Université Lille1
Sciences et Technologies

**Master d'informatique**
**1re année**
**Module d'architecture**
**des systèmes d'exploitation**

# Changement de contexte et ordonnancement

Philippe MARQUET Gilles GRIMAUD

Novembre 2003
mise à jour d'Octobre 2011

Ce sujet est disponible en ligne à `www.lifl.fr/~marquet/ens/ctx/`. Cet accès en ligne autorise des copier/coller... ne vous en privez pas.

## 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 );
}
```

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 );
}
```

□

**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.
```

□

***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`.
On se référera au tutoriel d'utilisation d'assembleur x86 en ligne disponible à `http://www-106.ibm.com/developerworks/linux/library/l-ia.html` ou à la documentation de GCC disponible à `http://gcc.gnu.org/onlinedocs/`. Une copie locale est disponible à partir de la version en ligne du présent document.

**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 */
        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);
}
```

□

***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 3.
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.

## 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`.

***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.

```
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`. □

**Question 4.2** La fonction `try()` sauvegarde un contexte et appelle la fonction passée en paramètre. Donnez une implantation de `try()`. □

**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()`. □

# 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).
2. Étendez la structure de donnée `struct ctx_s` qui décrit un tel contexte. □

**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`. □

# 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()`. □

# 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. □

**Exercice 9**
Donnez une implantation de `yield()`. □

# 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();
```

***Simulateur de matériel***

La bibliothèque `hw` qui vous est fournie dans

```
/home/enseign/ASE/src/hw.tgz
```

implémente l'interface décrite section 5 à l'aide signaux et timers POSIX. Vous pouvez en étudier le code ou l'utiliser sans vous en soucier...

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()`. □

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. □

# 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 ;
– la synchronisation de processus (un processus doit en attendre un autre pour continuer ou commencer son exécution).

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**

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);
```

□

**Exercice 13**

Proposez une implantation des deux primitives

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

□

***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 ?

Université Lille1
Sciences et Technologies

**Master informatique**
**1re année**
**Module d'architecture**
**des systèmes d'exploitation**

# Changement de contexte et ordonnancement - Compléments

Philippe MARQUET

mise à jour d'octobre 2007

## 7 Prévention des interblocages

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. □

**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. □

***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.

# 8 Création de contextes par duplication

Sur une idée de Laurent Noé.

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. □

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

**Exercice 18**
Discutez des moments opportuns auxquels réaliser cette réalllocation. □

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.

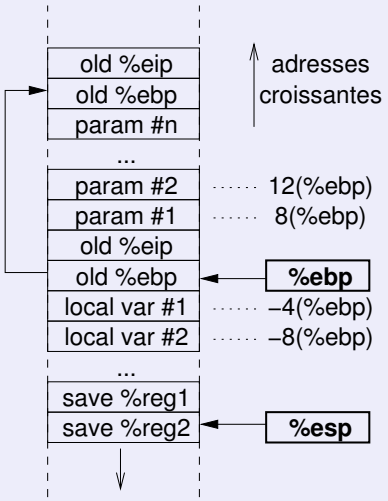
***Convention d'appel et organisation de la pile (partie 2)***

**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.

**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. □

## 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. □



fin (toujours provisoire...) du sujet