INSA Lyon
4 année, spécialité Informatique

Projet longue durée

Annexe : Systèmes d'exploitation

Kevin Marquet

Décembre 2011

Table des matières

Chapitre 1

Introduction

Ce document décrit la partie “Système d’exploitation” du *Projet Longue Durée* “Système et Réseaux”. Il guide l’implémentation de diverses parties d’un noyau de système d’exploitation, à savoir un ordonnanceur, un mécanisme de synchronisation d’exclusion mutuelle entre processus, un gestionnaire de mémoire dynamique et un petit système de fichier. Attention, parmi ces composants, seuls deux sont obligatoires : l’ordonnanceur et le gestionnaire de mémoire dynamique. Les deux autres font partie des modules optionnels parmi lesquels vous pouvez choisir (voir le sujet principal du PLD).

Attention, tous les développements effectués dans le cadre de cette partie du projet seront faits **exclusivement sous Linux**.

Je remercie vivement les enseignants du master informatique de Lille 1 pour m’avoir permis de m’inspirer de façon conséquente de leur travail.

Chapitre 2

Processus et ordonnancement

Vous mettrez en place votre code de la façon suivante :

1. Téléchargez l'archive `pld_ctx.tgz` sur moodle
2. Décompressez-la : `tar xzf pld_ctx.tgz`
3. Allez dans le dossier srchw : '`cd srchw`'
4. Compilez le simulateur de matériel. Pour cela, un `Makefile` vous est fourni et vous n'avez donc qu'à taper '`make`'
5. Installez les fichiers nécessaires en tapant '`make install`'. Par défaut, cette installation est faite dans le répertoire `../libhw`. Dans ce répertoire, on trouvera les répertoires suivants :
   - `lib` contenant la librairie avec laquelle votre code sera lié ;
   - `include` contenant les fichiers d'entête.
6. Allez dans le répertoire dans lequel vous implémenterez votre ordonnanceur : '`cd ../src`'. Le fichier `Makefile` fourni permet de compiler — tapez '`make`' — le fichier `main.c` en le liant avec le simulateur de matériel. Vous n'avez plus qu'à écrire ce fichier ! En réalité, il est conseillé d'en écrire plusieurs au fur et à mesure et de modifier le Makefile afin qu'il vous compile ce que vous voulez quand vous voulez de manière à travailler efficacement et confortablement.

2.1 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*.

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.

```
#include <setjmp.h>

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

Exercice 1 Illustration du mécanisme de `setjmp()/longjmp()`

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

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

int i = 0;
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 );
}
```

On passe dans le `else` puis on revient dans le `if` pour finalement afficher 0.
et de sa modification :

```
#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 );
}
```

Où l'on explique que ce n'est pas la pile d'exécution qui est sauvegardée (et donc la valeur de `j`) mais la référence vers la pile (qui est en registres `%esp`/`%ebp`) et d'autres registres . Après le `longjmp()`, `j` garde sa valeur 5. On affiche donc -5.

Exercice 2 Lisez la documentation

Expliquez en quoi le programme suivant est erroné :

```
#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)`

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

Un grand classique, le retour dans la pile qui n'est plus active !

Exercice 3 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 */
```

Assembleur en ligne dans du code C

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 :

```
int
main(void)
{
    int x = 10, y;

    asm ("movl %1, %%eax" "\n\t" "movl %%eax, %0"
          : "=r"(y) /* y is output operand */
          : "r"(x) /* x is 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`.

```
            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);
    }
```

On sauvegarde le contexte de `main()` avant l'appel à `mul()` dans une variable globale de type `jmp_buf` :

```
static jmp_buf ctx;
...
    if ( ! setjmp(ctx))
        product = mul(0);
    else
        product = 0;
```

et dans `mul()` on retourne à ce contexte à la rencontre d'un zéro :

```
            case 1 :
                if (i)
                    return i * mul(depth+1);
                else
                    longjmp(ctx, 1 /* dummy */);
```

Environnement 32 bits

Nous avons choisi de travailler sur les microprocesseurs Intel x86, c'est-à-dire compatibles avec le jeu d'instructions de l'Intel 8086. Les versions les plus récentes de cette famille (depuis 2003 quand même !) sont des microprocesseurs 64 bits (Athlon 64, Opteron, Pentium 4 Prescott, Intel Core 2, etc.).
Ces processeurs 64 bits peuvent fonctionner en mode compatibilité 32 bits avec le jeu d'instructions restreint de l'Intel 8086.
Sur ces machines, on indique au compilateur GCC de générer du code 32 bits en lui spécifiant l'option `-m32`.

Première réalisation pratique

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.

Implantez votre bibliothèque de retour dans la pile d'exécution et testez son comportement sur une variante du programme de l'exercice 2.1.
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.

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 ; voir l'encart page précédente.

`dump_sp()`. Ce ne peut être une fonction, ce doit être une macro. On note le `do { ... } while(0)` autorisant la déclaration d'un bloc et l'utilisation `dump_sp();` (on note le `;`).

```
#define dump_sp()                                \
do {                                             \
    void *esp, *ebp;                             \
    asm("mov %%esp,%0" "\n\t"                    \
        "mov %%ebp,%1"                           \
```

```
        : "=r" (esp),                           \
          "=r" (ebp));                          \
    printf("esp   =\t %p\n"                     \
           "ebp   =\t %p\n",                    \
           esp, ebp);                           \
} while(0)
```

Merci à *backslashify* d'emacs (`C-c C-\`).

Adresses des variables automatiques locales et des paramètres. Rien de sorcier à écrire.

```
unsigned int
foo(unsigned int first_arg,
    unsigned int dummy1a, unsigned int dummy2a, unsigned int dummy3a,
    unsigned int last_arg)
{
    unsigned int *first_decl = (unsigned int*) &first_decl;
    unsigned int dummy1d, dummy2d, dummy3d;
    unsigned int *last_decl = (unsigned int*) &last_decl;

    printf("f_arg =\t %p\n", (void *) &first_arg);
    printf("l_arg =\t %p\n", (void *) &last_arg);
    printf("f_decl=\t %p\n", (void *) first_decl);
    printf("l_decl=\t %p\n", (void *) last_decl);
    dump_sp();

    return (unsigned) last_decl;
}
```

Explications. On trouve sur la pile suivant des adresses croissantes :

- l'adresse référencée par `%esp`
- la dernière déclaration locale
- ...
- la première déclaration locale
- l'adresse référencée par `%ebp`
- le dernier argument
- ...
- le premier argument

On peut en conclure que la pile croit suivant des adresses décroissantes.

Implantation d'une bibliothèque de retour dans la pile d'exécution

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

```
typedef int (func_t)(int); /* a function that returns an int from an 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()`.

```
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()`.

Exercice 4 Implantation d'un retour dans la pile d'exécution

Question 4.1 Définissez la structure de données `struct ctx_s`. Les deux pointeurs de pile et un détrompeur :

```
struct ctx_s {
    void *ctx_esp;
    void *ctx_ebp;
#define CTX_MAGIC 0xDEADBEEF
    unsigned ctx_magic;
};
```

Question 4.2 La fonction `try()` sauvegarde un contexte et appelle la fonction passée en paramètre. Donnez une implantation de `try()`. La valeur retournée est celle retournée par la fonction `f()`. C'est immédiat, ce ne sont pas deux instructions assembleur qui doivent effrayer... Je ne pense pas que le `volatile` soit utile.

```
volatile int
try(struct ctx_s *pctx, func_t *f, int arg)
{
    pctx->ctx_magic = CTX_MAGIC;

    asm("mov %%esp,%0"
        "\n\t"
        "mov %%ebp,%1"
        : "=r" (pctx->ctx_esp),
          "=r" (pctx->ctx_ebp));

    return f(arg);
}
```

On peut rappeler la concaténation des chaînes de caractères littérales effectuée par le compilateur C (premier argument de `asm`)...

Question 4.3 Il s'agit de proposer 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 à `throw()`. Tout est dit, si ce n'est que l'on ne peut pas plus référencer les paramètres que les valeurs automatiques une fois le contexte restauré. On passe donc par une variable `static`, qu'elle soit globale ou, mieux, locale.

```
volatile int
throw(struct ctx_s *pctx, int r)
{
    static int throwr = 0; /* yes, static */

    assert(pctx->ctx_magic == CTX_MAGIC);
    pctx->ctx_magic = 0;

    throwr=r;

    asm("mov %0,%%esp"
        "\n\t"
        "mov %1,%%ebp"
        :
        : "r" (pctx->ctx_esp),
          "r" (pctx->ctx_ebp));
```

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.

```
    return throwr;
}
```

Attention, la fonction `try`est non terminale (elle comporte un appel de fonction). Le code assembleur généré (`gcc -S`) se termine donc par `leave` puis `ret`. Il doit en être de même pour `throw`. C'est le cas ici (appel de `assert`). Sinon le code assembleur généré est différent (un simple `popl` avant le `ret`...) et notre mécanisme ne fonctionne plus.

2.2 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.

Exercice 5

1. 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). Avec le modèle précédent, on obtient par mimétisme :

   ```
   typedef void (func_t)(void *);
   ```

2. Étendez la structure de donnée `struct ctx_s` qui décrit un tel contexte. Un `enum` pour les différents états possibles d'un contexte : READY, un contexte qui n'a encore jamais été activé ; ACTIVABLE, un contexte euh.. activable, TERMINATED : la fonction a terminée....

   Un champ par élément décrit ci-dessus dans la structure :

   ```
   enum ctx_state_e {CTX_READY, CTX_ACTIVABLE, CTX_TERMINATED} ;
   struct ctx_s {
       enum ctx_state_e ctx_status;

       func_t *ctx_f;
       void   *ctx_args;

       void *ctx_esp;
       void *ctx_ebp;
   ```

```
    unsigned char *ctx_stack;

#define CTX_MAGIC 0xCAFEBABE
    unsigned int ctx_magic;
};
```

Exercice 6

Proposez une procédure

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

qui initialise le contexte `ctx` avec une pile d'exécution de `stack_size` octets allouée dynamiquement (voir l'encart page précédente à propos de cette allocation). Lors de sa première activation ce contexte appellera la fonction `f` avec le paramètre `args`. On mémorise chacun des champs :

```
int init_ctx(struct ctx_s *ctx, int stack_size,
             func_t f, void *args)
{
    /* stack allocation */
    ctx->ctx_stack = malloc(stack_size);
    if (! ctx->ctx_stack)
        return RETURN_FAILURE;

    /* frame initialisation, the first int @ in the stack */
    ctx->ctx_esp = &ctx->ctx_stack[stack_size-4];
    ctx->ctx_ebp = &ctx->ctx_stack[stack_size-4];

    /* func and args */
    ctx->ctx_f = f;
    ctx->ctx_args=args;

    /* misc. */
    ctx->ctx_status = CTX_READY;
    ctx->ctx_magic = CTX_MAGIC;

    return RETURN_SUCCESS;
}
```

2.3 Changement de contexte

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...

```
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, 16384, f_ping, NULL);
    init_ctx(&ctx_pong, 16384, f_pong, NULL);
    switch_to_ctx(&ctx_ping);

    exit(EXIT_SUCCESS);
}

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

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

L'exécution de ce programme produit sans fin :

```
A1B2C1A2B1C2A1...
```

Cet exemple illustre la procédure

```
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).

Exercice 7

Proposez une implantation de la procédure `switch_to_ctx()`. On utilise une variable globale `current_ctx` qui référence le contexte courant. Elle est initialisée à `NULL` ;

```
static struct ctx_s *current_ctx = (struct ctx_s *)0;
```

Le changement de contexte mémorise l'ancien et passe au nouveau :

```
void
switch_to_ctx(struct ctx_s *newctx)
{
    assert(newctx->ctx_magic == CTX_MAGIC);

    assert(newctx->ctx_status == CTX_READY
           || newctx->ctx_status == CTX_ACTIVABLE);

    /* a null value of current_ctx on the very first switch_to */
    if (current_ctx) {
        asm("mov %%esp,%0 "
            : "=r" (current_ctx->ctx_esp));
        asm("mov %%ebp,%0 "
            : "=r" (current_ctx->ctx_ebp));
    }

    /* switch */
    current_ctx = newctx;

    asm("mov %0,%%esp"
        :
        : "r" (current_ctx->ctx_esp));
    asm("mov %0,%%ebp"
        :
        : "r" (current_ctx->ctx_ebp));

    /* is it the first swith to the context? */
    if(current_ctx->ctx_status == CTX_READY)
        start_current_ctx();
}
```

La première activation lance la fonction et ... ne termine pas pour le moment...

```
static void
start_current_ctx()
{
    struct ctx_s *ctx = current_ctx;

    /* change status */
    ctx->ctx_status = CTX_ACTIVABLE;

    /* execute function */
    ctx->ctx_f(ctx->ctx_args);

    /* the function has ended */
    ctx->ctx_status = CTX_TERMINATED;
    free(ctx->ctx_stack);

    /* i do not know any other context, so i keep the CPU... */
    while(1) ;
}
```

2.4 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 `init_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(int stack_size, func_t f, void *args);
void yield();
```

La primitive `create_ctx()` ajoute à l'ancien `init_ctx()` l'allocation dynamique initiale de la structure mémorisant le contexte. La primitive `yield()` permet au contexte courant de passer la main à un autre contexte ; ce dernier étant déterminé par l'ordonnancement.

Exercice 8

1. Étendez la structure de donnée `struct ctx_s` pour créer la liste chaînée des contextes existants.
2. Modifiez la primitive `init_ctx()` en une primitive `create_ctx()` pour mettre en place ce chaînage.
3. Traitez des conséquences sur les autres primitives.

Un champ `ctx_next` est simplement ajouté :

```
struct ctx_s {
    enum ctx_state_e ctx_status;

    func_t *ctx_f;
    void   *ctx_args;

    void *ctx_esp;
    void *ctx_ebp;

    unsigned char *ctx_stack;

    struct ctx_s *ctx_next;

#define CTX_MAGIC 0xCAFEBABE
    unsigned int ctx_magic;
};
```

Dans `create_ctx()` on réalise le chaînage, à partir du point d'entrée de la liste chaînée `ctx_ring` (vouloir confondre `ctx_ring` et `current_ctx` met la pagaille dans la première invocation de `switch_to_ctx()` :

```
static struct ctx_s *ctx_ring = (struct ctx_s *)0;

int create_ctx(int stack_size, func_t f, void *args)
{
    struct ctx_s *ctx;
```

```
    /* struct allocation */
    ctx = malloc(sizeof(struct ctx_s));
    if (! ctx)
        return RETURN_FAILURE;

[...]

    /* context ring */
    if (!ctx_ring)
        ctx->ctx_next = ctx;
        ctx_ring = ctx;
    } else {
        ctx->ctx_next = ctx_ring->ctx_next;
        ctx_ring->ctx_next = ctx;
    }

    /* misc. */
    ctx->ctx_status = CTX_READY;

[...]
}
```

Les autres conséquences sont

- la suppression du contexte courant de l'anneau des contextes à sa terminaison ;
- le passage effectif de la main à un autre contexte (si possible) à sa terminaison.

(suite)

On peut remarquer qu'il est difficile de supprimer un contexte alors qu'il est le contexte courant

- on va devoir libérer la mémoire qui correspond à la pile d'exécution courante !
- nous devons parcourir tout l'anneau pour mettre à jour le chaînage par `ctx_next`.

À la terminaison de la fonction, on peut donc identifier le contexte comme terminé et simplement passer la main.

De plus, avant de restaurer un contexte, si il est terminé, on libère les ressources qui lui sont associées.

On modifie donc `start_current_ctx()` :

```
static void
start_current_ctx()
{
    struct ctx_s *ctx = current_ctx;

    /* change status */
    ctx->ctx_status = CTX_ACTIVABLE;

    /* execute function */
    ctx->ctx_f(ctx->ctx_args);

    /* the function has ended */
    ctx->ctx_status = CTX_TERMINATED;

    /* release the CPU */
    yield();
}
```

et dans la fonction `switch_to_ctx`, on ajoute :

```
static void
switch_to_ctx(struct ctx_s *newctx)
```

```
{
    /* ... */

    /* assume we are called from yield() with
        switch_to_ctx(current_ctx->next) */
    while (newctx->ctx_next->ctx_status == CTX_TERMINATED) {
        if (newctx->ctx_next == current_ctx)
            /*** The End : back to main () or exit() ***/
        /* free context stack */
        free(newctx->ctx_stack);

        /* suppress the context from the ring */
        if (ctx_ring == newctx)
            ctx_ring = newctx->ctx_next;
        current_ctx->ctx_next = newctx->ctx_next;
        newctx = current_ctx->ctx_next;
    }
    /* ... */
}
```

Exercice 9

Donnez une implantation de `yield()`. On passe la main au suivant ! Attention, on devrait pouvoir écrire

```
void
yield()
{
    switch_to_ctx(current_ctx->ctx_next);
}
```

mais au premier appel, `current_ctx` n'est pas encore non nul !

```
void
yield()
{
    assert(ctx_ring); /* no context */

    if (! current_ctx)
        switch_to_ctx(ctx_ring);
    else
        switch_to_ctx(current_ctx->ctx_next);
}
```

2.5 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 fournit l'interface suivante :

```
typedef void (irq_handler_func_t)(void);

#define TIMER_IRQ       2

void setup_irq(unsigned int irq, irq_handler_func_t handler);
void start_hw();

void irq_disable();
void irq_enable();
```

La primitive `setup_irq()` associe la fonction `handler` à l'interruption `irq`. Seule l'interruption `TIMER_IRQ` est définie ; elle est remontée périodiquement du matériel.

La primitive `start_hw()` permet d'initialiser le matériel ; elle doit être invoquée pour démarrer la génération des interruptions.

Les deux primitives `irq_disable()` et `irq_enable()` permettent de délimiter des zones de code devant être exécutées de manière non interruptible.

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.

Exercice 10

1. Quel va être le rôle du gestionnaire d'interruptions associé à `TIMER_IRQ` ?
2. Proposez une implantation de `start_sched()`.

Le gestionnaire d'interruptions associé à `TIMER_IRQ` est chargé de réaliser un changement de contexte, i.e. un appel à `yield()`.

La fonction `start_sched()` initialise le matériel puis définit `yield` comme la fonction associée à `TIMER_IRQ` :

```
void
start_sched()
{
    start_hw();
    setup_irq(TIMER_IRQ, yield);
    while(1)
        pause();
}
```

Cette fonction ne rend pas la main de suite pour éviter que le programme ne termine avant qu'un IT soit survenue. On peut aussi imaginer appeler `yield()` de suite.

Notre ordonnanceur est maintenant préemptif, il reste à isoler les sections critiques de code ne pouvant être interrompues par un gestionnaire d'interruptions.

Exercice 11

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

Les manipulations des listes de contextes doivent être protégées ; dans `create_ctx()` on trouve :

```
/* context ring */
irq_disable();
```

```
if (!ctx_ring) {
    ctx->ctx_next = ctx;
    ctx_ring = ctx;
} else {
    ctx->ctx_next = ctx_ring->ctx_next;
    ctx_ring->ctx_next = ctx;
}

irq_enable();
```

et à la terminaison d'un contexte dans `start_current_ctx()` on trouve :

```
/* the function has ended */
ctx->ctx_status = CTX_TERMINATED;
irq_disable();
free(ctx->ctx_stack);

/* suppress the context from the ring */
if (ctx_ring == current_ctx)
    ctx_ring = current_ctx->ctx_next;
if (current_ctx->ctx_next == current_ctx) {
    /* last context... */
    fprintf(stderr, "No more context in ring, bye\n");
    exit(EXIT_SUCCESS);
} else {
    struct ctx_s *prev = ctx_ring;

    while (prev->ctx_next != current_ctx)
        prev = prev->ctx_next;
    prev->ctx_next = current_ctx->ctx_next;

    irq_enable();
    ....
```

Les manipulations de registres dans le changement de contexte lui-même doit être protégées :

```
irq_disable();

/* a null value of current_ctx on the very first switch_to */
if (current_ctx) {
    asm("mov %%esp,%0 "
        : "=r" (current_ctx->ctx_esp));
    asm("mov %%ebp,%0 "
        : "=r" (current_ctx->ctx_ebp));
}

/* switch */
current_ctx = newctx;

asm("mov %0,%%esp"
    :
    : "r" (current_ctx->ctx_esp));
asm("mov %0,%%ebp"
    :
    : "r" (current_ctx->ctx_ebp));

irq_enable();
```

Simulateur de matériel

La bibliothèque `hw` qui vous est fournie implémente l'interface décrite section 2.5 à l'aide signaux et timers POSIX. Vous pouvez en étudier le code ou l'utiliser sans vous en soucier...

2.6 Synchronisation entre contextes

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 processus 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 processus est choisi dans la file d'attente est 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 ;

Rappel de cours

Typiquement le sémaphore est initialisé au nombre de processus 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 processus (un processus doit en attendre un autre pour continuer ou commencer son exécution).

Rappel de cours

(Par exemple un processus 2 attend la terminaison d'un premier processus pour commencer.) On associe un sémaphore à l'événement, par exemple `findupremier`, initialisé à 0 (l'événement n'a pas eu lieu) :

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

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

Exercice 12

Le classique producteur consommateur

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

```
#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 ? Votre implémentation peut-elle détecter de tels comportements ?

1. En remarquant qu'un contexte donnée ne peut être bloqué que dans une unique file d'attente d'un sémaphore, étendez la structure de données associée à un contexte pour gérer les files d'attente des sémaphores.
2. Donnez la déclaration de la structure de donnée associée à un sémaphore.
3. Proposez une implantation de la primitive

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

   Il suffit de mémoriser que le contexte est bloqué en ajoutant un nouvel état :

   ```
   enum ctx_state_e {CTX_READY, CTX_ACTIVABLE, CTX_TERMINATED, CTX_BLOCKED};
   ```

   Un sémaphore est un couple (valeur, file d'attente). La file d'attente est l'ensemble des contextes bloqués sur **ce** liées par un champ additionnel `ctx_sem_list` :

   ```
   struct ctx_s {
       enum ctx_state_e ctx_status;
       ...
       struct ctx_s *ctx_sem_list; /* next task BLOCKED on the semaphore */
   };


   struct sem_s {
       int sem_cpt;
       struct ctx_s *sem_list; /* task BLOCKED on the semaphore */
   };
   ```

   `sem_create()` initialise le compteur avec le paramètre et la liste à vide :

   ```
   void
   sem_init(struct sem_s *sem, unsigned int val)
   {
       sem->sem_cpt = val;
       sem->sem_list = (struct ctx_s *)0;
   }
   ```

Exercice 13

Proposez une implantation des deux primitives

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

Si on lit le sujet : `sem_up` incrémente le compteur. Si il y a au moins un contexte bloqué, on libère **exactement un** contexte.

On utilise le fait que compteur ne peut être initialisé à une valeur négative (`unsigned`) : quand ce compteur est négatif, on est assuré qu'il y a un contexte bloqué sur le sémaphore :

```
void
sem_up(struct sem_s *sem)
{
    irq_disable();

    sem->sem_cpt++;

    if (sem->sem_cpt <= 0) {
        sem->sem_list->ctx_status = CTX_ACTIVABLE;
        sem->sem_list = sem->sem_list->ctx_sem_list;
    }
```

```
    irq_enable();
}
```

On peut commencer par une version sans les `irq_*able()`.

De même `sem_down()` décrémente le compteur. Si le compteur passe à une valeur négative, on se bloque :

```
void
sem_down(struct sem_s *sem)
{
    irq_disable();

    sem->sem_cpt--;

    if (sem->sem_cpt < 0) {
        current_ctx->ctx_status = CTX_BLOCKED;
        current_ctx->ctx_sem_list = sem->sem_list;
        sem->sem_list = current_ctx;

        irq_enable();
        yield();
    } else {
        irq_enable();
    }
}
```

Il s'agit aussi de modifier `switch_to()` pour ne pas donner la main à un contexte bloqué :

```
switch_to_ctx(struct ctx_s *newctx)
{
    struct ctx_s *first_newctx = newctx;

    assert(newctx->ctx_magic == CTX_MAGIC);

    /* skip BLOCKED contexts */
    while (newctx->ctx_status == CTX_BLOCKED) {
        newctx = newctx->ctx_next;
        if (newctx == first_newctx) /* all context of the ring are BLOCKED! */
            [...]
    }

    assert(newctx->ctx_status == CTX_READY
           || newctx->ctx_status == CTX_ACTIVABLE);

    [...]
```

Chapitre 3

Prévention des interblocages (optionnel)

Sur une idée de Gilles Grimaud

On ajoute aux sémaphores introduit précédemment un mécanisme d'exclusion mutuel sous la forme de simples verrous :

- un verrou peut être libre ou verrouillé par un contexte ; ce contexte est dit propriétaire du verrou ;
- la tentative d'acquisition d'un verrou non libre est bloquante.

L'interface de manipulation des verrous est la suivante :

```
void mtx_init(struct mtx_s *mutex);
void mtx_lock(struct mtx_s *mutex);
void mtx_unlock(struct mtx_s *mutex);
```

Comparés aux sémaphores, l'utilisation des verrous est plus contraignantes : seul le contexte propriétaire du verrou peut le libérer et débloquer un contexte en attente du verrou. De manière évidente, les verrous peuvent être simulés par des sémaphores dont la valeur initiale du compteur serait 1.

Exercice 14

L'académique et néanmoins classique problème des philosophes est le suivant : cinq philosophes attablés en cercle autour d'un plat de spaghettis mangent et pensent alternativement sans fin (faim ?). Une fourchette est disposée entre chaque couple de philosophes voisins. Un philosophe doit préalablement s'emparer des deux fourchettes qui sont autour de lui pour manger.

On désire élaborer une solution à ce problème en attachant un contexte à l'activité de chacun des philosophes et un verrou à chacune des fourchettes.

Montrez qu'une solution triviale peut mener à un interblocage, aucun des philosophes ne pouvant progresser. Une solution triviale est quelque chose de la forme suivante :

```
struct mtx_s forks[5];

for (i=0 ; i<5 ; i++)
    mtx_init(forks+i);

for (i=0 ; i<5 ; i++)
    create_ctx(16384, philo, (void *) i);

void
```

```
philo(void *arg)
{
    int i = (int) arg;
    think();
    mtx_lock(forks[i]);
    mtx_lock(forks[(i+1)%5]);
    eat()
    mtx_unlock(forks[i]);
    mtx_unlock(forks[(i+1)%5]);
}
```

Le scénario suivant amène à un interblocage si on considère par exemple un ordonnancement tel que les contextes perdent systématiquement la main entre l'acquisition des deux fourchettes. Chaque philosophe détient une fourchette et se bloque sur le second `mtx_lock`.

Bien entendu cette situation est improbable, en TP, on pourra forcer la préemption et écrire quelque chose comme :

```
void
philo(void *arg)
{
    int i = (int) arg;
    think();
    mtx_lock(forks[i]);
    yield();            /* looking after a deadlock */
    mtx_lock(forks[(i+1)%5]);
    eat()
    mtx_unlock(forks[i]);
    mtx_unlock(forks[(i+1)%5]);
}
```

Exercice 15

Comment le système peut-il prévenir de tels interblocages ?

On considérera que
- un contexte est bloqué sur un verrou ;
- un verrou bloque un ensemble de contextes ;
- un contexte détient un ensemble de verrous.

Considérez aussi les situations dans lesquelles toutes les activités ne participent pas à l'interblocage. Par exemple, une sixième activité indépendante existe en dehors des cinq philosophes.

On modifie l'interface de manipulation des verrous pour que le verrouillage retourne une erreur en cas d'interblocage :

```
void mtx_init(struct mtx_s *mutex);
int mtx_lock(struct mtx_s *mutex);
void mtx_unlock(struct mtx_s *mutex);
```

Exercice 16

Donner une implémentation de ces primitives détectant les interblocages.

3.1 Création de contextes par duplication

Sur une idée de Laurent Noé.

Convention d'appel et organisation de la pile (partie 1)

Une convention d'appel est une méthode qui assure, lors d'un appel de fonction, la cohérence entre ce qui est réalisé par la fonction appelante et la fonction appelée. En particulier, une convention d'appel précise comment les paramètres et la valeur de retour sont passées, comment la pile est utilisée par la fonction.

Conventions d'appel Plusieurs conventions sont utilisées :

- `__cdecl` est la convention qui supporte la sémantique du langage C et en particulier l'existence de fonctions variadiques (fonctions à nombre variable de paramètres, par exemple `printf`). Cette convention est le standard de fait, en particulier sur architecture x86 ;
- `__stdcall` est une convention qui suppose que toutes les fonctions ont un nombre fixe de paramètres. Cette convention simplifie le nettoyage la pile après un appel de fonction ;
- `__fastcall` est une convention qui utilise certains registres pour passer les premiers paramètres de la fonction.

Registres utilisés Sur Intel x86, ces conventions utilisent toutes les registres suivants :

- le registre `%esp` est manipulé implicitement par certaines instructions telles `push`, `pop`, `call`, et `ret`. Ce registre contient toujours l'adresse sur la pile du dernier emplacement utilisé (et non du premier emplacement libre). On oubliera pas que la pile est gérée suivant les adresses décroissantes. Le sommet de la pile est donc toujours à une adresse la plus basse.
- le registre `%ebp` est utilisé comme base pour référencer les paramètres, les variables locales, etc. dans la fenêtre courante. Ce registre n'est manipulé qu'explicitement. L'implantation d'une convention d'appel repose principalement sur ce registre.
- le registre `%eip` contient l'adresse de la prochaine instruction à exécuter. Le couple d'instructions `call`/`ret` sauvegarde et restaure ce registre sur la pile. Bien entendu, chacune des instructions de saut modifie ce registre.

On envisage maintenant la duplication d'un contexte existant. Cela nécessite quelques manipulations de la pile d'exécution qui sont préalablement introduites.

Manipulation de la pile d'exécution

La taille de la pile associée à un contexte est choisie à la création du contexte. Déterminer une taille optimale est une chose délicate. On se propose d'automatiser la variation de la taille de cette pile.

Exercice 17

Proposez une fonction ou macro

```
int check_stack(struct ctx_s *pctx, unsigned size);
```

qui vérifie que la taille de pile disponible pour un contexte donné est supérieure à une valeur en octets donnée. La pile croît selon les adresses décroissantes. La pile est donc presque pleine quand la valeur `ctx->ctx_esp` est proche de (mais plus grande que !) `ctx->ctx_stack`.

On écrit donc

```
#define check_stack(pctx,size)
    (((char *)(pctx->ctx_esp) - (char *)(pctx->ctx_stack)) >= size)
```

les `(char *)` pour faire le calcul sur des octets...

Nous nous proposons de réallouer cette pile quand la taille libre restante devient petite.

Exercice 18

Discutez des moments opportuns auxquels réaliser cette réalllocation. Par exemple vérifier la

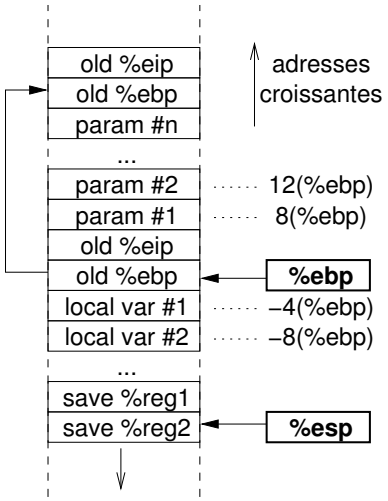
Convention d'appel et organisation de la pile (partie 2)

Appel d'une fonction `__cdecl` Les étapes suivantes sont réalisées lors de l'appel d'une fonction selon la convention `__cdecl` :

- **empilement des valeurs des paramètres** (de droite à gauche). L'appelant mémorise combien paramètres ont été empilés (en fait la taille en octets de l'ensemble des paramètres) ;
- **appel de la fonction** via un `call`. Le registre `%eip` est sauvegardé sur la pile avec la valeur de l'instruction qui suit le `call` ;
- **mise à jour du pointeur de pile**. On est maintenant dans le code de la fonction appelée. La pile de cette fonction débute au dessus de la pile de la fonction appelante ; on empile l'ancienne valeur de `%ebp` et on lui affecte la valeur actuelle de l'adresse de sommet de pile :

  ```
  push %ebp
  mov  %esp, %ebp
  ```

  À partir de cette nouvelle valeur de `%ebp`, la fonction appelée peut accéder à ses arguments par un déplacement positif (car la pile est rangée suivant les adresses décroissantes) : `8(%ebp)` référence le premier paramètre, `12(%ebp)` le second, etc.
  À partir de cette nouvelle valeur de `%ebp`, on peut aussi retrouver l'ancien pointeur d'instruction à `4(%ebp)` et l'ancienne valeur de `%ebp` à `0(%ebp)` ;



- **allocation des variables locales**. La fonction peut allouer ses variables locales dans la pile en décrémentant simplement le registre `%esp`. Les accès à ces variables se feront par un déplacement négatif par rapport à `%ebp` : `-4(%ebp)` pour la première variable, etc. ;
- **sauvegarde de registres**. Si la fonction utilise des registres pour l'évaluation d'expressions, elle les sauvegarde à ce niveau sur la pile et devra les restaurer avant le retour à la fonction appelante ;
- **exécution du code de la fonction**. L'état de la pile est ici cohérent et le registre `%ebp` est utilisé pour accéder aux paramètres et à variables locales. Le code de la fonction peut accéder aux registres qui ont été sauvegardés, mais ne peut pas modifier la valeur du registre `%ebp`. Les allocations supplémentaires dans la pile se font par des manipulation du registre `%esp` ; cependant ces allocations doivent être compensées par autant de libérations ;
- **restauration des registres**. Les registres sauvegardés à l'entrée de la fonction doivent maintenant être restaurés ;
- **restauration de l'ancien pointeur de pile**. La restauration de l'ancienne valeur de `%ebp` a pour effet de supprimer toutes les allocations réalisées par la fonction et de remettre la pile dans l'état correct pour la fonction appelante ;
- **retour à la fonction appelante**. L'instruction `ret` récupère l'ancienne valeur de `%eip` sur la pile et saute à cette adresse. Le flux d'exécution retourne ainsi dans la fonction appelante ;
- **nettoyage de la pile**. La fonction appelante se doit de nettoyer la pile des valeurs des paramètres qu'elle y avait empilées.

taille restante dans la pile d'un contexte avant de réaliser le `switch_to`vers ce contexte... Insuffisant mais...

Une fois la décision de réallouer la pile prise et une nouvelle zone mémoire obtenue, il s'agit de translater la pile dans cette nouvelle zone mémoire : les références comportant des adresses dans l'ancienne pile doivent être modifiées. En particulier, le chaînage des anciennes valeurs de `%ebp` doit être mis à jour.

Exercice 19

Listez l'ensemble des références vers des adresses dans la pile. En particulier explicitez comment identifier toutes les valeurs de chaînage des anciennes valeurs de `%ebp`.

Proposez une fonction

```
void translate_stack(struct ctx_s *ctx,
                     unsigned char *nstack, unsigned nstack_size);
```

translation de la pile d'un contexte. Cette fonction suppose que la mémoire de la nouvelle pile a été allouée et que le contenu de l'ancienne pile y a déjà été copié.

- primo : la valeur de la translation, car il s'agit bien d'une translation : l'adresse *haute* de la nouvelle pile - l'adresse *haute* de l'ancienne pile
  en effet la pile est organisée en utilisant d'abord les adresses hautes et en croissant vers les adresses basses
- on ne peut en l'état trouver l'adresse haute de l'ancienne pile : on ne connaît que les valeurs de `%esp` et `%ebp`, qui sont d'aucune utilité pour cela, et l'adresse basse de la pile. Il nous faut la taille de cette pile. On ajoute donc un champs `ctx_stack_size`à la structure `struct cxt_s`
- dans la structure `ctx`elle-même, les trois champs `ctx_stack`, `ctx_esp`, et `ctx_ebp`sont à translater
- chacune des valeurs la liste des anciens `%ebp` est à translater. Cette liste est chaînée et se termine quand une valeur sort de l'ancienne pile !

```
void
translate_stack(struct ctx_s *ctx,
                unsigned char *nstack, unsigned nstack_size)
{
    unsigned int delta;
    unsigned int *ebplist;

    delta = nstack - ctx->ctx_stack + nstack_size - ctx->ctx_stack_size;

    /* ebplist translation */
    ebplist = ctx->ctx_ebp;
    do {
        *(ebplist+delta) += delta;
        ebplist = (unsigned int *) *ebplist;
    } while (ctx->ctx_stack < (char *) ebplist
             && (char *) ebplist < ctx->ctx_stack + ctx->ctx_stack_size);

    /* ctx struct pointer translation */
    ctx->ctx_esp += delta;
    ctx->ctx_ebp += delta;
    ctx->ctx_stack = nstack;
    ctx->ctx_stack_size = nstack_size;
}
```

– il est bien entendu que les pointeurs dans le code utilisateurs qui référencent des valeurs dans la pile ne vont pas être mis à jour (ne peuvent pas être mis à jour...) et peuvent poser problème...

Duplication de contextes

On désire maintenant fournir une primitive

```
int dup_ctx();
```

du type de `fork`qui duplique le contexte courant. Les deux copies du contexte n'étant distinguées que par la valeur retournée par cette fonction de duplication, 0 dans le « fils » et une valeur différente de zéro dans le père.

Dupliquer un contexte consiste à dupliquer la structure `struct ctx_s`associée, mais aussi sa pile d'exécution. Ainsi le fils poursuivra son exécution à la sortie de la fonction `dup_ctx`, mais remontera aussi tous les appels de fonctions qui avaient été faits par le père avant le `dup_ctx`.

Dans un second temps, il sera nécessaire de prendre en compte le point technique suivant : l'exécution du contexte créé par duplication, comme celle de tous les contextes, va reprendre lors d'un appel à `switch_to`. Il est donc important que la pile d'exécution créée par duplication soit « compatible » avec une pile laissée par un appel à `switch_to`. Une telle pile peut être créée par un appel à une fonction de prototype identique à `switch_to`qui va sauvegarder le contexte dans la structure adéquate.

Enfin, on pourra se soucier de la mise en place d'un mécanisme pour retourner une valeur différente dans le contexte père et dans le contexte fils.

Exercice 20

Donnez le code de la fonction `dup_ctx`qui consiste donc à allouer une structure pour le nouveau

contexte, à y copier le contexte du père, à faire un appel à une fonction « compatible » avec `switch_to`qui va sauvegarder les registres dans les champs idoines pour pouvoir revenir à ce contexte ensuite, à allouer la pile de ce nouveau contexte, à y copier celle du père, à translater cette nouvelle pile, enfin à insérer ce nouveau contexte dans l'anneau des contextes.

– on débute par un dessin de la pile du fils qui sera complété au fur et à mesure de l'exécution de la duplication de contexte.
– on réalise les choses telles que précisées dans la question :

```
int
dup_ctx()   /* BUG */
{
    struct ctx_s *newctx;

    /* context struct allocation and copy (some fields will be update later) */
    newctx = malloc(sizeof(struct ctx_s));
    assert(newctx);
    memcpy(newctx, current_ctx, sizeof(struct ctx_x));

    /* newstack allocation */
    newctx->ctx_stack = malloc(current_ctx->ctx_stack_size);
    assert(newctx->ctx_stack);
    /* copy the context stack */
    memcpy(newctx->ctx_stack, current_ctx->ctx_stack,
           current_ctx->ctx_stack_size);

    /* translation of the stack (mainly epb list) */
    translate_stack(newctx, newctx->ctx_stack, newctx->ctx_stack_size);
```

```
    /* link newctx in the context ring */
    newctx->ctx_next = ctx_ring->ctx_next;
    ctx_ring->ctx_next = newctx;
}
```

(suite)

– la fonction « compatible » avec `switch_to`ne fait que sauvegarder les registres dans le nouveau contexte. On place un appel à cette fonction dans `dup_ctx`une fois ce contexte crée

```
    /* newstack allocation */
    ...
    /* copy the context stack */
    ...

    /* make newctx stack switch_to compatible */
    dup_ctx_switch_to(newctx);

    /* translation of the stack (mainly epb list) */
    ...
```

– la fonction elle même

```
void
dup_ctx_switch_to(struct ctx_s *newctx)   /* BUG */
{
    /* just save the context in newctx */
    asm("mov %%esp,%0 "
        : "=r" (newctx->ctx_esp));
    asm("mov %%ebp,%0 "
        : "=r" (newctx->ctx_ebp));
}
```

– On remarque que cette fonction est terminale : elle n'est donc pas entièrement compatible avec `switch_to`. On la modifie donc pour intégrer le code qui précédait l'appel de la fonction dans `dup_ctx` :

```
void
dup_ctx_switch_to(struct ctx_s *newctx)   /* BUG */
{
    /* just save the context in newctx */
    asm("mov %%esp,%0 "
        : "=r" (newctx->ctx_esp));
    asm("mov %%ebp,%0 "
        : "=r" (newctx->ctx_ebp));

    /* copy the context stack */
    memcpy(newctx->ctx_stack, current_ctx->ctx_stack,
           current_ctx->ctx_stack_size);
}
```

– il reste un gros soucis : à son retour de `dup_ctx_switch_to`le fils va continuer l'exécution avec une nouvelle translation de pile, etc.
– il reste aussi à se préoccuper de
  – la valeur de retour : qui va régler le problème précédent
  – le masquage des irqs

(suite)

– valeur de retour : il faut que le père retourne une valeur non nulle (par exemple identifiant le fils...) : simple. Que le fils retourne une valeur nulle... ? La valeur retournée est dans la pile. On va donc directement modifier cette valeur dans la pile du fils en en retrouvant son adresse :

```
int
dup_ctx()
{
    struct ctx_s *newctx;
    unsigned char *newstack;
    int result;
    int delta;

    /* context struct allocation and copy */
    ...

    /* result of the parent */
    result = (int) newctx; /* or any non-nul value */

    /* update the newctx with a call to a switch-to-like function */
    ...

    /* only the parent */
    if (result) {
        /* newstack allocation, copy and translation */
        ...

        /* link newctx in the context ring */
        ...

        /* change the return value of the newctx */
        delta = nstack - ctx->ctx_stack + nstack_size - ctx->ctx_stack_size;
        *(&result + delta) = 0;
    }

    return result;
}
```

– le placement des irqs, on fait simple : pas d'interruption durant l'exécution de toute la fonction pour le père.

Chapitre 4

Allocation dynamique de mémoire

4.1 Une première bibliothèque standard

La *bibliothèque C standard* fournit un ensemble de fonctions permettant l'accès aux services du système d'exploitation. Parmi ces services, on trouve l'allocation et la libération de mémoire, au travers les deux primitives suivantes :

void *malloc (unsigned size) ; La fonction `malloc()` de la bibliothèque retourne un pointeur sur un bloc d'au moins size octets.

void free (void *ptr) ; La fonction `free()` permet de libérer le bloc préalablement alloué pointé par `ptr`, quand il n'est plus utile.

Cette partie du sujet consiste à implémenter vos propres fonctions d'allocation et libération de mémoire (qu'on appelera `gmalloc` et `gfree`). Dans un premier temps, nous réaliserons une implémentation simple et efficace de ces primitives. Dans un deuxième temps, vous les optimiserez.

La mise en place de l'environnement de développement vous est laissée. Vous pouvez vous inspirer de ce qui vous est fourni pour la partie ordonnancement (Makefile...).

4.1.1 Principe

Lors de la création d'un processus, un espace mémoire lui est alloué, contenant la pile d'exécution de ce processus, le code de celui-ci, les variables globales, ainsi que le *tas*, dans lequel les allocations dynamiques effectuées au travers `gmalloc()` sont effectuées. Ce tas mémoire est accessible le champs `heap` de la structure associé au processus (voir le chapitre 2). Au début de l'exécution du processus, ce tas ne contient aucune donnée. Il va se remplir et se vider au grè des appels à `gmalloc()` et `gfree()` : à chaque appel à `gmalloc()`, un *bloc* va être alloué dans le tas, à chaque appel à `gfree`, un bloc va être libéré, menant à une fragmentation du tas.

Dans notre implantation de ces fonctions, l'ensemble des blocs mémoire libres va être accessible au moyen d'une liste chaînée. Chaque bloc contient donc un espace vide, la taille de cette espace vide et un pointeur sur le bloc suivant. Le dernier bloc pointera sur le premier.

4.1.2 Implémentation de gmalloc()

Lors d'un appel à `gmalloc()`, on cherche dans la liste de blocs libres un bloc de taille suffisante. L'algorithme *first-fit* consiste à parcourir cette liste chaînée et à s'arrêter au premier bloc de taille suffisante. Un algorithme *best-fit* consiste à utiliser le "meilleur" bloc libre (selon une définition de "meilleur' donnée'). Nous allons implémenter le *first-fit*.

Si le bloc a exactement la taille demandée, on l'enlève de la liste et on le retourne à l'utilisateur. Si le bloc est trop grand, on le divise en un bloc libre qui est gardé dans la liste chaînée, et un bloc qui est retourné à l'utilisateur. Si aucun bloc ne convient, on retourne un code d'erreur.

4.1.3 Implémentation de gfree()

La libération d'un espace recherche l'emplacement auquel insérer ce bloc dans la liste des blocs libres. Si le bloc libéré est adjacent à un bloc libre, on les fusionne pour former un bloc de plus grande taille. Cela évite une fragmentation de la mémoire et autorise ensuite de retourner des blocs de grande taille sans faire des appels au système.

4.2 Optimisations de la bibliothèque

Votre bibliothèque peut maintenant être optimisée. Implémentez donc les améliorations que vous saurez trouver/imaginer, en vous inspirant entre autres des points suivants :

Pré-allocation Une des optimisations effectuées par la librairie C standard sous Unix est la mise en place de listes chaînés de blocs d'une certaine taille. Gardez en mémoier que ce système est efficace pour de petites tailles.

Détection d'utilisation illégale Les primitives telles qu'elles sont définies peuvent être mal utilisées, et votre implémentation peut favoriser la détection de ces utilisations frauduleuses. Quelques exemples d'utilisation frauduleuse :
- Passage à `gfree()` d'un pointeur ne correspondant pas à un précédent `gmalloc()`
- Supposition de remplissage d'un segment alloué à zéro
- Débordement d'écriture
- Utilisation d'un segment après l'avoir rendu par `gfree()`

Outils Vous pouvez favoriser le débogage en fournissant des outils tels que l'affichage des listes chaînées ou l'affichage des blocs alloués.

Spécialisation aux applications Puisque vous connaissez l'utilisation qui sera faite de votre bibliothèque, vous pouvez spécialiser son fonctionnement. Bien sûr vous perdrez en généricité, il faut savoir dans quelle mesure.

Chapitre 5

Réalisation d'un petit système de fichier (optionnel)

On s'intéresse ici à la réalisation d'un système de fichiers au dessus d'un disque magnétique organisé en pistes et secteurs. Le système de fichiers que nous allons étudier est composé de différentes couches logicielles successives. Nous allons progressivement détailler les fonctionnalités et une implantation possible de ces couches.

5.1 Première couche logicielle : accès au matériel

Nous disposons d'un disque dur organisé en pistes (aussi nommées cylindres), chacune des pistes étant organisée en secteurs. La couche logicielle la plus base définit une interface C avec ce matériel. Cette interface est une forme simplifiée de la norme ATA-2 supportée par les fabricants de disque dur de type IDE (ceux installés dans des PC « standard »). La bibliothèque `hardware` qui constitue cette première couche vous est fournie comme point de départ sous la forme des fichiers `hardware.h` et `libhardware.a` ; voir l'encart page suivante. Cette bibliothèque permet d'émuler certains composants matériels d'un ordinateur, comme une carte ethernet ou un disque dur. Nous ne nous occuperons pour ce projet que du disque dur maître.

Le fichier `hardware.h` définit un jeu réduit de fonctions C qui permettent de contrôler, entre autres, l'activité du disque magnétique. Un fichier de configuration paramètre le fonctionnement du matériel émulé. Un tel fichier, nommé `hardware.ini`, vous est fourni avec la bibliothèque `hardware` ; il contient des valeurs par défaut. Ce paramétrage vous permet d'activer ou de désactiver les différents composants matériels émulés, il est impératif, pour ce qui nous concerne ici, que le disque dur maître soit bien activé (la valeur `ENABLE_HDA` doit être positionnée à 1 dans le fichier de configuration).

La bibliothèque `hardware` définit la fonction :

```
int init_hardware(const char *config_file);
```

Cette première fonction permet d'initialiser le matériel émulé (et donc le disque dur) à partir du fichier de configuration dont le chemin d'accès est fourni en paramètre. L'appel à cette fonction met sous tension le disque et calibre position et mouvement de la tête de lecture. Après initialisation, la tête de lecture est placée sur le secteur 0, piste 0.

Si cette phase d'initialisation du matériel n'est pas effectuée, le comportement du disque n'est pas prédictible. Il faut donc appeler cette fonction avant toute autre opération en guise d'initialisation de vos programmes.

Un fichier est créé pour contenir les informations du disque maître. Le nom par défaut de ce fichier est `vdiskA.bin`. Il peut être modifié grâce au fichier de configuration.

La communication avec le matériel (envoi de commandes et de données, envoi/réception de données) se fait via des ports, soit en écriture soit en lecture. Les numéros des ports utilisés pour

Mise en place des travaux pratiques

Vous mettrez en place votre code de la façon suivante :

1. Téléchargez l'archive `pld_fs.tgz` sur moodle
2. Décompressez-la : `tar xzf pld_fs.tgz`
3. Allez dans le dossier `srchardware` : '`cd srchardware`'
4. Compilez le simulateur de matériel. Pour cela, un `Makefile` vous est fourni et vous n'avez donc qu'à taper '`make`'
5. Installez les fichiers nécessaires en tapant '`make install`'. Par défaut, cette installation est faite dans le répertoire `../libhardware`. Dans ce répertoire, on trouvera les répertoires suivants :
   - `lib`, contenant la librairie avec laquelle votre code sera lié ;
   - `etc` contenant le fichier de configuration du disque dur ;
   - `include` contenant les fichiers d'entête.
6. Allez dans le répertoire dans lequel vous implémenterez votre système de fichier : '`cd ../src`'. Le fichier `Makefile` fourni permet de compiler — tapez '`make`' — le fichier `main.c` en le liant avec le simulateur de matériel. Ce fichier contient le code minimal d'initialisation du disque dur, vous n'avez plus qu'à vous en inspirer pour commencer à implémenter ce petit système de fichier.

Notez encore que dans la bibliothèque `hardware` que nous vous fournissons simule le fonctionnement des primitives ATA en utilisant en guise de disque un fichier Unix nommé `vdiskA.bin` (option par défaut pour le disque maître) créé dans le répertoire courant. Si ce fichier n'existe pas notre simulateur de disque le recrée, **dans un état non initialisé**. En supprimant ce fichier vous créez donc un nouveau disque...
En cas de problème de place sur votre compte Unix, vous pouvez modifier le fichier de configuration pour désigner un autre emplacement pour le disque maître. Utilisez par exemple `/tmp/vdiskA.bin` comme valeur. Le disque virtuel sera alors placé dans un répertoire temporaire qui n'imputera pas votre quota disque. Ce répertoire est périodiquement effacé. Pensez dans ce cas à sauvegarder le fichier si vous voulez garder le disque en l'état...

la communication avec le disque dur sont définis eux aussi dans le fichier de configuration. On trouve par exemple dans le fichier `harware.ini` fourni :

```
# Paramètres du controlleur IDE
ENABLE_HDA      = 1             # 0 => simulation du disque désactivée
HDA_CMDREG      = 0x3F6         # registre de commande du disque maitre
HDA_DATAREGS    = 0x110         # base des registres de données (r,r+1...r+15)
HDA_IRQ         = 14            # Interruption du disque

# Paramètres de la simulation
HDA_FILENAME    = "vdiskA.bin" # nom du fichier de stockage du disque simulé
```

On ne peut lire ou écrire qu'un seul octet sur un port. Pour écrire une valeurs codées sur plusieurs octets, on utilise un premier port pour l'octet de poids fort et les ports suivants pour les octets de poids plus faible. De la même manière, si une fonction retourne une valeur de plusieurs octets, l'octet de poids fort sera lu sur le premier port, les octets de poids plus faibles sur les ports suivants.

La fonction

```
int _in(int port);
```

réalise la lecture sur le port désigné. La valeur retournée correspond à l'octet qui a été lu sur ce port. Les numéros de port sont identifiés dans le fichier de configuration du matériel `hardware.ini`. La fonction

```
void _out(int port, int value);
```

réalise l'écriture d'une valeur d'un octet sur le port désigné.

L'envoi de commandes se fait également en écrivant sur le port désigné comme port de commande dans le fichier de configuration. Cela permet au microprocesseur de solliciter une opération

TABLE 5.1 – Commandes ATA-2

Nom	code	port de données (P0 ; P1 ; ... ; P15)	objet
`SEEK`	0x02	numCyl (int16) ; numSec (int16)	déplace la tête de lecture
`READ`	0x04	nbSec (int16)	lit `nbSec` secteurs
`WRITE`	0x06	nbSec (int16)	écrit `nbSec` secteurs
`FORMAT`	0x08	nbSec (int16) ; val (int32)	initialise `nbSec` secteurs avec `val`
`STATUS`	0x12	R.F.U.	R.F.U.
`DMASET`	0x14	R.F.U.	R.F.U.
`DSKNFO`	0x16	nbCyl (int16) ; nbSec (int16) ; tailleSec (int16)	retourne la géométrie d'un disque
`MANUF`	0xA2	Id du fabricant du disque (16 octets)	Identifie le disque
`DIAG`	0xA4	status	diagnostic du disque : 0 = KO / 1 = OK

du disque magnétique. Une fois que le microprocesseur envoie une commande, l'exécution de celle-ci débute immédiatement. Si la commande nécessite des données, il faut obligatoirement les avoir fournies **avant** de déclencher la commande. De la liste des commandes ATA-2 nous avons retenu le sous-ensemble décrit dans la table 5.1. Le fichier `harware.h` définit aussi des macros identifiant ces commandes :

```
#define CMD_SEEK        0x02
#define CMD_READ        0x04
...
```

Une fois une commande fournie au circuit ATA, celui-ci met un certain temps à la réaliser. Si une deuxième commande est passée entre temps, la première commande est « interrompue » laissant le matériel dans un état indéterminé... Pour informer le microprocesseur (et donc le système) de l'état d'avancement d'une commande le circuit ATA génère un signal d'interruption particulier.

Ce signal est émis après que la commande `SEEK` ait atteint la position demandée, ou pour chaque secteur lu (`READ`), écrit (`WRITE`), ou formaté (`FORMAT`). Enfin ce signal est émis après qu'un diagnostic complet ait été accompli, à ce moment seulement la valeur `OK` ou `KO` peut être lue dans le premier registre de données. Pour les autres commandes le résultat est immédiat.

Pour attendre que le disque ait terminé l'exécution de la commande en cours, il faut utiliser la fonction :

```
void _sleep(int irq_level);
```

Le niveau d'IRQ passé en paramètre dépend du matériel visé (ici le disque dur), ce niveau étant défini dans le fichier de configuration.

De plus, un traiteur d'interruption doit être associé à chacune des 16 IRQ (0 à 15). Un traiteur d'interruption est une fonction de type

```
typedef void (*func_irq_t)();
```

Le vecteur `IRQVECTOR[16]` identifie ces fonctions et se doit d'être initialisé. La fonction `IRQVECTOR[n]()` est appelée lorsque l'interruption de niveau `n` est déclenchée par le matériel.

Enfin la valeur `MASTERBUFFER` est un pointeur sur un `unsigned char` qui identifie le tampon exploité par le contrôleur de disque maître. Ce tampon est exploité par les commandes `READ` et `WRITE` pour stocker les données lues/à écrire.

Exercice 21 Afficher un secteur : *dump sector*, `dmps`

Comme premier outil, nous allons concevoir un petit programme qui prend deux arguments en paramètres, un numéro de piste et un numéro de secteur et qui affiche le contenu, octet par octet, du secteur (sous forme hexadécimale par exemple) du disque maître.

Question 21.1 Quelle est la suite de commandes matérielles qu'il faut solliciter pour lire un secteur ? Voici une séquence possible d'opérations :

1. écrire les valeurs requises dans les ports out avec le numéro de piste et le numéro de secteur ;
2. déclencher la commande `SEEK` par écriture dans le port de commande ;
3. attendre le signal d'interruption ; (`_sleep()`) ;
4. écrire dans les ports in le nombre de secteur à lire (1 secteur) ;
5. déclencher la commande `READ` par écriture dans le port de commande ;
6. attendre le signal d'interruption
7. afficher le contenu du tampon `MASTERBUFFER`.

Question 21.2 En supposant que les variables `int cylinder` et `int sector` contiennent respectivement le numéro de piste et de secteur à lire, expliquez les valeurs à écrire dans les ports pour désigner la position que la piste doit atteindre sur le disque. Expliquer comment on réalise la conversion : `int cylinder` vers Registre. dr0 = cylinder hi et dr1 = cylinder low :

- `int` est (en général) 4 octets
- on veut les bits 8 à 15 dans `dr[0]` et les bit 0 à 7 dans `dr[1]`
- donc `dr[0] = (unsigned char) ((cylinder >> 8) & 0xFF)`
- et `dr[1] = (unsigned char) (cylinder & 0xFF)`
- puis `_out(data_port, dr[0]) ; _out(data_port, dr[1])`

On peut faire un petit dessin pour montrer ce qu'on veut faire (décalage, masque, cast unsigned char), rappeler que `0xFF` c'est 8 bits à 1, etc.

Question 21.3 Réalisez le programme `dmps`. Voici la fonction `dump_sector()` du fichier `dmps.c`. (La variable `octal_dump` (resp. `ascii_dump`) est à vrai ssi on veut un dump octal (resp. ascii). Plutôt leur donner le code d'affichage que de les laisser patiner.

```
static void
dump_sector(unsigned int cylinder, unsigned int sector)
{
    int data_port = MASTER_DATA_PORT;
    int cmd_port = MASTER_CMD_PORT;
    int sector_size = get_sector_size();
    int irq = MASTER_IRQ;
    unsigned char *buffer = MASTERBUFFER;
    unsigned int i;

    /* Numéro de cylindre */
    _out(data_port, (cylinder >> 8) & 0xFF);
    _out(data_port+1, cylinder & 0xFF);
    /* Numéro de secteur */
    _out(data_port+2, (sector >> 8) & 0xFF);
    _out(data_port+3, sector & 0xFF);
    /* Demande au disque de se déplacer et attend la fin de l'exécution */
    _out(cmd_port, CMD_SEEK);
    _sleep(irq);

    /* Nombre de secteurs à lire (un seul) */
    _out(data_port, 0);
    _out(data_port+1, 1);
    /* Demande de lecture et attente */
    _out(cmd_port, CMD_READ);
    _sleep(irq);

    /* dump buffer */
```

```
    for (i=0; i<sector_size; i+=16) {
        /* offset */
        printf("%.8o",i);

        /* octal dump */
        if (octal_dump) {
            for(j=0; j<8; j++)
                printf(" %.2x", buffer[i+j]);
            printf(" - ");

            for( ; j<16; j++)
                printf(" %.2x", buffer[i+j]);

            printf("\n");
        }
        /* ascii dump */
        if (ascii_dump) {
            printf("%8c", ' ');

            for(j=0; j<8; j++)
                printf(" %1c ", isprint(buffer[i+j])?buffer[i+j]:' ');
            printf(" - ");

            for( ; j<16; j++)
                printf(" %1c ", isprint(buffer[i+j])?buffer[i+j]:' ');

            printf("\n");
        }
    }
}
```

On suppose que les macros `MASTER_DATA_PORT`, `MASTER_CMD_PORT` et cie sont définies par l'utilisateur (l'étudiant !) dans un fichier (mettons `hw.h`) qui est conservé cohérent avec les valeurs renseignées dans `hardware.ini`. Extrait de `hw.h` :

```
/* les deux disques */
#define DISK_MASTER             1
#define DISK_SLAVE              2

/* les ports */
#define MASTER_DATA_PORT        0x110
#define MASTER_CMD_PORT         0x3F6

#define SLAVE_DATA_PORT         0x376
#define SLAVE_CMD_PORT          0x170
```

On peut indiquer d'utiliser `long strtoul(char *string, NULL, 10)` de la librairie `string.h` pour obtenir une représentation entière à partir des chaînes de caractères arguments de la commande `dmps`.

Il n'est pas raisonnable que ce programme soit écrit tel que. Plutôt écrire une première petite bibliothèque `drive` de la question 5.1 et la commande `dmps` au dessus ; voir l'encart page 39.

Exercice 22 Formater un disque : `frmt`

On se propose maintenant d'écrire un programme qui détruit entièrement le contenu d'un disque physique en formatant chaque secteur du disque. Proposez un tel programme. Pas de problèmes

particuliers sur cette question. On a choisi ici de formater "piste par piste" mais on pourrait lancer un "grand format" (une commande pour tous les secteurs d'une piste) et attendre le bon nombre d'interruptions.

```
void
format()
{
    int data_port = MASTER_DATA_PORT;
    int cmd_port = MASTER_CMD_PORT;
    int nb_cylinders = get_nb_cylinders();
    int nb_sectors = get_nb_sectors();
    int irq = MASTER_IRQ;
    unsigned int i, j;

    for(i=0; i<nb_cylinders; ++i) {
        for(j=0; j<nb_sectors; ++j) {
            /* Cylindre */
            _out(data_port, (i >> 8) & 0xFF);
            _out(data_port+1, i & 0xFF);
            /* Secteur */
            _out(data_port+2, (j >> 8) & 0xFF);
            _out(data_port+3, j & 0xFF);
            /* Seek */
            _out(cmd_port, CMD_SEEK);
            _sleep(irq);

            /* Nombre de secteurs (1 ici) */
            _out(data_port, 0);
            _out(data_port+1, 1);
            /* Formate avec la valeur 0 (int32) */
            _out(data_port+2, 0);
            _out(data_port+3, 0);
            _out(data_port+4, 0);
            _out(data_port+5, 0);

            /* Format */
            _out(cmd_port, CMD_FORMAT);
            _sleep(irq);
        }
    }

    /* done */
    return RETURN_SUCCESS;
}
```

Exercice 23 Une bibliothèque pour l'accès physique : `drive`

Pour pouvoir simplifier notre tâche dans les développements à suivre, nous nous proposons d'écrire une première série de fonctions utilitaires qui formerons notre bibliothèque `drive` d'accès au périphérique.

```
void read_sector(unsigned int cylinder, unsigned int sector,
                 unsigned char *buffer);
void write_sector(unsigned int cylinder, unsigned int sector,
```

Première étape des travaux pratiques — Validation de la bibliothèque d'accès au matériel

Une validation *minimale* de la bibliothèque `drive` peut être obtenue en écrivant les commandes `dmps` et `frmt` au dessus de la bibliothèque. Observer attentivement le résultat de commandes telles les suivantes :
- création d'un disque (`mkhd`) ;
- visualisation d'un secteur quelconque (`dmps`, en particulier, pensez à tester les valeurs extrèmes des paramètres `cylinder` et `sector` des fonctions `read_sector()` et `write_sector()`) ;
- comparaison avec le contenu du fichier Unix `vdiskA.bin` (ou `vdiskB.bin` pour le disque esclave) dans lequel est simulé le disque ATA (commande Unix `od -x`) ;
- formatage du disque (`frmt`) ;
- visualisation d'un secteur quelconque (`dmps`).

Vous pouvez penser à comparer les résultats de votre commande avec ceux produits *sur le même disque* par la commande de vos camarades.
Pour tester l'écriture sur le disque, il est *nécessaire* de développer un programme ad hoc.
Deux remarques :

1. **Il est illusoire de poursuivre les développements qui s'appuieront sur cette bibliothèque `drive` sans que celle-ci ait été validée.**
2. **Il vous faudra rendre ou démontrer vos programmes de tests lors de l'évaluation de votre travail.**

```
                    const unsigned char *buffer);
void format_sector(unsigned int cylinder, unsigned int sector,
                   unsigned int nsector,
                   unsigned int value);
```

Il s'agit de la première couche, `drive`. Voir le fichier `drive.c`. Il n'y a rien de particulier à dire sur ces fonctions extrêmement simples à écrire une fois les questions précédentes traitées.

On peut passer par des fonctions utilitaires intermédiaires ; par exemple

```
static void goto_sector(unsigned int cylinder, unsigned int sector);
```

C'est aussi à ce niveau que l'on peut ajouter des fonctions comme

```
unsigned int get_sector_size(unsigned int atadev);
```

qui sera utile par la suite. Pour récupérer cette taille, on peut ne pas réaliser un accès systématique au matériel, mais définir une valeur

```
#define SECTOR_SIZE  1024
```

et vérifier au démarrage que cette valeur est bien cohérente avec le matériel. La connaissance statique de cette taille `SECTOR_SIZE` permet aussi des allocations statiques ou automatiques de buffers.

L'obtention de cette bibliothèque `drive` et de quelques tests, par exemple sous la forme d'une réécriture des commandes `dmps` et `frmt` au dessus de la bibliothèque doit être l'objectif de la première séance ; voir l'encart.

5.2 Seconde couche logicielle : gestion de volumes

D'un point de vue logique, les secteurs d'un disque sont regroupés pour former un « volume ». Un volume peut correspondre à l'ensemble des secteurs d'un disque donné, mais il arrive qu'un disque soit décomposé en plusieurs volumes distincts, chaque volume représentant une partie de la surface du disque.

Pour définir la place de chaque volume (ou partition) sur le disque on structure le disque. Le premier secteur du disque correspond au « master boot record », MBR. Nous convenons ici que ce premier secteur définit le nombre de volumes présents sur le disque, la position (en coordonnée piste/secteur) du premier secteur de chaque volume, ainsi que le nombre de secteurs consécutifs associés au volume. On conviendra d'un maximum de 8 volumes présents sur un disque. De plus une information associée à chaque volume permettra de déterminer si le volume est :

– le volume de base pour le système de fichiers ;
– un volume annexe du système de fichiers ;
– un autre type de volume qui ne peut être associé au système de fichiers.

Exercice 24 Secteur d'amorce primaire

Proposez une structure de donnée pour stocker les informations inscrites dans le MBR. Toutes les infos nécessaires sont dans les paragraphes précédents ! Le code extrait de `mbr.h` :

```
#define MAX_VOL         8

enum vol_type_e {base, annex, other};

struct vol_descr_s {
    unsigned int vol_first_cylinder;
    unsigned int vol_first_sector;
    unsigned int vol_n_bloc;
    enum vol_type_e vol_type;
};

struct mbr_descr_s {
    unsigned int mbr_magic; /* MBR_MAGIC */
    unsigned int mbr_n_vol;
    struct vol_descr_s mbr_vol[MAX_VOL]; /* first mbr_n_vol are in used */
};

extern struct mbr_descr_s mbr;
```

Il est à remarquer que la taille d'une structure de type `struct mbr_descr_s` est/doit être inférieure à la taille d'un secteur (512 octets, définie par `SECTOR_SIZE` de `hardware.h`). Ceci est vérifié par la fonction suivante de `mbr.c` :

```
static void
compatible_mbr ()
{
    fatal(SECTOR_SIZE >= sizeof(struct mbr_descr_s),
          "compatible_mbr",
          "sector and mbr of incompatible size");
}
```

On note aussi la *déclaration* de la variable globale `mbr` qui sera utilisée par les fonctions suivantes. La *définition* se trouve dans `mbr.c` (une explication n'est pas superflue !) :

```
#include "mbr.h"

struct mbr_descr_s mbr;
```

Exercice 25 Initialisation des volumes

Proposez une fonction C qui lit le MBR est initialise, avec le résultat de cette lecture, une structure de donnée globale accessible par toutes autres procédures. Cette structure de donnée sera gardée en mémoire durant toute l'utilisation du disque.

Proposez de même une fonction de sauvegarde de la structure de données vers le MBR qui sera appelée en fin d'utilisation du disque. Fonctions très simples à écrire en réutilisant les `read_sector()` et `write_sector()`. Par contre, il faut traiter le cas ou le mot magique n'est pas correct. On propose d'initialiser la variable `mbr` avec le bon mot magique et un nombre de volumes égal à zéro.

```
int
load_mbr()
{
    unsigned char sbuf[SECTOR_SIZE];

    /* check sector size */
    compatible_mbr();

    /* read cyclinder 0, sector 0 to fill mbr */
    read_sector(HD_MASTER, 0, 0, sbuf);
    memcpy(&mbr, sbuf, sizeof (struct mbr_descr_s));

    /* if no magic, assume a new disk, and return RETURN_FAILURE */
    if (mbr.mbr_magic != MBR_MAGIC) {
        mbr.mbr_magic = MBR_MAGIC;
        mbr.mbr_n_vol = 0;
        return RETURN_FAILURE;
    }

    return RETURN_SUCCESS;
}
```

On peut aussi leur expliquer que l'on peut ne pas utiliser `memcpy()` et faire

```
mbr = *((struct mbr_descr_s *)sbuf);
```

Exercice 26 Conversion d'adressage

Un bloc est un secteur du disque qui est associé à un volume. Les blocs d'un volume sont contigus.

Aussi un bloc est identifié par un simple numéro de bloc relatif au volume. Proposez une formule de conversion qui permette de transformer un couple (numéro de volume, numéro de bloc) en un couple (numéro de cylindre, numéro de secteur). En dessinant une grille à deux dimensions (secteur et piste) et en montrant comment évoluent les numéros de piste et de secteur au fur et à mesure qu'on incrémente les numéros de blocs, les étudiants devinent assez rapidement et on écrit les deux fonctions de `vol.c` :

```
unsigned int
cylinder_of_bloc(unsigned int vol, unsigned int nbloc)
{
    fatal(vol < mbr.mbr_n_vol, "cylinder_of_bloc",
          "volume out of range (may be an uninitialized disk)");

    fatal(nbloc < mbr.mbr_vol[vol].vol_n_bloc, "cylinder_of_bloc",
          "bloc out of range");

    return (mbr.mbr_vol[vol].vol_first_cylinder
            + ((mbr.mbr_vol[vol].vol_first_sector + nbloc)
               / max_sector(drive_of_vol(vol))));
}

unsigned int
sector_of_bloc(unsigned int vol, unsigned int nbloc)
{
    fatal(vol < mbr.mbr_n_vol, "sector_of_bloc",
          "volume out of range (may be an uninitialized disk)");
```

```
    fatal(nbloc < mbr.mbr_vol[vol].vol_n_bloc, "sector_of_bloc",
          "bloc out of range");

    return ((mbr.mbr_vol[vol].vol_first_sector + nbloc)
            % max_sector(drive_of_vol(vol)));
}
```

On remarque que

- on utilise directement la structure `mbr` *qui est en mémoire* ;
- on utilise la fonction `max_sector()` fournie par `drive.h`

Exercice 27 Bibliothèque d'accès aux volumes : `vol`

Pour pouvoir utiliser l'organisation en volumes du disque, nous nous proposons de réaliser un ensemble de fonctions qui permettront de lire, écrire ou formater des blocs :

```
void read_bloc(unsigned int vol, unsigned int nbloc,
               unsigned char *buffer);
void write_bloc(unsigned int vol, unsigned int nbloc,
                const unsigned char *buffer);
void format_vol(unsigned int vol);
```

Notez que l'utilisation de ces fonctions suppose que le disque ait été initialisé et que le MBR ait été lu en mémoire. Une ligne de code par fonction :

```
void
read_bloc(unsigned int vol, unsigned int nbloc, unsigned char *buffer)
{
    read_sector(cylinder_of_bloc(vol, nbloc),
                sector_of_bloc(vol, nbloc),
                buffer);
}

void
write_bloc(unsigned int vol, unsigned int nbloc, const unsigned char *buffer)
{
    write_sector(cylinder_of_bloc(vol, nbloc),
                 sector_of_bloc(vol, nbloc),
                 buffer);
}

void
format_vol(unsigned int vol)
{
    format_sector(cylinder_of_bloc(vol, 0),
                  sector_of_bloc(vol,0),
                  mbr.mbr_vol[vol].vol_n_bloc,
                  0);
}
```

Exercice 28 Gestionnaire de partitions

Il s'agit de réaliser un petit programme qui permette de lister les partitions présentes sur un disque, de créer une nouvelle partition, de supprimer une partition... Voir l'encart page suivante à ce propos. On peut discuter de

Squelette d'un gestionnaire de partitions

L'archive

`vm-skel.tgz`

disponible sur moodle contient le squelette d'un gestionnaire de volumes. Vous pouvez l'utiliser comme point de départ de votre développement d'un gestionnaire de partitions. En particulier cela vous décharge complètement de la gestion de l'interaction avec l'utilisateur.
Il vous faut copier le fichier `vm-skel.c` pour créer un fichier `vm.c` que vous compléterez. Un `Makefile` vous est fourni pour construire les différents exécutables.
Pour faciliter les choses, on peut introduire dans la bibliothèque `drive` d'accès au matériel une fonction `init_master()` qui initialisera le disque maître utilisé. Dans la suite des développements, nous ne nous préoccuperons plus du disque esclave.
La fonction principale de notre gestionnaire de partitions se doit de faire appel aux fonctions d'initialisation de ce disque et se doit de charger le MBR en mémoire :

```
/* init master drive and load MBR */
init_master();
load_mbr();
```

Validation de la bibliothèque de gestion de volumes

Comme précédemment, il s'agit de valider votre travail avant de poursuivre les développements.
Le protocole suivant vérifie un minimum de cohérence dans votre implantation :
– créez une partition `P1` encadrée d'une partition `P0` et d'une partition `P2` ;
– listez l'état de votre disque ;
– détruisez la partition `P1` ;
– listez l'état de votre disque ;
– recréez la partition `P1` ;
– listez à nouveau l'état de votre disque.
En particulier, vous pouvez ou non quitter et relancer le gestionnaire de volume entre chaque étape.

  - la manière d'afficher la géométrie du disque, les informations du MBR, les informations suivantes pour chacune des partitions :
    - numéro de partition,
    - origine (cyl, sect),
    - nombre de blocs,
    - type du volume,
    - *coordonnes de fin de partition (cyl, sect)*

    En particulier cette dernière information oblige les étudiants à/permet de tester les fonctions `cylinder_of_bloc()` et `sector_of_bloc()` de la bibliothèque de la question précédente ;
  - comment ajouter un volume. Quels sont les tests à effectuer ?
  - comment supprimer un volume ?

On peut mentionner qu'il ne faut pas oublier de faire un `save_mbr()` une fois `mbr` modifié.

Si un gestionnaire de partition ne peut être validé, pour continuer les TPs, les étudiants doivent au minimum produire une commande ad hoc qui initialise un disque, initialise un MBR, et initialise une première partition.

5.3 Troisième couche logicielle, 1re partie : structure d'un volume

Chaque volume manipulé par notre système de fichiers dispose d'un descripteur de volume, son superbloc, inspiré par celui décrit dans le cours. Il dispose à la base

– d'un mot magique en guise de détrompeur ;
– d'un numéro de série ;
– d'un nom composé au maximum de 32 caractères ;
– d'un identifiant qui donne le lien vers le premier inœud (associé au fichier de base du disque : son répertoire racine).

De plus le système de fichiers utilise un mécanisme de gestion des blocs libres gérés sous forme d'une liste chaînée de blocs.

Exercice 29 Descripteurs de volume

Définissez les structures de données du superbloc de chaque volume, du chaînage des blocs libres.

Extrait de `super.h`, pour le superbloc :

```
struct super_s {
    unsigned int super_magic; /* SUPER_MAGIC */
    unsigned int super_root;
    unsigned int super_free_size; /* number of blocs */
    unsigned int super_first_free;
    char super_name[32];
    unsigned int super_serial;
};
```

On peut discuter de la pertinence du champs `super_free_size` en argumentant sur la complexité algorithmique pour calculer cette information à partir d'une liste chaînée de blocs libres, alors que lorsqu'on met à jour le `super_first_free`, il est facile de mettre à jour l'information d'espace disponible en même temps.

À propos des deux champs `super_name` et `super_serial`, on peut glisser un mot en TD sur les images « ghost » de disque et sur l'intérêt de rendre unique chaque volume. `serial` est supposé être choisi de manière univoque par celui qui initialise le volume, en générant par exemple un nombre aléatoire pour différencier chaque volume dans un parc informatique conséquent.

Pour le chaînage des blocs libres, caché dans `super.c` (et non `super.h`, seuls `new_bloc()` et `free_bloc()` utilisant cette structure) :

```
struct free_bloc_s {
    unsigned int fb_n_blocs;
    unsigned int fb_next;
};
```

Exercice 30 Initialiser un volume

Proposez une fonction

```
void init_super(unsigned int vol);
```

qui permet d'initialiser le superbloc d'un volume, en particulier d'y associer le chaînage des blocs libres. Il s'agit de construire un superbloc initial

– ce sera par convention le bloc numéro 0 :

```
#define SUPER           0               /* bloc # of the superbloc */
```

– les autres blocs du volumes sont libres, un seul descripteur de blocs libres qui sera le bloc numéro 1 (`SUPER + 1`).

```
void
init_super(unsigned int vol, const char *name, unsigned int serial)
{
    struct super_s super;
```

```
    struct free_bloc_s fb;

    unsigned int free_size;
    unsigned char buf[BLOC_SIZE];

    /* initialize the superbloc */
    super.super_magic = SUPER_MAGIC;

    /* all blocs but the superbloc are free */
    free_size = mbr.mbr_vol[vol].vol_n_bloc - 1;
    super.super_free_size = free_size;

    /* the first free bloc is the bloc #1 */
    super.super_first_free = 1;

    /* a non significant value for the root file */
    super.super_root = 0;

    /* super name and serial */
    strncpy(super.super_name, name, 32);
    super.super_name[32] = 0;
    super.super_serial =  serial;

    /* write the super bloc */
    *((struct super_s *) buf) = super;
    write_bloc(vol, SUPER, buf);

    /* initialize the first and sole pack of free blocs */
    fb.fb_n_blocs = free_size;
    fb.fb_next = 0;

    /* write this free bloc */
    *((struct free_bloc_s *) buf) = fb;
    write_bloc(vol, SUPER+1, buf);
}
```

Exercice 31 Sélection et mise à jour d'un volume

Il s'agit de réaliser les deux fonctions suivantes :

```
int load_super(unsigned int vol);
void save_super();
```

qui permettent respectivement de sélectionner le volume courant en chargeant le superbloc dans une variable globale et de mettre à jour le superbloc chargé en mémoire, après qu'il ait été modifié. `load_super()` retourne une erreur ssi le mot magique ne correspond pas.

On a donc une variable globale pour le superbloc courant, mais aussi un numéro de volume courant ; par exemple pour pouvoir réécrire ce superbloc :

```
unsigned int current_volume;
struct super_s super;
```

Les fonctions de manipulation de fichiers n'auront ainsi plus à spécifier le volume utilisé. Voir l'encart page 48. C'est ensuite trivial :

```
int
load_super(unsigned int vol)
{
    unsigned char buf[BLOC_SIZE];

    /* update current_volume */
    current_volume = vol;

    /* fill super */
    read_bloc(vol, SUPER, buf);
    super = *((struct super_s *)buf);

    /* valid superbloc? */
    if (super.super_magic == SUPER_MAGIC)
        return RETURN_SUCCESS;
    else
        return RETURN_FAILURE;
}

void
save_super()
{
    unsigned char buf[BLOC_SIZE];

    *((struct super_s *)buf) = super;
    write_bloc(current_volume, SUPER, buf);
}
```

Exercice 32 Allouer et libérer des blocs

Proposer des fonctions de gestion des blocs pour allouer et libérer des blocs dans un volume :

```
unsigned int new_bloc();
void free_bloc(unsigned int bloc);
```

- `new_bloc()` et `free_bloc()` travaillent sur le volume courant ;
- `new_bloc()` retourne un numéro de bloc, `BLOC_NULL` l'allocation n'est pas possible ;
- on a ici un seul allocateur pour tous les blocs (de données, d'indirection, d'inœud) ;

```
unsigned int
new_bloc()
{
    unsigned int newb; /* the new bloc number */
    struct free_bloc_s fb;
    unsigned char buf[BLOC_SIZE];

    /* ensure there is space! */
    if (super.super_free_size == 0)
        return BLOC_NULL;

    /* read the first free bloc */
    read_bloc(current_volume, super.super_first_free, buf);
    fb = *((struct free_bloc_s *)buf);

    /* one bloc less inthe file system */
```

```
    super.super_free_size--;

    if (fb.fb_n_blocs == 0) {
        /* it was the last bloc of fb, fb is no more a free bloc */
        newb = super.super_first_free;
        super.super_first_free = fb.fb_next;
    } else {
#ifdef _STUPID_
        /* return the last free bloc of fb */
        newb = super.super_first_free + fb.fb_n_blocs - 1;
        /* update the number of free blocs */
        fb.fb_n_blocs--;
        /* and copy fb back */
        *((struct free_bloc_s *)buf) = fb;
        write_bloc(current_volume, super.super_first_free, buf);
#else
        /* return the first free bloc of fb */
        newb = super.super_first_free;
        /* update the number of free blocs */
        fb.fb_n_blocs--;
        /* update the ref to the first free bloc in super */
        super.super_first_free++;
        /* copy fb to this new bloc */
        *((struct free_bloc_s *)buf) = fb;
        write_bloc(current_volume, super.super_first_free, buf);
#endif
    }

    return newb;
}
```

La libération avec insertion en tête, on ne cherche pas à « recoller » les blocs libres contigus :

```
void
free_bloc(unsigned int bloc)
{
    struct free_bloc_s fb;
    unsigned char buf[BLOC_SIZE];

    /* 0 is not a valid bloc number */
    fatal(bloc != BLOC_NULL, "free_bloc", "attempt to free bloc 0");

    /* the bloc is free, you know.
       It will be the first free bloc.
       No vectorisation of free blocs */
    fb.fb_n_blocs = 1 ;
    fb. fb_next = super.super_first_free;

    /* write the bloc */
    *((struct free_bloc_s *)buf) = fb;
    write_bloc (current_volume, bloc, buf);

    /* super now references this new free bloc */
    super.super_first_free = bloc;
```

Partition courante — Commandes *mknfs* et *dfs*

Dans l'ensemble des programmes qui seront maintenant développés, nous travaillerons sur un unique volume désigné par la valeur de la variable d'environnement `CURRENT_VOLUME`.
Dans le même esprit, le fichier de configuration de la bibliothèque `hardware` utilisé sera désigné par la variable d'environnement `HW_CONFIG`. Une valeur par défaut pourra être choisie.
On développera en particulier un programme `mknfs` (*make new filesystem*), pendant de la commande Unix `mkfs`, qui initialisera ce volume courant et un programme `dfs` (*display filesystem*), pendant de la commande Unix `df`, qui affichera l'état des partitions, et pour la partition courante sont taux d'occupation.

Validation de la bibliothèque d'allocation/libération de blocs

Pour valider votre travail, concevez un programme qui
– fait appel à la fonction `new_bloc()` jusqu'à ce qu'elle retourne une erreur ;
– vérifie que le disque est plein ;
– itère un nombre aléatoire de fois sur la libération d'un bloc `free_bloc()` ;
– affiche le statut du disque (taille libre) ;
– alloue des blocs tant que le disque est non plein et retourne le nombre de blocs ayant pu être alloués.

```
        /* one more free bloc */
        super.super_free_size++;
    }
```

5.4 Troisième couche logicielle, 2e partie : structure d'un fichier

Chaque fichier ou répertoire est défini par un inœud, tel que le cours le présente, avec un type, une taille de fichier en octets, et les tables de numéro de blocs, « direct », « indirect » et « double indirect ».

Exercice 33 Structure d'un inœud

Définissez la structure `struct inode_s`qui sera utilisée pour représenter un inœud. On trouve dans `inode.h`

```
enum file_type_e {ordinary, directory, special};

struct inode_s {
    enum file_type_e ind_type;
    unsigned int ind_size; /* in char */
#define N_DIRECT_BLOCS ((BLOC_SIZE/sizeof(int))-4)
    unsigned int ind_direct[N_DIRECT_BLOCS];
    unsigned int ind_indirect;
    unsigned int ind_d_indirect;
};
```

On remarque que l'on a un (seul) inœud par secteur (mais c'est inhabituel). On met donc autant de numéros de bloc direct que possible. Attention, il faudra des fichiers de « grande » taille pour tester les accès aux blocs indirects et doubles indirects.

On définit aussi

```
/* a bloc full of zeros */
#define BLOC_NULL 0
```

pour identifier le 0 comme numéro de bloc, par exemple dans les champs `ind_direct`, `ind_indirect`... on en reparlera.

Les inœuds sont bien entendu enregistrés sur le disque. Nous choisissons d'enregistrer un unique inœud par bloc. (Cela est inhabituel, la taille d'un inœud étant petite devant celle d'un bloc.) Un inœud est identifié par son inombre qui ne sera dans notre cas rien d'autre qu'un numéro de bloc.

Nous définissons deux fonctions utilitaires pour réaliser l'écriture et la lecture sur le disque d'un inœud.

```
void read_inode(unsigned int inumber, struct inode_s *inode);
void write_inode(unsigned int inumber, struct inode_s *inode);
```

ainsi que deux fonctions de création et destruction d'un inœud :

```
unsigned int create_inode(enum file_type_e type);
int delete_inode(unsigned int inumber);
```

La fonction `create_inode`est chargé de l'allocation d'un bloc pour y ranger l'inœud, et de l'initialisation de celui-ci. Elle retourne le inombre correspondant.

La fonction `delete_inode`libère l'ensemble des blocs de données associés à l'inœud puis le bloc de l'inœud lui-même.

Exercice 34 Lecture et écriture d'inœuds

Donnez une implémentation des fonctions `read_inode`et `write_inode`. C'est immédiat : de simples appels à `read_bloc`et `write_bloc` !

```
void
read_inode (unsigned int inumber, struct inode_s *inode)
{
    read_bloc(current_volume, inumber, (unsigned char *)inode);
}

void
write_inode (unsigned int inumber, struct inode_s *inode)
{
    write_bloc(current_volume, inumber, (unsigned char *)inode);
}
```

On peut imaginer ajouter un détrompeur et des valeurs de retour indiquant une erreur...

Exercice 35 Création et suppression d'un inœud

Donnez une implémentation des fonctions de création et destruction d'un inœud : `create_inode`et `delete_inode`.

`create_inode`

- initialise un inœud : type fourni en paramètre, taille 0, direct[] 0, indirect 0, double indirect 0. (taille 0, type 1, direct[x] = 0, indirect = 0 indirect2 = 0) ;
- alloue un bloc et y copie l'inœud ;
- écrit le bloc ;
- retourne le inombre.

On peut expliquer qu'il est nécessaire de mettre des zéros dans les trois champs direct, indirect... et parler des fichiers « creux » : les indirections vers le bloc zéro (`BLOC_NULL`) sont des indirections vers des pages pleines de zéros qu'il n'est pas nécessaire d'écrire sur le disque.

Le code

```
unsigned int
create_inode(enum file_type_e type)
{
    struct inode_s inode;
    unsigned int bloc;

    /* fill the inode */
    inode.ind_type = type;
    inode.ind_size = 0;
    memset(inode.ind_direct, 0, N_DIRECT_BLOCS * sizeof(unsigned int));
    inode.ind_indirect = 0;
    inode.ind_d_indirect = 0;

    /* allocate a bloc for the inode */
    bloc = new_bloc();
    if (!bloc)
        return RETURN_FAILURE;

    /* copy the inode to the bloc */
    write_inode(bloc, &inode);

    /* done */
    return bloc;
}
```

`delete_ifile`libère non seulement l'inœud, mais aussi tous les blocs associés :

1. lire l'inœud pour pouvoir y accéder !
2. libération des blocs directs.
3. libération des blocs indirects.
4. libération des blocs double indirects.
5. libération de l'inœud.

une libération = `free_bloc`. Attention à ne pas faire de `free_bloc(BLOC_NULL)`.

Quelques outils (`inode.h` et `inode.c`) :

```
/* number of bloc numbers in a bloc */
#define NBLOC_PER_BLOC ((BLOC_SIZE/sizeof(int)))

/* free all the blocs of the given array of bloc indexes */
static void
free_blocs(unsigned int blocs[], unsigned int size)
{
    unsigned int i;

    for (i=0 ; i<size ; i++)
        if (blocs[i])
            free_bloc(blocs[i]);
}
```

et le code :

```
int
delete_inode(unsigned inumber)
{
    struct inode_s inode;
    unsigned int blocs[NBLOC_PER_BLOC];
```

```
    unsigned int bblocs[NBLOC_PER_BLOC];
    unsigned char buf[BLOC_SIZE];
    unsigned int i;

    /* load the inode */
    read_inode(inumber, &inode);

    /* free direct blocs */
    free_blocs(inode.ind_direct, N_DIRECT_BLOCS);

    /* free indirect blocs */
    if (inode.ind_indirect) {
        /* load the indirect bloc */
        read_bloc(inode.ind_indirect, buf);
        memcpy(blocs, buf, BLOC_SIZE);

        /* free indirect blocs */
        free_blocs(blocs, NBLOC_PER_BLOC);

        /* free the indirect bloc */
        free_bloc(inode.ind_indirect);
    }

    /* free dbl indirect blocs */
    if (inode.ind_d_indirect) {
        /* load the dbl indirect bloc */
        read_bloc(inode.ind_d_indirect, buf);
        memcpy(bblocs, buf, BLOC_SIZE);

        /* iterate throuch the dbl indirect bloc */
        for (i=0 ; i < NBLOC_PER_BLOC ; i++) {
            if (bblocs[i]) {
                /* load the indirect blocs */
                read_bloc(bblocs[i], buf);
                memcpy(blocs, buf, BLOC_SIZE);

                /* free the leaves */
                free_blocs(blocs, NBLOC_PER_BLOC);

                /* free the indirect bloc */
                free_bloc(bblocs[i]);
            }
        }

        /* free the dbl indirect bloc */
        free_bloc(inode.ind_d_indirect);
    }

    /* free the inode bloc */
    free_bloc(inumber);

    /* done */
    return RETURN_SUCCESS;
}
```

Afin de développer la couche logicielle supérieure, nous allons présenter un fichier sous la forme d'une suite continue de blocs de données. Il est donc nécessaire de convertir un indice de bloc d'un inœud en un numéro de bloc dans le volume. Une telle conversion repose sur le parcours de la table des numéros de blocs directs, indirects et double indirects de l'inœud.

Exercice 36 Bloc volume d'un bloc inœud

Développez une fonction

```
unsigned int vbloc_of_fbloc(unsigned int inumber,
                            unsigned int fbloc);
```

qui retourne le numéro de bloc sur le volume qui correspond au `fbloc`-ième bloc de l'inœud. Dans un premier temps, cette fonction retourne une valeur nulle si le bloc n'a pas été alloué. On note qu'une valeur de numéro de bloc nulle (bloc non alloué) est considéré comme désignant un bloc plein de zéros, et ce à tout niveau : blocs de données, bloc d'indirection, etc.

Pour écrire `vbloc_of_fbloc`, il s'agit de différencier les différents cas :

– on est dans un bloc direct, indirect, double indirect.

Un bon moyen de ne pas se mélanger dans les différentes plages de numéros de bloc est de changer l'origine (les decrémentations de `fbloc`) dans le code suivant (`inode.c`) :

```
unsigned int
vbloc_of_fbloc(unsigned int inumber, unsigned int fbloc)
{
    struct inode_s inode;

    /* read the inode */
    read_inode(inumber, &inode);

    /* a direct bloc? */
    if (fbloc < N_DIRECT_BLOCS)
        return inode.ind_direct[fbloc];

    /* an indirect bloc? */
    fbloc -= N_DIRECT_BLOCS;

    if (fbloc < NBLOC_PER_BLOC) {
        unsigned int blocs[NBLOC_PER_BLOC];

        if (! inode.ind_indirect)
            return BLOC_NULL;

        /* read the indirect bloc */
        read_bloc(current_volume, inode.ind_indirect, (unsigned char*)&blocs);

        /* return the right number */
        return blocs[fbloc];
    }

    /* a dbl indirect bloc? */
    fbloc -= NBLOC_PER_BLOC;

    if (fbloc < NBLOC_PER_BLOC * NBLOC_PER_BLOC) {
        unsigned int blocs[NBLOC_PER_BLOC];
        unsigned int blevel1 = fbloc / NBLOC_PER_BLOC;
```

```
        unsigned int blevel2 = fbloc % NBLOC_PER_BLOC;

        if (! inode.ind_d_indirect)
            return BLOC_NULL;

        /* read the dbl indirect bloc */
        read_bloc(current_volume, inode.ind_d_indirect,
                  (unsigned char*)&blocs);

        if (! blocs[blevel1])
            return BLOC_NULL;

        /* read the second indirection bloc */
        read_bloc(current_volume, blocs[blevel1], (unsigned char*)&blocs);

        return blocs[blevel2];
    }

    /* access behond limit */
    return fatal(0==1, "vbloc_of_fbloc", "file access out of range");
}
```

On peut faire remarquer que `vbloc_of_fbloc()` commence par lire un inœud qui est certainement déjà en mémoire dans la fonction appelante ; mais une gestion d'un cache d'inœuds permet cela sans surcoût...

Cette fonction `vbloc_of_fbloc`est utilisée pour accéder en lecture à un bloc de données correspondant à un inœud. Si l'on désire accéder en écriture à un tel bloc, il est nécessaire d'allouer ce bloc et de le rattacher à la structure des blocs directs, indirects, double indirects de l'iœud.

Nous ajoutons un paramètre booléen à la fonction `vbloc_of_fbloc`qui indique si le comportement de la fonction doit être de retourner une valeur nulle ou d'allouer le bloc en cas de bloc non encore existant.

Exercice 37 Allocatation d'un bloc d'inœud

Développez la nouvelle version de la fonction

```
unsigned int vbloc_of_fbloc(unsigned int inumber,
                            unsigned int bloc,
                            bool_t do_allocate);
```

qui retourne le numéro du bloc dans le volume qui correspond au `fbloc`-ième bloc de l'inœud. Si ce bloc n'est pas alloué et `do_allocate`, la fonction se charge de l'allocation du bloc et de le connecter à la structure de l'inoœud. Une valeur nulle sera retournée uniquement en cas d'erreur : erreur d'allocation de bloc ; disque plein.

On reprend le schéma précédent de la fonction. Avant chaque retour d'un numéro de bloc dans la solution précédente, on réalise une allocation et continue. Par exemple

```
    /* a direct bloc? */
    if (fbloc < N_DIRECT_BLOCS)
return inode.ind_direct[fbloc];
```

est remplacé par

```
    /* a direct bloc? */
    if (bloc < N_DIRECT_BLOCS) {
/* not already allocate? */
if (do_allocate && ! inode.ind_direct[bloc]) {
```

Validation de la troisième couche logicielle : structure d'un volume et d'un fichier

Vous pourrez valider votre bibliothèque de gestion d'un volume et de gestion d'inœuds à l'aide de l'archive que nous vous fournissons et qui permet de produire les commandes listées dans l'encart ≪ Validation de la bibliothèque de manipulation de fichier par leur inombre ≫ page 59. Voir

`ifile.tgz`

.

```
    /* allocation */
    if (! (inode.ind_direct[bloc] = new_bloc()))
return BLOC_NULL;
    /* write the inode */
    write_inode(inumber, &inode);
}
return inode.ind_direct[bloc];
    }
```

Notons que les blocs alloués doivent être initialisés ! La valeur d'initialisation est zéro. C'est la sémantique des fichiers Unix. La fonction `new_bloc`ou une fonction intermédiaire peut s'en charger...

5.5 Quatrième couche logicielle : manipulation de fichiers

Pour manipuler les fichiers, les utilisateurs du système disposent de différentes fonctions : création, destruction, ouverture, fermeture, lecture, écriture, positionnement, vidage du tampon d'écriture.

Dans un premier temps, notre système de fichiers identifie un fichier directement par son numéro d'inœud. Nous modifierons cette interface par la suite pour identifier les fichier par des noms.

Ces fonctions sont donc les suivantes (`ifile` pour *inumber file*) :

```
unsigned int create_ifile(enum file_type_e type);
int delete_ifile(unsigned int inumber);

int open_ifile(file_desc_t *fd, unsigned int inumber);
void close_ifile(file_desc_t *fd);
void flush_ifile(file_desc_t *fd);
void seek_ifile(file_desc_t *fd, int r_offset);  /* relatif */
void seek2_ifile(file_desc_t *fd, int a_offset); /* absolu */

int readc_ifile(file_desc_t *fd);
int writec_ifile(file_desc_t *fd, char c);
int read_ifile(file_desc_t *fd, void *buf, unsigned int nbyte);
int write_ifile(file_desc_t *fd, const void *buf, unsigned int nbyte);
```

Exercice 38 Création et suppression d'un fichier

Réalisez la fonction `create_ifile()` en allouant et renseignant un inœud dont le numéro sera retourné. Puis réalisez `delete_ifile()`, qui supprime un fichier en libérant tous les blocs qui peuvent être associés au fichier, y compris le bloc d'inœud.

Ces deux fonctions sont de simples appels à `create_inode`et `delete_inode`.

Lorsque un fichier est manipulé par un programme, celui-ci utilise la fonction `int open_ifile(file_desc_t *fd, unsigned inumber)` qui « ouvre » un fichier en initialisant une structure de donnée `file_desc_t`. Cette structure de donnée, qui sert la manipulation d'un fichier, contient en fait toutes les informations nécessaires à la gestion d'un fichier. On y trouve notamment :

- le numéro de l'inœud qui décrit le fichier visé ;
- la position en octet du curseur dans le fichier. Cette position donne l'octet à lire/écrire dans le fichier ouvert, et elle est incrémentée après chaque opération d'accès ;
- la taille totale du fichier (utilisée lors des accès en lecture) ;
- un tableau d'octets qui sert de tampon entre le bloc du disque et le programme ;
- un drapeau qui indique si le tampon courant à été modifié ou pas.

Exercice 39 Structure d'accès au fichier

Décrivez la structure de données `file_desc_t`. C'est immédiat :

```
struct file_desc_s {
    unsigned int fds_inumber;           /* inode number of the file */
    unsigned int fds_size;              /* file size in char */
    unsigned int fds_pos;               /* cursor in the file, in char */
    unsigned char fds_buf[BLOC_SIZE];   /* memory copy of bloc at fds_pos */
    char fds_dirty;                     /* buffer write back is needed */
};

typedef struct file_desc_s file_desc_t;
```

On peut expliquer qu'il est inhabituel que cette structure de données soit dans l'espace « utilisateur », mais cela facilite l'implantation. On a habituellement juste un `FILE` qui est un `int` servant de clé pour l'accès à une structure interne.

Exercice 40 Accès au fichier

Proposez d'abord le code pour la fonction `open_ifile()` qui initialise un descripteur de fichier

en « ouvrant » le fichier désigné. La fonction `open_sfile()` retourne un statut. On explique ce statut : erreur ssi pas de fichier existant. Donc détrompeur sur les inœuds ? Et remise à zéro lors de la libération ?

Le reste est trivial :

- initialiser la structure `file_desc_t` par rapport à l'inœud donné en paramètre ;
- initialiser le tampon :
  - le charger depuis un bloc (le premier)
  - ou le remplir de zéros.

```
int
open_sfile(file_desc_t *fd, unsigned inumber)
{
    struct inode_s inode;

    /* load the file inode */
    read_bloc(current_volume, inumber, (unsigned char*) &inode);

    /* we are opening this file ! */
    fd->fds_inumber = inumber;

    /* other trivial init */
```

```
    fd->fds_size = inode.ind_size;
    fd->fds_pos = 0;

    /* the buffer is full of zeros if the first bloc is zero, loaded
       with this first bloc otherwise */
    if (! inode.ind_direct[0])
        memset(fd->fds_buf, 0, BLOC_SIZE);
    else
        read_bloc(current_volume, inode.ind_direct[0], fd->fds_buf);

    /* last trivial */
    fd->fds_dirty = 0;

    return RETURN_SUCCESS;
}
```

Puis implantez la fonction `flush_ifile()` qui vide le tampon courant sur le disque (si le drapeau indique que le tampon et modifié). Et enfin proposez le programme qui termine l'utilisation d'un fichier : `close_ifile(file_desc_t *fd).` Pour la fermeture, c'est simplement un flush.

Pour le flush, il s'agit de savoir s'il y a quelque chose à flusher ou pas. Si oui, il s'agit de réaliser l'écriture du tampon.

On choisit une implantation qui assure que le flush n'a pas à allouer de bloc, on ne tombera donc pas sur une erreur "`out-of-bloc`". C'est l'écriture qui alloue (qui aura préalablement alloué) le nouveau bloc si nécessaire et qui tient à jour les informations de l'inœud.

En conséquence, flush est finalement assez simple à écrire, sauf le calcul du bloc à écrire :

```
void
flush_ifile(file_desc_t *fd)
{
    unsigned int fbloc; /* bloc index in the file */
    unsigned int vbloc; /* bloc index in the volume */

    if (fd-> fds_dirty) {
        /* compute the number of the bloc on the volume associated to
           the buffer */
        fbloc = bloc_of_pos(fd->fds_pos);
        vbloc = vbloc_of_fbloc(fd-> fds_inumber, fbloc);

        /* write back the buffer */
        write_bloc(current_volume, vbloc, fd->fds_buf);

        /* done */
        fd-> fds_dirty = 0 ;
    }
}
```

La macro `bloc_of_pos()` retourne le rang du bloc, au sein des blocs du fichier, dans lequel se trouve une position donnée dans le fichier (`cfile.h`) :

```
#define bloc_of_pos(pos) ((pos) / BLOC_SIZE)
```

Exercice 41 Se déplacer dans un fichier

Les deux fonctions `seek_ifile()` et `seek2_ifile()` déplacent le curseur d'accès dans le fichier ouvert. Ces fonctions doivent mettre à jour le tampon afin que les données du tampon mémoire

soient cohérentes avec le curseur dans le fichier. Bien entendu, un déplacement relatif peut être implanté à partir d'un déplacement absolu, ou (exclusif!) l'inverse. Vu cette dernière remarque, on ne traite qu'un des cas. Il s'agit de

- calculer la nouvelle position ;
- si la position fait que l'on change de bloc (on utilise `bloc_of_pos()` pour le savoir) :
  - on écrit l'ancien bloc : flush
  - on lit le nouveau bloc : calculer son index, le lire

Le code, que l'on peut ne pas détailler : c'est un mixte de ce qui est fait pour `open_sfile()` et `flush_sfile()` :

```
void
seek_ifile(file_desc_t *fd, int offset)
{
    unsigned int old_pos = fd->fds_pos;
    unsigned int fbloc, vbloc;

    /* update the position */
    fd->fds_pos += offset;

    /* does the seek imply a jump in another bloc? */
    if (bloc_of_pos(fd->fds_pos) != bloc_of_pos(old_pos)) {
        /* flush */
        flush_sfile(fd);

        /* the bloc index of the new buffer */
        fbloc = bloc_of_pos(fd->fds_pos);
        vbloc = vbloc_of_fbloc(fd->fds_inumber, fbloc);

        if (! vbloc)
            /* the bloc #0 is full of zeros */
            memset(fd->fds_buf, 0, BLOC_SIZE);
        else
            /* load the bloc */
            read_bloc(current_volume, vbloc, fd->fds_buf);
    }
}
```

Exercice 42 Lire et écrire un octet

Finalement réalisez les fonctions d'accès : `int readc_ifile()` et `writec_ifile()` qui permettent de lire ou d'écrire un octet dans un fichier. Une fois l'accès réalisé, le descripteur de fichier se place « automatiquement » sur l'octet suivant.

Lire un octet devient trivial, il suffit de lire dans le tampon modulo `BLOC_SIZE`, puis de faire un `seek_ifile(-,1)` :

```
int
readc_ifile(file_desc_t *fd)
{
    char c;

    /* eof? */
    if (fd->fds_pos >= fd->fds_size)
        return READ_EOF;
```

```
    /* the data is in the buffer, just return it */
    c = fd->fds_buf[ibloc_of_pos(fd->fds_pos)];

    /* seek + 1 */
    seek_ifile(fd, 1);

    return c;
}
```

Si la fin de fichier est atteinte l'acte de lecture retourne une valeur négative (`READ_EOF`) et la position dans le fichier n'est pas modifiée.

En cas d'écriture au delà de la fin, le fichier est automatiquement « étendu » en conséquence. Notez encore qu'un accès en lecture sur un bloc qui n'est défini retourne zéro. Un accès en écriture sur le même octet implique l'allocation d'un nouveau bloc, initialisé à zéro avec seulement l'octet écrit de modifié. On définit donc (`cfile.h`) :

```
#define READ_EOF                -2
#if READ_EOF >= 0
#  error "READ_EOF must be negative"
#endif
```

Le gros du travail est l'allocation éventuelle d'un nouveau bloc dans l'écriture. Il s'agit d'écrire la fonction (`inode.h`) :

```
/* allocate and return a bloc on the volume (in order to write in the
   file).
   This may imply indirect and d_indirect bloc creation.
   Return BLOC_NULL if no allocation was possible. */
unsigned int allocate_vbloc_of_fbloc(unsigned int inumber, unsigned int bloc);
```

Cette fonction est très similaire à `vbloc_of_fbloc()` si ce n'est que lorsque l'on à déterminé le numéro du bloc, il faut vérifier qu'il est déjà alloué (non `BLOC_NULL`), et sinon l'allouer (`new_bloc()`) et remplir le tampon de zéros, exemple :

```
    /* not already allocate? */
    if (! inode.ind_direct[bloc]) {
        /* allocation */
        if (! (inode.ind_direct[bloc] = new_bloc()))
            return BLOC_NULL;
        /* write the inode and bye */
        write_inode(inumber, &inode);
    }
    return inode.ind_direct[bloc];
```

À partir de là, on écrit facilement `writec_ifile()`.

Exercice 43 Lecture et écriture dans un fichier

À l'aide des fonctions précédentes, écrivez les fonctions :

```
int read_ifile(file_desc_t *fd, void *buf, unsigned int nbyte);
int write_ifile(file_desc_t *fd, const void *buf, unsigned int nbyte);
```

qui, respectivement, lit `nbyte` octets depuis le fichier `fd` en les stockant en mémoire centrale dans le tableau de caractère `buf`, et écrit `nbyte` octets lus depuis le tableau de caractère `buf` en mémoire centrale vers le fichier `f`. `for(i=0; ...)`, you know !

Validation de la bibliothèque de manipulation d'un fichier par son inombre

Pour valider votre travail, vous pouvez écrire des versions des commandes de l'exercice 5.7 qui identifie les fichiers paramètres par leur inombre ; la création d'un fichier retournant le inombre associée au fichier :

`if_status` liste les informations associées à la partition courante.

`if_pfile` (*print file*) affiche sur la sortie standard le contenu d'un fichier dont le inombre est passée en paramètre.

`if_nfile` (*new file*) crée un fichier. Le contenu du fichier est lu sur l'entrée standard. Le inombre du fichier est retournée sur la sortie standard ; il servira à identifier le fichier lors de futures commandes.

`if_dfile` (*delete file*) supprime le fichier dont le inombre est passé en paramètre.

`if_cfile` (*copy file*) copie le contenu du fichier dont le inombre est donné en paramètre dans le second fichier. Ce second fichier est donc créé ; son inombre est affiché sur la sortie standard.

5.6 Cinquième couche logicielle : système de noms de fichier

Exercice 44 Structure d'un répertoire

Les répertoires sont « simplement » des fichiers particuliers. Ils contiennent au minimum une liste de noms (les fichiers et répertoires contenus dans le répertoire) et pour chaque nom un numéro d'inœud associé. Ainsi un répertoire est un tableau dont chaque entrée est le descripteur d'un fichier. Soit `struct entry_s` le type d'un tel descripteur. La taille du tableau est directement calculable en fonction de la taille d'une entrée et de la taille du fichier-répertoire.

Pour simplifier le retrait d'un fichier dans la liste des fichiers d'un répertoire, on identifie les entrées détruites (par exemple avec une valeur remarquable pour un des champs de `entry_s`).

Question 44.1 Donnez la déclaration de la structure de donnée associée à une entrée dans le répertoire : `struct entry_s`. Juste un numéro d'inœud et un nom d'entrée (`dir.c`) :

```
struct entry_s {
    unsigned int ent_inumber;
    char ent_basename[ENTRYMAXLENGTH];
};
```

On convient que les entrées de inombre nul sont libres.

On peut expliquer que cette implantation n'est pas habituelle et que sous Unix les répertoires sont formées de blocs **complets** dont les entrées non utilisées sont initialisées comme libres.

Question 44.2 Proposez un programme qui affiche la liste des fichiers contenu dans un répertoire (le répertoire étant identifié par son numéro d'inœud). Un exercice de mise en jambes :

- lire l'inœud ;
- vérifier que c'est un répertoire ;
- pour chacune des entrées (taille du fichier / taille d'une entrée)
  - lire une entrée à l'aide de `read_ifile()`
  - si le inombre est non nul : affichage

Question 44.3 Proposez les fonctions utilitaires

```
unsigned int new_entry(file_desc_t *dir_fd);
int find_entry(file_desc_t *dir_fd, const char *basename);
```

`new_entry()` retourne l'index dans un répertoire (vu comme un tableau d'entrées) de la première entrée libre. Le répertoire est identifié par un descripteur de fichier, il a donc préalablement été ouvert.

`find_entry()` retourne l'index dans un répertoire d'une entrée dont le nom est donnée, une valeur négative si aucune entrée est trouvée. On rappelle que les entrées libres sont identifiées par un inombre nul.

```
static unsigned int
new_entry(file_desc_t *fd)
{
    struct entry_s entry;
    unsigned int ientry = 0; /* the new entry index */

    /* seek to begin of dir */
    seek2_ifile(fd, 0);

    /* look after a null inumber in an entry */
    while (read_ifile (fd, &entry, sizeof(struct entry_s)) != READ_EOF) {
        if (! entry.ent_inumber)
            return ientry;
        ientry++;
    }

    /* need to append an entry in the dir */
    return ientry;
}
```

Une utilisation typique de cette fonction (création d'une entrée dans un répertoire, extrait de `int add_entry()`) :

```
/* the new entry position in the file */
ientry = new_entry(fd);


/* seek to the right position */
seek2_ifile(fd, ientry * sizeof(struct entry_s));

/* write the entry */
nbyte = write_ifile(fd, &entry, sizeof(struct entry_s));
```

`find_entry()` est construite sur le même modèle.

Question 44.4 Proposez maintenant les deux fonctions de création et destruction d'une entrée dans un répertoire connu `add_entry()` et `del_entry()`. Immédiat, voir question précédente. La difficulté est d'identifier les paramètres devant être passés à ces fonctions :

```
int add_entry(unsigned int idir, unsigned int inumber, const char *basename);
int del_entry(unsigned int idir, const char *basename);
```

On en est pas à référencer les répertoires par leur nom, mais uniquement par leur inombre, `idir`. Les informations nécessaires pour ajouter une entrée sont : son inombre, `inumber`, et son nom dans le répertoire, `basename`. Pour supprimer une entrée, son nom suffit.

On peut aussi se soucier de ne pas créer une entrée dont le nom existe déjà ou ne pas s'en soucier et laisser la chose à la charge de l'appelant...

Exercice 45 Nom de fichier et inœud

Il s'agit d'identifier le inœud correspondant à un fichier dont on connaît le nom complet absolu.

Cette identification se fait de proche en proche, par exemple pour le fichier `/usr/bin/emacs` :

1. on identifie le inœud de la racine `/` : il est contenu dans le superbloc ;
2. on recherche un fichier nommée `usr` dans le répertoire correspondant à ce inœud, on en identifie le inœud ;
3. on recherche alors un fichier nommé `bin` dans le répertoire correspondant à ce inœud ;

4. etc.

Pour réaliser cette opération, on peut écrire successivement les fonctions suivantes :

```
unsigned int inumber_of_basename(unsigned int idir, const char *basename);
unsigned int inumber_of_path(const char *pathname);
unsigned int dinumber_of_path(const char *pathname, const char **basename);
```

La fonction `inumber_of_basename()` retourne le inombre de l'entrée de nom `basename` (qui ne doit pas comporter de `/`) dans le répertoire `idir`, 0 en cas d'échec.

La fonction `inumber_of_path()` retourne le inombre d'un nom de fichier *absolu*, 0 en cas d'échec.

La fonction `dinumber_of_path()` retourne à la fois le inombre d'un nom de fichier absolu, mais aussi le nom relatif de ce fichier dans son répertoire dans le paramètre `basename`. La valeur retournée pour `basename` est un pointeur dans `pathname`. `inumber_of_basename()` se termine sur un échec si `idir` ne correspond pas à un répertoire ou si l'entrée `basename` n'existe pas dans le répertoire. L'implantation consiste à ouvrir le répertoire, appeler `find_entry()`, lire le inombre dans le `struct entry_s`.

L'implantation de `inumber_of_path()` consiste en un traitement de chaînes de caractères, la définition des « bonnes » variables (`basename`, `pos`, et `lg`) aide beaucoup :

```
unsigned int
inumber_of_path(const char *pathname)
{
    unsigned int icurrent;      /* the inumber of the current dir */

    /* an *absolute* pathname */
    if (*pathname != '/')
        return 0;

    /* start at root */
    icurrent = super.super_root;

    while (*pathname) {
        if (*pathname != '/') {
            char basename[ENTRYMAXLENGTH];
            char *pos;          /* the first / position */
            int lg;             /* the length of the first basename */

            /* length of the leading basename */
            pos = strchr(pathname, '/');
            lg = pos ? pos - pathname : strlen (pathname);

            /* copy this leading basename to basename */
            strncpy (basename, pathname, min(ENTRYMAXLENGTH, lg));
            basename[min(ENTRYMAXLENGTH, lg)] = 0;

            /* look after this basename in the directory.
               this entry inumber is the new current */
            icurrent = inumber_of_basename(icurrent, basename);
            if (! icurrent)
                return 0;

            /* skip the basename in pathname */
            pathname += lg;

            /* may end here */
```

```
            if (! *pathname) break;
        }
        pathname++ ;
    }
    return icurrent ;
}
```

Et le bien pratique mais pas bien beau `inumber_of_path()` :

```
unsigned int
dinumber_of_path(const char *pathname, const char **basename)
{
    char *dirname = strdup(pathname);
    unsigned int idirname = 0;
    struct inode_s inode;

    /* an *absolute* pathname */
    if (*pathname != '/')
        goto free;

    /* the last basename (there is at least a '/') */
    *basename = strrchr (pathname, '/');
    (*basename)++;

    /* the dirname stops at the last '/'. ugly isn't it! */
    *(dirname + ((*basename) - pathname)) = 0;

    /* the dirname inumber */
    idirname = inumber_of_path(dirname);
    if (! idirname)
        goto free;

    /* is dirname a directory? */
    read_inode(idirname, &inode);
    if (inode.ind_type != directory)
        idirname = 0;

 free:
    /* free dirname strdup() */
    free(dirname);

    return idirname;
}
```

Exercice 46 Une bibliothèque pour travailler avec les fichiers

La couche finale de votre travail consiste en une bibliothèque d'accès aux fichiers, avec les fonctions de création, suppression, ouverture, lecture, écriture, déplacement, vidage des tampons et fermeture de fichiers en respectant l'interface donnée dans l'encart page suivante. Pas de difficultés ici. Peut-être un mot sur le fait que la création d'un répertoire nécessite de créer deux entrées `.` et `..` et que la destruction de ces entrées est impossible.

5.7 Des programmes de base : commandes de manipulation de fichiers

Exercice 47 Des programmes de base pour le système de fichiers

En utilisant votre bibliothèque, réalisez les programmes de base suivants. Il travaille tous avec le fichier de configuration désigné par la variable d'environnement `HW_CONFIG`et la partition désignée par `CURRENT_VOLUME`, voir l'encart page 48 :

`mknfs` (*make new file system*) crée un système de fichiers sur le volume courant.

`dfs` (*display file system*) liste les informations associées à la partition courante ; voir l'encart page 48

`pdir` (*print directory*) affiche la liste de entrées du répertoire donné en paramètre sous la forme d'un nom absolu.

`ndir` (*new directory*) crée un répertoire dont le nom (absolu) est passé en paramètre.

`pfile` (*print file*) affiche sur la sortie standard le contenu d'un fichier dont le nom (absolu) est passée en paramètre.

`nfile` (*new file*) crée le fichier dont le nom est passé en paramètre. Le contenu du fichier est lu sur l'entrée standard. Si le fichier existait ; il est préalablement détruit.

`dfile` (*delete file*) supprime le fichier ou répertoire (qui doit être vide) dont le nom est passé en paramètre.

`cfile` (*copy file*) copie le contenu du premier fichier dont le nom est donné en paramètre dans le second fichier dont le nom est donné en paramètre.

C'est du facile. On peut préciser que soit `mknfs` est utilisé pour créer le répertoire racine `/` ou qu'une opération de création d'un système de fichiers vide existe...