Kernel Documentation : `https://www.kernel.org/doc/html/v4.15/core-api/kernel-api.html`
— KernelNewbies/FAQ : Linked Lists : `https://kernelnewbies.org/FAQ/LinkedLists`

6.2 Exercice : création d'un contexte

— Étendez la structure de donnée `struct ctx_s` pour créer la liste chaînée des contextes existants, en utilisant la bibliothèque de liste chaînées de Linux.
— Modifiez la primitive `init_ctx()` en une primitive `create_ctx()` pour mettre en place ce chaînage.
— Traitez des conséquences sur les autres primitives.

6.3 Exercice : yield()

Donnez une implantation de `yield()`.

7 Ordonnancement sur interruptions

L'ordonnancement développé jusque ici est un ordonnancement avec partage volontaire du processeur. Un contexte passe la main par un appel explicite à `yield()`. Nous allons maintenant développer un ordonnancement préemptif avec partage involontaire du processeur : l'ordonnanceur va être capable d'interrompre le contexte en cours d'exécution et de changer de contexte. Cet ordonnancement est basé sur la génération d'interruptions. Une interruption déclenche l'exécution d'une fonction associée à l'interruption (un gestionnaire d'interruptions ou `handler`). Le matériel sur lequel nous travaillons permet de définir une fonction `handler()` qui sera associée à l'interruption `TIMER_IRQ` :

```
idt_setup_handler(TIMER_IRQ, handler);
```

Cette interruption est remontée périodiquement du matériel.

Les deux primitives `irq_disable()` et `irq_enable()` devront être définies. Elles permettrons de délimiter des zones de code devant être exécutées de manière non interruptible (voir `idt.h`).

La nouvelle interface que va fournir notre ordonnanceur est la suivante :

```
int create_ctx(int stack_size, func_t f, void *args);
void start_sched();
```

La fonction `start_sched()` va installer les gestionnaires d'interruptions et initialiser le matériel. Notre ordonnanceur est maintenant préemptif, il reste à isoler les sections critiques de code ne pouvant être interrompues par un gestionnaire d'interruptions.

7.1 Exercice : activer / désactiver les interruptions

Ajoutez les appels nécessaires à `irq_disable()` et `irq_enable()` dans le code de l'ordonnanceur.

7.2 Exercice : ordonnanceur round-robin

Coder la fonction `start_sched()` qui installe un handler du timer ; puis l'handler qui appelles l'ordonnanceur pour sélectionner le prochain contexte en exécution et change vers ce contexte.

L'ordonnanceur qu'on vient de realiser est appelé *Round Robin* (tourniquet), parce que, à intervals réguliers il passe la main au prochain contexte.

8 Synchronisation

On introduit un mécanisme de synchronisation entre contextes à l'aide de sémaphores. Un sémaphore est une structure de données composée

— d'un compteur ;
— d'une liste de contextes en attente sur le sémaphore.

Le compteur peut prendre des valeurs entières positives, négatives, ou nulles. Lors de la création d'un sémaphore, le compteur est initialisé à une valeur donnée positive ou nulle ; la file d'attente est vide.

Un sémaphore est manipulé par les deux actions *atomiques* suivantes :

— `sem_down()` (traditionnellement aussi nommée `wait()` ou `P()`). Cette action décrémente le compteur associé au sémaphore. Si sa valeur est négative, le contexte appelant se bloque dans la file d'attente.
— `sem_up()` (aussi nommée `signal()`, `V()`, ou `post()`). Cette action incrémente le compteur. Si le compteur est négatif ou nul, un contexte est choisi dans la file d'attente et devient actif.

Deux utilisations sont faites des sémaphores :

— la protection d'une ressource partagée (par exemple l'accès à une variable, une structure de donnée, une imprimante...). On parle de sémaphore d'exclusion mutuelle ;
  — Typiquement le sémaphore est initialisé au nombre de contextes pouvant concurremment accéder à la ressource (par exemple 1) et chaque accès à la ressource est encadré d'un couple

```
sem_down(S) ;
  <accès à la ressource>
sem_up(S) ;
```

— la synchronisation de contextes (un contexte doit en attendre un autre pour continuer ou commencer son exécution).
  — Par exemple un contexte 2 attend la terminaison d'un premier contexte pour commencer.) On associe un sémaphore à l'événement, par exemple `findupremier`, initialisé à 0 (l'événement n'a pas eu lieu) :

```
Contexte 1 :                    Contexte 2 :
  <action 1>                      sem_down(findupremier);
  sem_up(findupremier) ;          <action 2>
```

Bien souvent on peut assimiler la valeur positive du compteur au nombre de contextes pouvant acquérir librement la ressource ; et assimiler la valeur négative du compteur au nombre de contextes bloqués en attente d'utilisation de la ressource. Un exemple classique est donné dans la section suivante.

8.1 Producteur / consommateur

Une solution du problème du producteur consommateur au moyen de sémaphores est donnée ici. Les deux utilisations types des sémaphores sont illustrées.

```c
#define N 100                          /* nombre de places dans le tampon */

struct sem_s mutex, vide, plein;

sem_init(&mutex, 1);                 /* controle d'acces au tampon */
sem_init(&vide, N);                  /* nb de places libres */
sem_init(&plein, 0);                 /* nb de places occupees */

void producteur (void)
{
  objet_t objet ;

  while (1) {
    produire_objet(&objet);          /* produire l'objet suivant */
    sem_down(&vide);                 /* dec. nb places libres */
    sem_down(&mutex);                /* entree en section critique */
    mettre_objet(objet);             /* mettre l'objet dans le tampon */
    sem_up(&mutex);                  /* sortie de section critique */
    sem_up(&plein);                  /* inc. nb place occupees */
  }
}

void consommateur (void)
{
  objet_t objet ;

  while (1) {
    sem_down(&plein);                /* dec. nb emplacements occupes */
    sem_down(&mutex);                /* entree section critique */
    retirer_objet (&objet);          /* retire un objet du tampon */
    sem_up(&mutex);                  /* sortie de la section critique */
    sem_up(&vide);                   /* inc. nb emplacements libres */
    utiliser_objet(objet);           /* utiliser l'objet */
  }
}
```

Persuadez-vous qu'il n'est pas possible pour le producteur (resp. le consommateur) de prendre le sémaphore `mutex` avant le sémaphore `plein` (resp. `vide`).

Testez votre implantation des sémaphores sur un exemple comme celui-ci.

Ajoutez une boucle de temporisation dans le producteur que le changement de contexte puisse avoir lieu avant que le tampon ne soit plein.

Essayez d'inverser les accès aux sémaphores `mutex` et `plein=/=vide` ; que constatez-vous ?

8.2 Exercice : implantation des sémaphores

— Donnez la déclaration de la structure de donnée associée à un sémaphore.

— Proposez une implantation de la primitive

```c
void sem_init(struct sem_s *sem, unsigned int val);
```

— Proposez une implantation des deux primitives

```c
void sem_down(struct sem_s *sem);
void sem_up(struct sem_s *sem);
```

9 Un pilote de péripherique

9.1 Exercice : 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 fourni dans 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 produit par le contrôleur clavier ne sont pas des codes ascii. Mais surtout plusieurs codes sortent pour une seule frappe.

— expliquez les différents code 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.

Attention : avec l'option `-curses`, QEMU n'a pas accès directement à votre clavier, il ne reçoit que le flux de caractères transmis par le terminal. Ainsi, s'il reçoit le caractère `T` (lettre T majuscule), il va produire une séquence d'événements clavier qui aurait produit un `T`, pas forcément celle que vous avez réellement effectué. En particulier, vous verrez les mêmes événements que vous utilisiez **Caps Lock** ou **Shift** pour produire votre `T`. Ce problème ne se pose pas avec la version graphique, car QEMU a alors accès aux véritables événements clavier.

9.2 Exercice : pilote du clavier

Le programme de saisie de touche au clavier que vous avez réalisé dans l'exércice précèdent 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 trés 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 perdue.

Pour éviter cela, il est plus pertinent d'organiser l'architecture de votre petit système de tel sorte que la reception 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

Developpez une fonction associée à l'interruption clavier (interruption de niveau 1). 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 préssée. Pour réaliser cela, proposez une solution qui utilise les sémpahores implémentées précédemment. Lorsqu'une interruption clavier à lieu, elle peut "débloquer" un contexte qui attendrait ou bien éviter au prochain appel d'être bloqué, puisque un charactère est disponible.

Question 3.3 : Gérer une file d'entrées au clavier

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