CSC4103 – Programmation système

François Trahay
Gaël Thomas

2024

Contents

Chapter 1

Le langage C

1.1 Présentation du module

Objectifs du module:

- Maîtriser le langage C
- Savoir s'adresser au système d'exploitation depuis un programme

Modalités:

- Un peu de théorie
- Beaucoup de pratique

1.1.1 Contenu du module

Partie *Programmation*

- CI 1 – Le langage C
- CI 2 – Les types composés / qu'est-ce qu'une adresse ?
  - Exercice Hors-Présentiel
- CI 3 – Faire des programmes modulaires en C
- CI 4 – Les pointeurs
- CI 5 – Debugger un programme

Partie *Système*

- CI 6 – Les fichiers
- CI 7 – Les processus
- CI 8 – Appels système et Sémaphores
- CI 9 – Signaux

Evaluation

- CI 10 – Exercice de synthèse
- CF1 (sur papier) – questions sur l'exercice de synthèse

1.1.2 Déroulement d'une séance

Système de *classe inversée*. Pour chaque séance :

- **Avant** la séance
  - Etude de la partie cours de la séance à venir
- **Pendant** la séance:
  - Mini-évaluation de la partie cours (Kahoot!)
  - Explications sur les points mal compris
  - Travaux pratiques : expérimentations sur les concepts vus en cours

Attention ! Cela ne fonctionne que si vous travaillez sérieusement **avant** la séance.

Hypothèse: les étudiants suivant ce cours sont des adultes responsables.

1.1.3 Ressources disponibles

Pour vous aider, vous avez à votre disposition:

- Le poly contenant l'ensemble des transparents commentés
- Les transparents en version pdf
- La documentation des fonctions C standard (`man 2 <fonction>` ou `man 3 <fonction>`)
- Une équipe enseignante de choc !

1.2 C vs. Java

- langage de *bas niveau* vs. *haut niveau*
- En C, manipulation de la mémoire et de ressources proches du matériel
- *"Un grand pouvoir implique de grandes responsabilités"*[1]
- programmation impérative vs. programmation objet

[1] *B. Parker, Amazing Fantasy, 1962*

1.3 Mon premier programme en C

Fichier ***.c**

```c
/* hello_world.c */
#include <stdio.h>
```

```c
#include <stdlib.h>

int main(int argc, char** argv) {
  printf("Hello World!\n");
  return EXIT_SUCCESS;
}
```

Compilation/execution:

```
$ gcc hello_world.c -o hello_world -Wall -Werror
$ ./hello_world
Hello World!
```

- Les `#include <stdio.h>` indiquent que le programme a besoin des outils `stdio`. Il s'agit donc d'un équivalent du `import package` de Java
- Pour afficher un message dans le terminal, on utilise la fonction `printf("message\n");`
- Le `return EXIT_SUCCESS;` à la fin du `main` permet de spécifier le code retour du programme (accessible depuis le shell en faisant `echo $?`). En cas d'erreur, on peut retourner `EXIT_FAILURE` à la place de `EXIT_SUCCESS`.

1.3.1 Déclaration de variable

Pour les types simples, déclaration identique à Java:

```c
int var1;
int var2, var3, var4;
int var5 = 42;
```

Types disponibles: * pour les entiers: `int`, `short`, `long`, `long long` * pour les flottants: `float`, `double` * pour les caractères: `char`

Pour les entiers: possibilité de préfixer le type par `unsigned`. Les variables sont alors non-signées (ie. positives).

La taille d'une variable entière (ie. le nombre de bits/octets) dépend de l'implémentation. Le standard C ne spécifie que la taille minimum. Ainsi, un `int` doit faire au moins 16 bits, alors que la plupart des implémentations modernes utilisent 32 bits pour les `int`. Il convient donc de ne pas se reposer sur ces types lorsqu'on a besoin d'un nombre précis de bits/octets.

Pour cela, il est préférable d'utiliser les types fournis par `stdint.h`: `uint8_t` (8 bits), `uint16_t` (16 bits), `uint32_t` (32 bits), ou `uint64_t` (64 bits).

Comme en Java, les variables déclarées dans une fonction sont *locales* à la fonction (elles disparaissent donc dès la sortie de la fonction). Les variables déclarées en dehors d'une fonction sont *globales*: elles sont accessibles depuis n'importe quelle fonction.

1.3.2 Opérateurs et Expressions

La liste des opérateurs disponibles est à peu près la même qu'en Java:

- arithmétique : `+`, `-`, `*`, `/`, `\%`
- affectation : `=`, `+=`, `-=`, `*=`, `/=`, `\%=`
- incrémentation/décrémentation: `++`, `{-}`
- comparaison: `<`, `<=`, `>`, `>=`, `==`, `!=`
- logique: `!`, `\&\&`, `||`

Mais également:

- `sizeof n`: donne le nombre d'octets qui constitue une variable/un type `n`

1.3.3 Opérateurs bit à bit

Possibilité de travailler sur des *champs de bits*.

- Opération sur les bits d'une variable
- décalage: `<<`, `>>`
- OR : `|`, AND :`&`, XOR : `^`, NOT : `~`
- affectation: `<<=`, `>>=`, `|=`, `&=`, `^=`, `~=`

```c
/* bits.h */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(int argc, char** argv) {
  uint32_t v = 1;
  int i;

  /* l'operateur << decale vers la gauche */
  for(i=0; i<32; i++) {
    /* v << i  decale les bits de v de i places vers la gauche
     * c'est equivalent à calculer v*(2^i)
     */
    printf("v<<%d = %u\n", i, v<<i);
  }

  v = 5;
  /* v | 3 effectue un OU logique entre les bits de v et la representation binaire de 3
   * 101 | 11 = 111 (7)
   */
  printf("%u | %u = %u\n", v, 3, v|3);

  /* v & 3 effectue un ET logique entre les bits de v et la representation binaire de 3
```

```
   * 101 & 11 = 001 (1)
   */
  printf("%u & %u = %u\n", v, 3, v&3);

  /* v ^ 3 effectue un XOR logique entre les bits de v et la representation binaire de 3
   * 101 ^ 011 = 110 (6)
   */
  printf("%u ^ %u = %u\n", v, 3, v^3);

  /* ~v effectue un NON logique des bits de v
   * ~ 00...00101 = 11..11010 (4294967290)
   */
  printf("~%u = %u\n", v, ~v);

  return EXIT_SUCCESS;
}
```

1.3.4 Remarque

Lorsqu'on opère un décalage (avec `<<`) sur une valeur signée (par exemple, un `int`), le bit de signe n'est pas modifié par le décalage. Par exemple, si les bits d'un `int a` sont à `1010 0000 0000 0000 0000 0000 0000 0000`, le résultat de `a >> 1` est `1001 0000 0000 0000 0000 0000 0000 0000`.

1.3.5 Structures algorithmiques

Comme en Java:

- `for(i=0; i<n; i++) { ... }`
- `while(cond) {... }`
- `do { ... } while(cond);`
- `if (cond) { ... } else { ... }`

1.3.6 Affichage / Lecture

- Pour afficher: `printf("%d exemple de %f format \n", v1, v2);`
- Pour lire: `scanf("%d-%f", &v1, &v2);`

```
/* formats.c */
#include <stdio.h>

int main(int argc, char** argv) {
  int v;
  printf("Entrez la valeur de v:\n");
  scanf("%d", &v);
  printf("v = %d (en decimal)\n", v);
```

```
  printf("v = %u (en decimal non signe)\n", v);
  printf("v = %x (en hexadecimal)\n", v);
  printf("v = %o (en octal)\n", v);
  printf("v = %c (en ASCII)\n", v);

  double a;
  scanf("%lf", &a);
  printf("a = %f (en flottant)\n", a);
  printf("a = %lf (en flottant double precision)\n", a);
  printf("a = %e (en notation scientifique)\n", a);

  char *chaine = "Bonjour";
  printf("chaine = %s\n", chaine);
  printf("chaine = %p (adresse)\n", chaine);

  printf("On peut aussi afficher le caractère %%\n");
}
```

1.3.7 Fonctions

Déclaration:

```
type_retour nom_fonc(type_param1 param1, type_param2 param2) {
/* déclaration des variables locales */
/* instructions à exécuter */
}
```

En C, il est d'usage de nommer les fonctions en minuscule, en séparant les mots par `_`. Par exemple, l'équivalent en C de la fonction Java `calculerLeMinimum()` sera `calculer_le_minimum()`.

Chapter 2

Les structures et les tableaux

2.1 Du type primitif au type composé

- Jusqu'à maintenant, vous avez vu les types primitifs
  - `char`, `short`, `int`, `long long` (signés par défaut, non signés si préfixés de `unsigned`)
  - `float` (4 octets), `double` (8 octets)
- Dans ce cours, nous apprenons à définir de nouveaux types de données
  - Une **structure** est constituée de sous-types **hétérogènes**
  - Un **tableau** est constitué de sous-types **homogènes**

2.1.1 Les structures

Une structure est une définition d'un nouveau type de données

- composé de sous-types nommés (primitifs ou composés)
- possédant un nom

Remarque: les sous-types d'une structure s'appellent des champs

Définition d'une nouvelle structure avec :

```
struct nom_de_la_structure {
  type1 nom_du_champs1;
  type2 nom_du_champs2;
  ...
};
```

Par exemple:

```
struct nombre_complexe {
  int partie_reelle;
```

```
  int partie_imaginaire;
};
```

Par convention, les noms de structures commencent par une minuscule en C

Une structure peut aussi être composée à partir d'une autre structure, comme dans cet exemple:

```
struct point {
  int x;
  int y;
};

struct segment {
  struct point p1;
  struct point p2;
};
```

En revanche, une structure ne peut pas être composée à partir d'elle-même. À titre d'illustration, l'exemple suivant n'est pas correct:

```
struct personnage {
  struct personnage ami;
  int               point_de_vie;
};
```

Cette construction est impossible car il faudrait connaître la taille de la structure `personnage` pour trouver la taille de la structure personnage.

2.1.1.1 Déclaration d'une variable de type structure

- Une déclaration d'une variable de type structure se fait comme avec un type primitif:

```
struct nombre_complexe z1, z2, z3;
```

- On peut aussi initialiser les champs de la structure lors de la déclaration:

```
/* partie_relle de z prend la valeur 0 */
/* partie_imaginaire de z prend la valeur 1 */
struct nombre_complexe i = { 0, 1 };
/* autre solution : */
struct nombre_complexe j = { .partie_reelle=0, .partie_imaginaire=1 };
```

L'initialisation d'une variable de type structure est différente lorsque la variable est déclarée globalement ou localement. On vous rappelle qu'une variable globale si elle est déclarée en dehors de toute fonction. Sinon, on dit qu'elle est locale.

Lorsqu'une variable de type structure est déclarée en tant que variable globale sans être initialisée, le compilateur initialise chacun de ces champs à la valeur 0. En revanche, lorsqu'une structure est déclarée en tant que variable locale dans une fonction sans être initialisée, ces champs prennent une valeur aléatoire.

Par exemple, dans :

```
struct nombre_complexe i;

void f() {
  struct nombre_complexe j;
}
```

Les champs de `i` sont initialisés à 0 alors que ceux de `j` prennent une valeur aléatoire.

On peut aussi partiellement initialiser une structure comme dans l'exemple suivant :

```
struct nombre_complexe j = { 1 };
```

Dans ce cas, le champs `partie_relle` prend la valeur 1 et le champs `partie_imaginaire` prend soit la valeur 0 si la variable est globale, soit une valeur aléatoire si la variable est locale à une fonction.

2.1.1.2 Accès aux champs d'une variable de type structure

- L'accès aux champs d'une variable de type structure se fait en donnant le nom de la variable, suivi d'un point, suivi du nom du champs :

```
struct point {
  int x;
  int y;
};

struct ligne {
  struct point p1;
  struct point p2;
};

void f() {
  struct point p;
  struct ligne l;

  p.x = 42;
  p.y = 17;
  l.p1.x = 1;
  l.p1.y = 2;
  l.p2 = p; /* copie p.x/p.y dans l.p2.x/l.p2.y */

  printf("[%d %d]\n", p.x, p.y);
}
```

2.1.2 Les tableaux

- Un tableau est un type de données composé de sous-types homogènes
  - Les éléments d'un tableau peuvent être de n'importe quel type (primitif, structure, mais aussi tableau)
  - Pour déclarer un tableau:

  `type_des_elements nom_de_la_variable[taille_du_tableau];`

  Par exemple:

```
int          a[5];      /* tableau de 5 entiers */
double       b[12];     /* tableau de 12 nombres flottants */
struct point c[10];     /* tableau de 10 structures points */
int          d[12][10]; /* tableau de 10 tableaux de 12 entiers */
                        /* => d est une matrice 12x10 */
```

2.1.2.1 Accès aux éléments d'un tableau

- L'accès à l'élément `n` du tableau `tab` se fait avec `tab[n]`
- Un tableau est indexé à partir de zéro (éléments vont de 0 à N - 1)

```
void f() {
  int x[3];
  int y[3];
  int i;

  /* 0 est le premier élément, 2 est le dernier */
  for(i=0; i<3; i++) {
    x[i] = i;
    y[i] = x[i] * 2;
  }
}
```

2.1.2.2 Tableaux et structures

- On peut mixer les tableaux et les structures, par exemple :

```
struct point {
  int x;
  int y;
};

struct triangle {
  struct point sommets[3];
};
```

```c
void f() {
  struct triangle t;

  for(i=0; i<3; i++) {
    t.sommets[i].x = i;
    t.sommets[i].y = i * 2;
  }
}
```

2.1.2.3 Différences par rapport à Java

- On ne peut pas accéder à la taille d'un tableau
- Lors d'un accès en dehors des bornes du tableau, l'erreur est silencieuse :

  c'est une erreur, mais elle n'est pas signalée immédiatement

  => parmi les erreurs les plus fréquentes (et les plus difficiles à repérer) en C

```c
void f() {
  int x[3];

  x[4] = 42; /* Erreur silencieuse !!! */
             /* Écriture à un emplacement aléatoire en mémoire */
             /* le bug pourra apparaître n'importe quand */
}
```

2.1.2.4 Initialisation d'un tableau lors de sa déclaration

- Un tableau peut être initialisé lorsqu'il est déclaré avec

```c
type_element nom_variable[taille] = { e0, e1, e2, ... };
```

- Par exemple: `int x[6] = { 1, 2, 3, 4, 5, 6 };`
- Comme pour les structures, on peut partiellement initialiser un tableau
  - Par exemple: `int x[6] = { 1, 1, 1 };`

En l'absence d'initialisation : * Si le tableau est une variable globale, chaque élément est initialisé à 0 * Sinon, chaque élément est initialisé à une valeur aléatoire

Lorsqu'on initialise un tableau lors de sa déclaration, on peut omettre la taille du tableau. Dans ce cas, la taille du tableau est donnée par la taille de la liste d'initialisation.

```c
int x[] = { 1, 2, 3 };  /* tableau à trois éléments */
int y[6] = { 1, 2, 3 }; /* tableau à six éléments, avec les trois premiers initialisés */
```

2.1.2.5 Initialisation mixte de tableaux et structures

- On peut composer des initialisations de tableaux et de structures

```
struct point {
  int x;
  int y;
};

struct triangle {
  struct point sommets[3];
};

struct triangle t = {
  { 1, 1 },
  { 2, 3 },
  { 4, 9 }
};
```

2.1.2.6 Tableaux et chaînes de caractères

Une chaîne de caractère est simplement un tableau de caractères terminé par le caractère `'\0'` (c'est à dire le nombre zéro)

```
char yes[] = "yes";
```

est équivalent à

```
char yes[] = { 'y', 'e', 's', '\0' };
```

2.1.3 Passage par valeur et par référence

- En C, il existe deux types de passage d'arguments :
  - Passage **par valeur** : l'argument est copiée de l'appelé vers l'appelant
    => l'argument et sa copie sont deux variables différentes
  - Passage **par référence** : une référence vers l'argument de l'appelant est donné à l'appelé
    => l'appelant et l'appelé partagent la même donnée
- Par défaut :
  - Les **tableaux** sont passés par **référence**
    - Un argument de type tableau est déclaré avec `type nom[]`, sans la taille
  - Les **autres types** sont passés par **copie**

2.1.3.1 Passage par valeur – les types primitifs

```c
/* le x de f et le x du main sont deux variables distinctes */
/* le fait qu'elles aient le même nom est anecdotique */
void f(int x) {
  x = 666;
  printf("f : x = %d\n", x);            /* f : x = 666 */
}

int main() {
  int x = 42;
  f(x);                           /* x est copié dans f */
      /* => le x de main n'est donc pas modifié par f */
  printf("g : x = %d\n", x);             /* g : x = 42 */
  return 0;
}
```

2.1.3.2 Passage par valeur – les structures

```c
struct point {
  int x;
  int y;
};

void f(struct point p) {
  p.x = 1;
  printf("(%d, %d)\n", p.x, p.y);         /* => (1, 2) */
}

int main() {
  struct point p = { -2, 2 };
  f(p);                           /* p est copié dans f */
  printf("(%d, %d)\n", p.x, p.y);       /* => (-2, 2)  */
  return 0;
}
```

2.1.3.3 Passage par référence – les tableaux

```c
void print(int x[], int n) {
  for(int i=0; i<n; i++) {
    printf("%d ", x[i]);
  }
  printf("\n");
}
```

```c
int main() {
  int tab[] = { 1, 2, 3 };

  print(tab, 3);  /* => 1 2 3 */
  f(tab);
  print(tab, 3); /* => 1 42 3 */

  return 0;
}

/* x est une référence vers le tableau original */
void f(int x[]) {
  x[1] = 42;           /* => modifie l'original */
}
```

2.1.4 Notions clés

- Les structures
  - Une structure définit un nouveau type de donné
  - Définition : `struct nom { type_1 champs_1; type_2 champs_2; ... };`
  - Déclaration : `struct nom var = { v1, v2 };`
  - Utilisation : `var.champs_i`
- Les tableaux
  - Un tableau est un type de donné
  - Déclaration : `int tab[] = { 1, 2, 3 };`
  - Utilisation : `tab[i]`
  - Une chaîne de caractère est un tableau de caractère terminé par un zéro
- Passage par valeur ou par référence
  - Les tableaux sont passés par référence
  - Les autres types sont passés par valeur

Chapter 3

Modularité

3.1 Objectifs de la séance

- Savoir modulariser un programme
- Savoir définir une chaîne de compilation
- Maîtriser les options de compilations courantes

3.2 Modularité en C vs. Java

Beaucoup de concepts sont les même qu'en Java

- **Interface** d'un module: partie publique accessible d'autres modules
  - prototype des fonctions
  - définition des constantes et types
- **Implémentation** d'un module: partie privée
  - définition et initialisation des variables
  - implémentation des fonctions
- **Bibliothèque** (en anglais: *library*) : regroupe un ensemble de modules
  - Equivalent des *packages* java

3.2.1 Module en C

Deux fichiers par module. Par exemple, pour le module `mem_alloc`:

- Interface: fichier `mem_alloc.h` (fichier d'*entête / header*)
  - défini les constantes/types

  - déclare les prototypes des fonctions “publiques” (*ie.* accessible par les autres modules)
- Implémentation: fichier `mem_alloc.c`
  - utilise `mem_alloc.h` : `#include "mem_alloc.h"`
  - utilise les constantes/types de `mem_alloc.h`
  - déclare/initialise les variables
  - implémente les fonctions
- Utiliser le module `mem_alloc` (depuis le module `main`)
  - utilise `mem_alloc.h` : `#include "mem_alloc.h"`
  - utilise les constantes/types de `mem_alloc.h`
  - appelle les fonctions du module `mem_alloc`

3.2.2 Exemple: le module `mem_alloc`

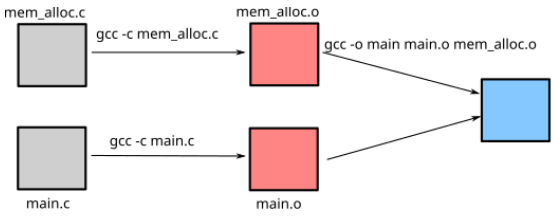
```c
/* mem_alloc.h */
#include <stdlib.h>
#include <stdint.h>
#define DEFAULT_SIZE 16

typedef int64_t mem_page_t;
struct mem_alloc_t {
  /* [...] */
};

/* Initialize the allocator */
void mem_init();

/* Allocate size consecutive bytes */
int mem_allocate(size_t size);

/* Free an allocated buffer */
void mem_free(int addr, size_t size);


/* mem_alloc.c */
#include "mem_alloc.h"
struct mem_alloc_t m;
void mem_init() {  /* ... */  }
int mem_allocate(size_t size) {
  /* ... */
}
void mem_free(int addr, size_t size) {
  /* ... */
}


#include "mem_alloc.h"
```

```c
int main(int argc, char**argv) {
  mem_init();
  /* ... */
}
```

3.3 Compilation de modules

Compilation en trois phases:

- Le **preprocesseur** prépare le code source pour la compilation
- Le **compilateur** transforme des instructions C en instructions "binaires"
- L'**éditeur de liens** regroupe des fichiers objets et crée un `exécutable`



3.3.1 Préprocesseur

Le **préprocesseur** transforme le code source pour le compilateur

- génère du code source
- interprète un ensemble de directives commençant par #
  - `#define N 12` substitue `N` par `12` dans le fichier
  - `#if <cond> ... #else ... #endif` permet la compilation conditionnelle
  - `#ifdef <var> ... #else ... #endif` (ou l'inverse: `#ifndef`) permet de ne compiler que si `var` est défini (avec `#define`)
  - `#include "fichier.h"` inclue (récursivement) le fichier `"fichier.h"`
- résultat visible avec : `gcc -E mem_alloc.c`

La directive `#if` permet, par exemple, de fournir plusieurs implémentations d'une fonction. Cela peut être utilisé pour des questions de portabilité.

```c
#if __x86_64__
  void foo() { /* implementation pour CPU intel 64 bits */ }
#elif __arm__   /* équivalent à #else #if ... */
  void foo() { /* implementation pour CPU ARM */ }
#else
  void foo() {
    printf("Architecture non supportée\n");
    abort();
#endif
```

Il y a deux syntaxes pour la directive `#include`: * `#include <fichier>`: le préprocesseur cherche `fichier` dans un ensemble de répertoires systèmes (`/usr/include` par exemple) * `#include "fichier"" le préprocesseur cherche`fichier‘ dans le répertoire courant, puis dans les répertoires systèmes.

On utilise donc généralement `#include "fichier"` pour inclure les fichiers d'entête définis par le programme, et `#include <fichier>` pour les fichiers d'entête du système (`stdio.h`, `stdlib.h`, etc.)

3.3.2 Compilateur

Compilation : transformation des instructions C en instructions "binaires"

- appliquée à chaque module
- `gcc -c mem_alloc.c`
- génère le **fichier objet** `mem_alloc.o`
- génère des instructions "binaires" dépendantes du processeur

---

3.3.3 Editeur de liens

Edition de liens : regroupement des fichiers **objets** pour créer un **exécutable**

```
gcc -o executable mem_alloc.o module2.o [...] moduleN.o
```

Règles de compilations

- En cas de modification du corps d'un module (par exemple `mem_alloc.c`, il est nécessaire de regénérer le fichier objet (`mem_alloc.o`), et de regénérer l'exécutable. Il n'est toutefois pas nécessaire de recompiler les modules utilisant le module modifié.
- En cas de modification de l'interface d'un module (par exemple `mem_alloc.h`), il est nécessaire de recompiler le module, ainsi que tous les modules utilisant le module modifié. Une fois que tous les fichiers objets (les fichiers `*.o`) concernés ont été regénérés, il faut refaire l'édition de liens.

Lorsque le nombre de module devient élevé, il devient difficile de savoir quel(s) module(s) recompiler. On automatise alors la chaîne de compilation, en utilisant l'outil `make`.

Puisque la compilation se fait en 3 phases, 3 types d'erreurs peuvent survenir: * une erreur du préprocesseur (ie. une macro est mal écrite):

```
$ gcc -c foo.c
foo.c:1:8: error: no macro name given in #define directive
 #define
       ^
```

- une erreur lors de la compilation (ie. le programme est mal écrit):

```
  $ gcc -c erreur_compil.c
erreur_compil.c: In function ‘f’:
erreur_compil.c:3:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

- une erreur lors de l'édition de liens (ie. il manque des morceaux):

```
$ gcc -o plip main.o
main.o : Dans la fonction « main » :
main.c:(.text+0x15) : référence indéfinie vers « mem_init »
collect2: error: ld returned 1 exit status
```

3.3.4 Fichiers ELF

- Les fichiers objets et exécutables sont sous le format **ELF** (*Executable and Linkable Format*)
- Ensemble de sections regroupant les symboles d'un même type:
  - `.text` contient les fonctions de l'objet
  - `.data` et `.bss` contiennent les données initialisées (`.data`) ou pas (`.bss`)
  - `.symtab` contient la *table des symboles*
- Lors de l'édition de liens ou du chargement en mémoire, les sections de tous les objets sont fusionnés

La liste complète des sections du format ELF est disponible dans la documentation (`man 5 elf`).

La *table des symboles* contient la liste des fonctions/variables globales (ou statiques) définies ou utilisées dans le fichier. L'outil `nm` permet de consulter cette table. Par exemple:

```
$ nm mem_alloc.o
0000000000000000 C m
0000000000000007 T mem_allocate
0000000000000012 T mem_free
0000000000000000 T mem_init
$ nm main.o
0000000000000000 T main
                 U mem_init
```

Pour chaque symbole, `nm` affiche l'adresse (au sein d'une section), le type (donc, la section ELF), et le nom du symbole.

Ces informations sont également disponible via la commande `readelf`:

```
$ readelf  -s mem_alloc.o

Table de symboles « .symtab » contient 12 entrées :
   Num:    Valeur          Tail Type    Lien   Vis      Ndx Nom
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
```

```
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS mem_alloc.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000001     0 OBJECT  GLOBAL DEFAULT  COM m
     9: 0000000000000000     7 FUNC    GLOBAL DEFAULT    1 mem_init
    10: 0000000000000007    11 FUNC    GLOBAL DEFAULT    1 mem_allocate
    11: 0000000000000012    14 FUNC    GLOBAL DEFAULT    1 mem_free
```

L'utilitaire `objdump` permet lui aussi d'examiner la table des symboles:

```
$ objdump -t mem_alloc.o

mem_alloc.o:     format de fichier elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 mem_alloc.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000       O *COM*  0000000000000001 m
0000000000000000 g     F .text  0000000000000007 mem_init
0000000000000007 g     F .text  000000000000000b mem_allocate
0000000000000012 g     F .text  000000000000000e mem_free
```

3.3.5 Portée des variables locales

Une variable déclarée dans une fonction peut être

- **locale** : la variable est allouée à l'entrée de la fonction et désallouée à sa sortie
  - exemple: `int var;` ou `int var2 = 17;`
- **statique** : la variable est allouée à l'initialisation du programme.
  - Sa valeur est conservée d'un appel de la fonction à l'autre.
  - utilisation du mot-clé `static`
  - exemple: `static int var = 0;`

Puisqu'une variable locale statique est allouée au chargement du programme, elle apparaît dans la liste des symboles :

```
$ nm plop.o
0000000000000000 T function
```

```
0000000000000000 d variable_locale_static.1764
```

Ici, le symbole `variable_locale_static.1764` correspond à la variable `variable_locale_static` déclarée `static` dans la fonction `function`. Le suffixe `.1764` permet de différencier les variables nommées `variable_locale_static` déclarées dans des fonctions différentes.

3.3.6 Portée des variables globales

Une variable déclarée dans le fichier `fic.c` en dehors d'une fonction peut être:

- **globale** : la variable est allouée au chargement du programme et désallouée à sa terminaison
  - exemple: `int var;` ou `int var2 = 17;`
  - la variable est utilisable depuis d'autres objets
- **extern** : la variable est seulement déclarée
  - équivalent du prototype d'une fonction: la déclaration indique le type de la variable, mais celle ci est allouée (ie. déclarée globale) ailleurs
  - utilisation du mot-clé `extern`
  - exemple: `extern int var;`
- **statique** : il s'agit d'une variable globale accessible seulement depuis `fic.c`
  - utilisation du mot-clé `static`
  - exemple: `static int var = 0;`

Les variables globales (déclarées `extern`, `static`, ou "normales") se retrouvent dans la table des symboles de l'objet, mais dans des sections ELF différentes :

```
$ nm plop.o
0000000000000000 T function
                 U var_extern
0000000000000000 D var_globale
0000000000000004 d var_static_globale
```

La variable `var_extern` (déclarée avec `extern int var_extern;`) est marquée `"U"` (*undefined*). Il s'agit donc d'une référence à un symbole présent dans un autre objet.

La variable `var_globale` (déclarée avec `int  var_globale = 12;`) est marquée `"D"` (*The symbol is in the initialized data section*). Il s'agit donc d'une variable globale initialisée [1].

La variable `var_static_globale` (déclarée avec `static int var_static_globale = 7;`) est marquée "`d`" (*The symbol is in the initialized data section*). Il s'agit donc d'une variable globale "interne". Il n'est donc pas possible d'accèder à cette variable depuis un autre objet:

```
$ gcc plop.o  plip.o  -o executable
plip.o : Dans la fonction « main » :
plip.c:(.text+0xa) : référence indéfinie vers « var_static_globale »
collect2: error: ld returned 1 exit status
```

---

[1] D'après la documentation de nm à propos du type de symbole : "*If lowercase, the symbol is usually local; if uppercase, the symbol is global (external)*".

3.4 Bibliothèque

- Regroupement de fichiers objets au sein d'une bibliothèque
  - Equivalent d'un *package* Java
  - Accès à tout un ensemble de modules
- Utilisation
  - dans le code source: `#include "mem_alloc.h"`, puis utilisation des fonctions
  - lors de l'édition de lien: ajouter l'option `-l`, par exemple: `-lmemory`
    - Utilise la bibliothèque `libmemory.so` ou `libmemory.a`

Avantages/inconvénients des bibliothèques statiques:

- Taille de l'exécutable important (puisqu'il inclut la bibliothèque);

- En cas de nouvelle version d'une bibliothèque (qui corrige un bug par exemple), il faut recompiler toutes les applications utilisant la bibliothèque;

- Duplication du code en mémoire;

+ L'exécutable incluant une bibliothèque statique fonctionne "tout seul" pas besoin d'autres fichiers).

Avantages/inconvénients des bibliothèques dynamiques:

+ Taille de l'exécutable réduite (puisqu'il n'inclut qu'une référence à la bibliothèque);

+ En cas de nouvelle version d'une bibliothèque (qui corrige un bug par exemple), pas besoin de recompiler les applications utilisant la bibliothèque;

+ Une instance du code en mémoire est partageable par plusieurs processus;

- L'exécutable incluant une bibliothèque dynamique ne fonctionne pas "tout seul": il faut trouver toutes les bibliothèques dynamiques nécessaires.

Les "dépendances" dues aux bibliothèques dynamiques sont visibles avec `ldd`:

```
$ ldd executable
    linux-vdso.so.1 (0x00007fff9fdf6000)
    libmem_alloc.so (0x00007fb97cb9f000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb97c7ca000)
    /lib64/ld-linux-x86-64.so.2 (0x0000555763a9b000)
```

3.4.1 Création d'une bibliothèque

Il existe 2 types de bibliothèques

- **Bibliothèque statique** : `libmemory.**a**`
  - Intégration des objets de la bibliothèque au moment de l'édition de liens
  - Création: `ar rcs libmemory.a mem_alloc.o mem_plip.o mem_plop.o [...]`
- **Bibliothèque dynamique** : `libmemory.**so**`
  - Intégration des objets de la bibliothèque à l'exécution
  - Lors de l'édition de liens: une référence vers la bibliothèque est intégrée à l'exécutable

- Création: `gcc -shared -o libmemory.so mem_alloc.o mem_plip.o mem_plop.o [...]`
  - les objets doivent être créés avec l'option `-fPIC`:
  - `gcc -c mem_alloc.c -fPIC`

3.4.2 LD_LIBRARY_PATH

Pour exécuter un programme utilisant une bibliothèque dynamique, le système doit charger en mémoire le programme ainsi que la biblbiothèque. Si la bibliothèque n'est pas installée dans un répertoire standard (typiquement dans `/usr/lib`), il peut être nécessaire d'indiquer où trouver cette bibliothèque grâce à la variable d'environnement `LD_LIBRARY_PATH`.

```
$ ./mon_programme
mon_programme: error while loading shared libraries: libtruc.so: cannot open shared object file: No suc

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/trahay/libs/libtruc/

$ ./mon_programme

It works !
```

3.4.3 Avantages et inconvénients

L'avantage d'une bibliothèque statique est qu'il n'est nécessaire de connaître son emplacement qu'au moment d'édition de liens. Une fois l'exécutable crée, celui-ci inclue la bibliothèque statique. On peut donc déplacer l'exécutable, supprimer la bibliothèque statique, recopier l'exécutable sur une autre machine sans empêcher son exécution.

Toutefois, lorsqu'une bibliothèque statique est mise à jour (par exemple pour corriger un bug ou une faille de sécutité), il est nécessaire de recompiler tous les programmes utilisant cette bibliothèque. Pour des bibliothèques très utilisées (par exemple, la libc), cela peut être long et on risque d'oublier de recompiler certains programmes.

A l'inverse, si une bibliothèque dynamique est mise à jour, cette mise à jour est directement disponible pour toutes les applications utilisant la bibliothèque. Ceci explique pourquoi la plupart des distributions Linux reposent aujourd'hui sur des bibliothèques dynamiques plutot que statiques.

3.4.4 Changement d'ABI

Attention, si une bibliothèque dynamique est mise à jour et que son ABI (Application Binary Interface) change (par exemple si la signature d'une fonction change), il sera nécessaire de recompiler les applications utilisant cette biblliothèque.

3.4.5 Organisation

Organisation classique d'un projet:

- src/
  - module1/
    - module1.c
    - module1.h
    - module1_plop.c
  - module2/
    - module2.c
    - module2.h
    - module2_plip.c
  - etc.
- doc/
  - manual.tex
- etc.

Besoin d'utiliser des `flags`:

- `-Idir` indique où chercher des fichiers `.h`

```
gcc -c main.c -I../memory/
```

- `-Ldir` indique à l'éditeur de lien où trouver des bibliothèques

```
gcc -o executable main.o -L../memory/ -lmem_alloc
```

A l'exécution:

- la variable `LD_LIBRARY_PATH` contient les répertoires où chercher les fichiers `.so`

```
export LD_LIBRARY_PATH=.:../memory
```

Par défaut, le compilateur va chercher les fichiers d'entête dans un certain nombre de répertoires. Par exemple, `gcc` cherche dans:

- `/usr/local/include`
- `<libdir>/gcc/<target>/version/include`
- `/usr/<target>/include`
- `/usr/include`

L'option `-I` ajoute un répertoire à la liste des répertoires à consulter. Vous pouvez donc utiliser plusieurs fois l'option `-I` dans une seule commande. Par exemple:

```
gcc -c main.o -Imemory/ -Itools/ -I../plop/
```

De même, l'éditeur de liens va chercher les bibliothèques dans un certain nombre de répertoires par défaut. La liste des répertoires parcourus par défaut par `ld` (l'éditeur de lien utilisé par `gcc`) est dans le fichier `/etc/ld.so.conf`. On y trouve généralement (entre autre):

- `/lib/<target>`
- `/usr/lib/<target>`
- `/usr/local/lib`
- `/usr/lib`
- `/lib32`
- `/usr/lib32`

Si la variable `LD_LIBRARY_PATH` est mal positionnée, vous risquez de tomber sur ce type d'erreur au lancement de l'application:

```
  $ ./executable
./executable: error while loading shared libraries: libmem_alloc.so: cannot open \
 shared object file: No such file or directory
```

3.5 Makefile

- Arbre des dépendances
  - "Pour créer `executable`, j'ai besoin de `mem_alloc.o` et `main.o`"
- Action
  - "Pour créer `executable`, il faut lancer la commande `gcc -o executable mem_alloc.o main.o`"
- Syntaxe: dans un fichier `Makefile`, ensemble de règles sur deux lignes:

```
cible : dependance1 dependance2 ... dependanceN`
<TAB>commande
```

- Pour lancer la compilation: commande `make`
  - Parcourt l'arbre de dépendance et détecte les cibles à regénérer
  - Exécute les commandes pour chaque cible

3.6 La commande `make`

- La commande `make` parcourt le fichier `Makefile` du répertoire courant et tente de produire la première *cible*.
- Il est également possible de spécifier une cible à produire en utilisant `make cible`.
- Dès qu'une *action* génère une erreur, la commande `make` s'arrête
- La liste des cibles à regénérer est calculée à partir de l'arbre de dépendance décrit dans le fichier `Makefile` et des date/heure de dernière modification des fichiers: si un fichier de l'arbre est plus récent que la cible à générer, tout le chemin entre le fichier modifié et la cible est regénéré.

3.7 Le fichier `Makefile`

Voici un exemple de fichier `Makefile`:

```
all: executable

executable: mem_alloc.o main.o
	gcc -o executable main.o mem_alloc.o

mem_alloc.o: mem_alloc.c mem_alloc.h
	gcc -c mem_alloc.c
```

```
main.o: main.c mem_alloc.h
        gcc -c main.c
```

- La (ou les) commande(s) à exécuter pour générer chaque cible est précédée du caractère *Tabulation*. Sous `emacs`, lorsque vous éditez un fichier nommé `Makefile`, les tabulations sont colorisées en rose par défaut.

Si vous utilisez des espaces à la place de la tabulation, la commande `make` affiche le message d'erreur suivant:

```
Makefile:10: *** missing separator (did you mean TAB instead of 8 spaces?). Arrêt.
```

- Puisque la commande `make` (sans argument) génère la première cible, on ajoute généralement une premier cible "artificielle" `all` décrivant l'ensemble des exécutables à générer:

```
all: executable1 executable2
```

Dans ce cas, seule la cible est spécifiée. Il n'y a pas d'action à effectuer.

3.7.1 Règle `clean`

- On ajoute également souvent une règle artificielle `clean` pour "faire le ménage" (supprimer les exécutables et les fichiers `.o`):

```
clean:
<TAB>rm -f executable1 executable2 *.o
```

- Lorsqu'on écrit un fichier `Makefile`, on peut utiliser certaines notations symboliques:
  - `$@` désigne la cible de la règle
  - `$^` désigne l'ensemble des dépendances
  - `$<` désigne la première dépendance de la liste
  - `$?` désigne l'ensemble des dépendances plus récentes que la cible
- Il est possible de définir et d'utiliser des variables dans un fichier `Makefile`. Par exemple:

```
BIN=executable
OBJETS=mem_alloc.o main.o
CFLAGS=-Wall -Werror -g
LDFLAGS=-lm

all: $(BIN)

executable: $(OBJETS)
        gcc -o executable main.o mem_alloc.o $(LDFLAGS)

mem_alloc.o: mem_alloc.c mem_alloc.h
        gcc -c mem_alloc.c $(CFLAGS)

main.o: main.c mem_alloc.h
        gcc -c main.c $(CFLAGS)
```

```
clean:
        rm -f $(BIN) $(OBJETS)
```

3.8 Configuration et dépendances

- Dans “la vraie vie”, l’outil `make` n’est qu’une partie de la chaîne de configuration et de compilation.
- Des outils comme `autoconf/automake` ou `Cmake` sont fréquemment utilisés pour écrire des “proto-makefiles”
- Ces outils permettent de détecter la configuration de la machine (quel CPU ? la bibliothèque X est-elle installée ? etc.) et de définir les options de compilation de manière portable.

Chapter 4

Pointeurs

4.1 Espace mémoire d'un processus

- Espace mémoire dans lequel un processus peut stocker des données/du code
- Séparé en plusieurs parties (*segments*), dont:
  - *pile* (*stack*): les variables locales et paramètres de fonctions
  - *tas* (*heap*): les variables globales
  - *segment de code* : le code (binaire) du programme

4.2 Adresse mémoire

* On peut faire référence à n'importe quel octet de l'espace mémoire grace à son adresse * Adresse mémoire virtuelle codée sur k bits[1] * donc 2^k octets accessibles (de `00...00` à `11...11`)

- exemple: à l'adresse `0x1001` est stocké l'octet `0x41`
  - peut être vu comme un `char` (le caractère `A`)
  - peut être vu comme une partie d'un `int` (par exemple l'entier `0x11**41**2233`)

[1] k dépend de l'architecture. Sur les processeurs modernes (64 bits), on a $k = 64$.

4.3 Rappel: hexadécimal

Les valeurs préfixées par `0x` sont représentées en hexadécimal (en base 16). Ainsi, `0x200d` correspond au nombre qui s'écrit `200D` en base 16, soit le nombre $2 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 13 \times 16^0 = 8205$ écrit en base 10.

La notation hexadécimale est couramment utilisée pour représenter des octets car deux chiffres en hexadécimal permettent de coder 256 (soit 2^8) valeurs différentes. On peut donc représenter les 8 bits d'un octet avec deux chiffres hexadécimaux. Par exemple, `0x41` représente l'octet `0100 0001`.

4.4 Architecture des processeurs

Les processeurs équipant les ordinateurs modernes sont généralement de type `x86_64`. Pour ces processeurs, les adresses virtuelles sont codées sur 64 bits. Un processus peut donc adresser 2^{64} octets (16 Exaoctets ou 16 x 1024 Pétaoctets) différents.

ARM est une autre architecture de processeur très répandue puisqu'elle équipe la plupart des smartphones. Jusqu'à très récemment, les processeurs ARM fonctionnaient en 32 bits. Un processus pouvait donc accéder à 2^{32} octets (4 Gigaoctets). Les processeurs ARM récents sont maintenant 64 bits, ce qui permet à un processus d'utiliser une plus grande quantité de mémoire.

4.4.1 Adresse d'une variable

- `&var` désigne l'adresse de `var` en mémoire
- affichable avec `%p` dans `printf`:

```
printf("adresse de var: %p\n", &var);
```

affiche:

```
adresse de var: 0x7ffe8d0cbc7f
```

Il est possible de manipuler l'adresse de n'importe quel objet en C, que ce soit une variable, le champ d'une structure, ou une case d'un tableau.

Le programme suivant:

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

struct point{
  float x;
  float y;
  float z;
  int id;
};

int main() {
  char var='A';
  printf("adresse de var: %p\n", &var);

  struct point p = {.x = 2.5, .y = 7.2, .z=0, .id=27};
  printf("adresse de p: %p\n", &p);
  printf("adresse de p.x: %p\n", &p.x);
```

```
  printf("adresse de p.y: %p\n", &p.y);
  printf("adresse de p.z: %p\n", &p.z);
  printf("adresse de p.id: %p\n", &p.id);

  char tab[] = "hello";
  printf("adresse de tab[2] = %p\n", &tab[2]);
  return EXIT_SUCCESS;
}
```

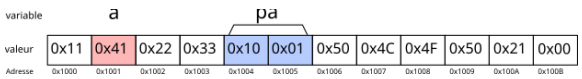
peut donner cet affichage:

```
adresse de var: 0x7ffe44d7c6df
adresse de p: 0x7ffe44d7c6c0
adresse de p.x: 0x7ffe44d7c6c0
adresse de p.y: 0x7ffe44d7c6c4
adresse de p.z: 0x7ffe44d7c6c8
adresse de p.id: 0x7ffe44d7c6cc
adresse de tab[2] = 0x7ffe44d7c6b2
```

4.5 Pointeur

- Variable dont la valeur est une adresse mémoire
- valeur binaire codée sur k bits (k dépend de l'architecture du processeur)
- déclaration: `type* nom_variable;`
  - `type` désigne le type de la donnée "pointée"
- exemple: `char* pa;` crée un pointeur sur une donnée de type `char`:

```
// pour l'exemple, les adresses sont codees sur 32 bits
char a = 'A'; // a est stocke a l'adresse 0x0000FFFF
              // la valeur de a est 0x41 ('A')
char* pa = &a; // pa est une variable de 32 bits stockee
               // aux adresses 0xFFFB a 0xFFFE
               // la valeur de pa est 0x0000FFFF (l'adresse de a)
```



On peut ensuite manipuler l'adresse de `a` (`0xFFFF`) ou la valeur de `pa` (`0xFFFF`) indifféremment:

```
printf("&a = \%p\n", &a); // affiche 0xFFFF
printf("pa = \%p\n", pa); // affiche 0xFFFF
printf("&pa = \%p\n", &pa); // affiche 0xFFFB, soit l'adresse de pa
```

Un pointeur étant une variable comme les autres, on peut donc stocker son adresse dans un pointeur.
Par exemple:

```
char a = 'A'; // a est stockee a l'adresse 0xFFFF et contient 0x41 ('A' ou 65)
char* pa = &a; // pa est stockee a l'adresse 0xFFFB et contient 0xFFFF (l'adresse de a)
char** ppa = &pa; // ppa est stockee a l'adresse 0xFFF7 et contient 0xFFFB (l'adresse de pa)
```

4.6 Conseil de vieux baroudeur

Quand vous déclarez un pointeur, initialisez-le immédiatement, soit avec l'adresse d'une variable, soit avec la valeur `NULL` (définie dans `stdlib.h`) qui est la valeur pointant sur "rien". Dit autrement, ne laissez **jamais** une variable pointeur avec un contenu non initialisé.

4.7 Arithmétique de pointeur

Les opérateurs `+`, `-`, `++`, et `--` sont utilisables sur des pointeurs, mais avec précaution.

Incrémenter un pointeur sur `type` aura pour effet d'ajouter `sizeof(type)` à la valeur du pointeur. Par exemple:

```
char* pa = &a;  // pa vaut 0xFFFF
pa--;           // enleve sizeof(char) (c'est a dire 1) a pa
                // donc pa vaut 0xFFFE

char**ppa = &pa // ppa vaut 0xFFFB
ppa--;          // enleve sizeof(char*) (c'est a dire 4) a ppa
                // donc ppa vaut 0xFFF7

ppa = ppa - 2;  // soustrait 2*sizeof(char*) (donc 8) a ppa
                // ppa vaut 0xFFEF
```

4.8 Exemple complet

Sur https://codecast.france-ioi.org/, vous pouvez visualiser le contenu de la mémoire d'un programme. Pour cela, saisissez le code source du programme, cliquez sur "compiler", puis exécutez le programme pas à pas en cliquant sur "next expression". {'A} chaque instant, le contenu de la mémoire est représenté en bas de la page. Essayez avec ce programme:

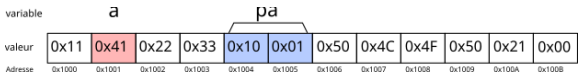
```
#include <stdio.h>
int main() {
  //! showMemory(cursors=[a, pa, ppa], start=65528)
  char a = 'A';
  char* pa= &a;
  char** ppa = &pa;

  printf("a = %d, &a=%p\n", a, &a);
  printf("pa = %p, &pa=%p\n", pa, &pa);
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  pa--; // enleve sizeof(char)=1 a pa
  ppa--; // enleve sizeof(char*) a ppa

  printf("a = %d, &a=%p\n", a, &a);
  printf("pa = %p, &pa=%p\n", pa, &pa);
```

```
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  ppa = ppa - 2; // enleve 2*sizeof(char*) a ppa
  printf("ppa = %p, &ppa=%p\n", ppa, &ppa);

  return 0;
}
```

4.8.1 Déréférencement

- Permet de consulter la valeur stockée à l'emplacement désigné par un pointeur
  - `* ptr`
  - exemple:

```
char a = 'A'; // valeur 0x41 (cf. codage ASCII)
char* pa = &a;
printf("pa = %p\n", pa); // affiche "pa = 0xFFFF"
printf("*pa = %c\n", *pa); // affiche "*pa = A"
*pa = 'B';   // modifie l'emplacement memoire 0xFFFF
             // (donc change la valeur de a)
printf("a = %c\n", a); // affiche "a = B"
```



A partir d'un pointeur, on peut donc afficher 3 valeurs différentes:

- `printf("pa = %p\n", pa);` – affiche la valeur du pointeur (ici, l'adresse `0x1001`)
- `printf("*pa = %c\n", *pa);` – affiche la valeur "pointée" par `pa` (ici, la valeur de `a` : `'A'`)
- `printf("&pa = %c\n", &pa);` – affiche l'adresse du pointeur (ici, l'adresse `0x1004`)

4.8.2 Déréférencement dans une structure

Lorsqu'un pointeur `ptr` contient l'adresse d'une structure, l'accès au champ `c` de la structure peut se faire en déréférençant le pointeur, puis en accédant au champ : `(*ptr).c`

Cela devient plus compliqué lorsque la structure contient un pointeur (`p1`) vers une structure qui contient un pointeur (`p2`) sur une structure. La syntaxe devient rapidement indigeste: `(*(*(*ptr).p1).p2).c`

Pour éviter cette notation complexe, on peut utiliser l'opérateur `->` qui combine le déréférencement du pointeur et l'accès à un champ. Ainsi, `ptr->c` est équivalent à `(*ptr).c`. On peut donc remplacer la notation `(*(*(*ptr).p1).p2).c` par `ptr->p1->p2->c`.

4.8.3 Tableaux et pointeurs (1/3)

Si un tableau est **un argument de fonction**

- la déclaration est **remplacée** par celle d'un pointeur
  - `void f(int x[])` <=> `void f(int* x)`
- un accès effectue un décalage + déréférencement
  - `tab[i]` réécrit en `*(tab + i)`
  - exemple: si `tab = 0x1000` et `i=5`
  - `tab[i]` calcule `0x1000 + (5*sizeof(int)) = 0x1000 + 0x14 = 0x1014`
- `sizeof(tab)` donne la taille d'un pointeur
- Remarque : `&tab` donne l'adresse de `int[] tab`, donc `&tab != tab`

4.8.4 Tableaux et pointeurs (2/3)

Si un tableau est **une variable locale ou globale**

- le tableau **n'est pas remplacé par un pointeur**
- le tableau doit avoir une taille connue
  - `int tab[3]; // alloue 3 int`
    - `tab` est le nom de cet espace mémoire
  - `int tab[] = { 1, 2, 3 }; // idem + initialisation`
  - `int tab[]; // interdit !`
- `sizeof(tab)` renvoie la taille du tableau

4.8.5 Tableaux et pointeurs (3/3)

Si un tableau est **une variable locale ou globale** (suite)

- `&tab` donne l'adresse du tableau
  - Remarque : `&tab == &tab[0]` car `tab` et `tab[0]` désignent les mêmes emplacements mémoires
- `tab` est **implicitement transtypé vers son pointeur** au besoin
- Exemple :
  - `int* tab2 = tab;` réécrit en `int* tab2 = &tab`
  - `if(tab == &tab)` récrit en `if(&tab == &tab)`
  - `f(tab)` réécrit en `f(&tab)`
  - `printf("%p %p\n", tab, &tab);` réécrit en `printf("%p %p\n", &tab, &tab);`
  - `tab[i]` réécrit en `(&tab)[i]` puis en `*(&tab + i)`
  - `*(tab + i)` réécrit en `*(&tab + i)`

4.8.6 Passage par référence

Rappel:

- *Passage par référence: une référence vers l'argument de l'appelant est donné à l'appelé* (cf. CI2)
- Cette référence est un pointeur

```c
void f(int* px) {
  *px = 666;          // la variable pointee par px est modifiee
}

int main() {
  int x = 42;
  f(&x);                    // l'adresse de x est donnee à f
            //  => le x de main est modifié par f
  printf("x = \%d\n", x); // la nouvelle valeur de x : 666
  return EXIT_SUCCESS;
}
```

4.9 Allocation dynamique de mémoire

- `void* malloc(size_t nb_bytes);`
  - Alloue `nb_bytes` octets et retourne un pointeur sur la zone allouée
- usage:
  - `char* str = malloc(sizeof(char)* 128);`
  - renvoie `NULL` en cas d'erreur (par ex: plus assez de mémoire)

Attention ! Risque de "fuite mémoire" si la mémoire allouée n'est jamais libérée

4.10 Signification de `void*`

Le `void*` renvoyé par `malloc` signifie que la fonction retourne un pointeur vers n'importe quel type de donnée. Ce pointeur (qui est donc une adresse) vers `void` peut être converti en pointeur (une adresse) vers `int` ou tout autre type.

4.11 Recommandation

Vérifiez systématiquement si `malloc` vous a renvoyé `NULL` et, si c'est le cas, arrêtez votre programme. Une manière simple et lisible de faire cela est d'utiliser la macro `assert` (définie dans `assert.h`) comme dans l'exemple suivant :

```c
char* str = malloc(sizeof(char)* 128);
assert(str);
```

4.12 Fuites mémoire

Lorsque l'on déclare une variable (un `int`, un tableau, une structure, ou toute autre variable) depuis une fonction `foo`, l'espace mémoire de cette variable est réservé sur la pile. Lorsque l'on sort de `foo`, la pile est "nettoyée" et l'espace réservé pour les variables locales est libéré.

Lorsque l'on alloue de la mémoire avec `malloc` depuis une fonction `foo`, la mémoire est allouée sur le *tas*. Lorsque l'on sort de la fonction `foo`, l'espace mémoire réservé reste accessible. Si on "perd"

l'emplacement de cette zone mémoire, elle devient donc inaccessible, mais reste réservée: c'est une *fuite mémoire*.

Si la fuite mémoire fait "perdre" quelques octets à chaque appel de la fonction `foo`, la mémoire de la machine risque, à terme, d'être remplie de zones inutilisées. Le système d'exploitation n'ayant plus assez de mémoire pour exécuter des processus devra donc en tuer quelques uns pour libérer de la mémoire.

4.12.1 Libération de mémoire

- `void free(void* ptr);`
- Libère la zone allouée par `malloc` est situé à l'adresse `ptr`
- Attention à ne pas libérer plusieurs fois la même zone!

A chaque fois que vous faites `free` sur un pointeur, pensez à remettre ensuite ce pointeur à `NULL` (pour être sûr que vous n'avez pas un pointeur qui pointe sur une zone de mémoire libérée). Dit autrement, tout "`free(ptr);`" doit être suivi d'un "`ptr = NULL;`".

4.12.2 Notions clés

- L'espace mémoire d'un processus
- Les pointeurs
  - Adresse mémoire d'une variable (`&var`)
  - Pointeur sur `type` : `type* ptr;`
  - Arithmétique de pointeurs (`ptr++`)
  - Adresse nulle: `NULL`
  - Déréférencement d'un pointeur:
    - types simples: `*ptr`
    - structures: `ptr->champ`
    - tableaux: `ptr[i]`
  - Passage de paramètre par référence
- Allocation dynamique de mémoire * allocation: `int* ptr = malloc(sizeof(int)*5);` * désallocation: `free(ptr);`

Chapter 5

Fichiers

5.1 Entrées-sorties bufferisées

- Le système[1] fournit des primitives d'entrées/sorties (E/S) bufferisées
  - permet d'accéder au contenu de fichiers
  - *bufferisées*: les E/S sont d'abord groupées en mémoire, puis exécutées sur le périphérique
  - permet d'améliorer les performances
    - exemple: 1024 écritures de 1 octet sont regroupées en une écriture de 1ko (donc gain important en performances)
  - `FILE*`: type "opaque" désignant un fichier ouvert

[1] Pour être exact, il s'agit de la bibliothèque standard (la libc)

Plus précisement, `FILE*` désigne un *flux*. Ce flux peut être un fichier, mais également des *flux standards* (`stdin`, `stdout`, ou `stderr`), des tubes, des sockets, etc.

5.1.1 Ouverture/fermeture

- `FILE* fopen(char* fichier, char* mode);`
  - `mode`: mode d'ouverture
    - `"r"` : lecture seule
    - `"w"` : écriture seule
    - `"r+"` ou `"a"` : écriture seule (ajout)
    - `"a+"` : lecture et écriture (ajout)
- `int fclose(FILE* f);` * Complète les opérations et ferme le fichier

Après appel à la fonction `fclose`, `f` (le `FILE*`) devient inutilisable : le pointeur pointe vers une zone mémoire qui a peut être été libérée par `fclose`. Il convient donc de ne plus utiliser le fichier !

5.1.2 Primitives d'écriture

- `int fprintf(FILE* f, char* format, ...);`
  - similaire à `printf`, mais écrit dans le fichier `f`
  - écrit une chaîne de caractères dans un fichier
- `size_t fwrite(void* ptr, size_t size, size_t nmemb, FILE* f);`
  - écrit les `size`×`nmemb` octets situés à l'adresse `ptr` dans `f`

5.1.3 binaire vs. ascii

Quelle est la différence entre `printf` et `fwrite` ? La fonction `fprintf` écrit un ensemble de caractères ASCII dans le fichier, alors que `fwrite` écrit un ensemble de bits.

Ainsi, pour écrire la valeur `17` dans un fichier en ASCII, on peut exécuter:

```
fprintf(f, "%d", 12);
```

Le fichier contiendra donc les octets `0x31` (le caractère `'1'`), et `0x32` (`'2'`).

Pour écrire la valeur `12` en binaire, on peut exécuter:

```
int n=12;
fwrite(&n, sizeof(int), 1, f);
```

Le fichier contiendra donc les octets `0x0C 0x00 0x00 0x00`, c'est à dire les 4 octets d'un `int` dont la valeur est `12`.

5.1.4 Sortie d'erreur

Puisqu'un `FILE*` désigne en fait un flux, on peut utiliser `fprintf` pour écrire sur la sortie standard d'erreur :

```
fprintf(stderr, "Warning: flux capacitor overlow !\n");
```

5.1.5 Exemple

Voici un programme montrant l'utilisation de primitives de lecture et d'écriture:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char**argv) {
  if (argc != 3) {
    fprintf(stderr, "USAGE = %s fichier_source fichier_destination\n", argv[0]);
    return EXIT_FAILURE;
  }

  /* open the input and output files */
  FILE*f =fopen(argv[1], "r");
  assert(f);
```

```
  FILE*f_out =fopen(argv[2], "w");
  assert(f_out);
  char line[1024];
  int lineno = 1;
  /* read the input file line by line */
  while(fgets(line, 1024, f)) {
    /* write the line to the output file */
    fprintf(f_out, "%s", line);
    lineno++;
  }

  /* close the files */
  fclose(f);
  fclose(f_out);
  return 0;
}
```

5.1.6 Primitives de lecture

- `int fscanf(FILE* f, char* format, ...);`
  - similaire à `scanf`, mais lit depuis le fichier `f`
- `size_t fread(void* ptr, size_t size, size_t nmemb, FILE* f);`
  - lit `nmemb`×`size` octets et les stocke à l'adresse `ptr`
  - retourne le nombre d'items lus
  - `fread` renvoie une valeur < `nmemb` si la fin du fichier (`EOF`) est atteinte
- `char* fgets(char* s, int size, FILE* f);`
  - lit au plus `size` caractères et les stocke dans `s`
    - arrête la lecture avant `size` si lit `n` ou `EOF`

Ces 3 fonctions sont généralement utilisées chacune dans un cas précis:

- `fscanf` est utilisée pour lire des valeurs et les stocker dans des variables. Par exemple:

```
int a;
float b;
fscanf(f, "%d\t%f\n", &a, &b);
```

- `fread` est utilisée pour charger le contenu d'une (ou de plusieurs) structure(s):

```
struct s {
  int a;
  float b;
  char c;
};
struct s tab[10];
fread(tab, sizeof(struct s), 10, f);
```

- `fgets` est utilisée pour lire un fichier ligne par ligne:

```c
char line[1024];
int lineno = 1;
while(fgets(line, 1024, f)) {
  printf("line %d: %s\n", lineno, line);
  lineno++;
}
```

5.1.7 Curseur

- Position dans le fichier à laquelle la prochaine opération aura lieu
  - Initialisé à 0 (le début du fichier) généralement
  - sauf si ouvert en mode “a” ou “a+” (dans ce cas: positionné à la fin du fichier)
- Avance à chaque opération de lecture/écriture
- `long ftell(FILE *stream);`
  - Indique la position courante (en nombre d’octets depuis le début du fichier)
- `int fseek(FILE *f, long offset, int whence);`
  - déplace le curseur de `offset` octets depuis
    - le début du fichier (si `whence` vaut `SEEK_SET`)
    - la position courante (si `whence` vaut `SEEK_CUR`)
    - la fin du fichier (si `whence` vaut `SEEK_END`)

Chapter 6

Debugging

6.1 Debugging

But: comprendre l'exécution d'un programme

- Pourquoi le programme *crash* ?
- Pourquoi le résultat est `12` ? Ca devrait être 17, non ?
- Pourquoi le programme est bloqué ?

Selon une étude[1], un développeur passe 50 % de son temps à debugger des programmes. Apprendre à debugger efficacement est donc nécessaire si vous souhaitez réduire la durée de cette activité pénible.

[1] T. Britton et al. *Reversible debugging software*. University of Cambridge-Judge Business School, 2013, Technical Report.

6.1.1 Debugging "manuel"

- On part d'un état (supposé) correct du programme
- Ajout de `printf("entrée dans foo(n=%d)\n", n);` dans le code source
- But: suivre le flot d'exécution et l'évolution des variables
- Avantages:
  - Facile à mettre en oeuvre
- Inconvénients
  - Beaucoup de code à modifier
  - Besoin de recompiler
  - Besoin de nettoyer le code après le debugging

6.2 Utilisation d'un Debugger

- Exécution du programme controllée par un debugger
- Permet:
  - d'examiner la valeur d'une variable
  - d'exécuter le programme en mode pas à pas
  - de mettre en pause le programme
  - d'insérer des *points d'arrêt*
- Exemple: GDB - The GNU Project Debugger

6.2.1 Exemple d'utilisation de GDB

```
$ ./sigsegv
Debut du programme
Erreur de segmentation

$ gdb ./sigsegv
[...]
(gdb) run
Starting program: ./sigsegv
Debut du programme

Program received signal SIGSEGV, Segmentation fault.
0x000000000040050b in main (argc=1, argv=0x7fffffffdd68) at sigsegv.c:7
7     *ptr=5;
(gdb) print ptr
$1 = (int *) 0x0
```

6.2.2 Utiliser GDB

- Pré-requis: compiler le programme avec l'option `-g`
  - Inclue les noms de fonctions/variables dans le binaire pour faciliter le débugging
- Charger le programme dans `gdb`: `gdb ./programme`
- Lancer le programme: `r` (ou `run`)
- Arrêter le programme: `Ctrl+C`
- Quitter gdb: `q` (ou `quit`)

Vous trouverez sur https://www-inf.telecom-sudparis.eu/COURS/CSC4103/Supports/?page=annexe-gdb un récapitulatif des principales commandes `gdb`.

6.2.3 Examiner l'état du programme

La commande `bt` (ou `backtrace`)

- affiche la pile d'appel
- pour chaque niveau
  - *callsite*: emplacement de l'appel de fonction
  - possibilité d'inspecter les variables locales
- sélection de la *stack frame* `#x` avec la commande `frame [x]`.

```
  (gdb) bt
#0  baz (a=2) at backtrace.c:7
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
#2  0x00000000004005ae in foo (n=4) at backtrace.c:21
#3  0x0000000000400559 in baz (a=5) at backtrace.c:9
[...]
```

La *backtrace* permet d'examiner l'état actuel du processus, ainsi que l'enchaînement d'appels de fonctions qui a mené à cet état.

```
  (gdb) bt
#0  baz (a=2) at backtrace.c:7
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
#2  0x00000000004005ae in foo (n=4) at backtrace.c:21
#3  0x0000000000400559 in baz (a=5) at backtrace.c:9
[...]
(gdb) frame
#0  baz (a=2) at backtrace.c:7
7     if(a<=2)
(gdb) print a
$1 = 2
(gdb) frame 1
#1  0x0000000000400581 in bar (n=5, m=3) at backtrace.c:15
15    return baz(m-1);
(gdb) print m
$2 = 3
```

Ici, `gdb` nous indique que le programme est arrêté dans la fonction `baz`, à la ligne 7 du fichier `backtrace.c`. Cette fonction a été appelée (`frame #1`) par la fonction `bar` à la ligne 15. La fonction `bar` a été appelée par `foo` à la ligne 31 (cf la `frame #2`).

En sélectionnant une `frame`, on peut examiner l'état des variables locales au site d'appel.

6.2.4 Etat des variables d'un processus

- commande `p [var]` (ou `print [var]`)
  - affiche la valeur de `[var]`
- commande `display [var]`
  - affiche la valeur de `[var]` à chaque arrêt du programme
- `[var]` peut être une variable (eg. `p n`)
- `[var]` peut être une expression (eg. `p ptr->next->data.n`)

Il est possible de choisir le format d'affichage:

- `p/d [var]` affiche la valeur décimale de `[var]`
- `p/u [var]` affiche la valeur décimale (non signée) de `[var]`
- `p/x [var]` affiche la valeur hexadécimale de `[var]`
- `p/o [var]` affiche la valeur octale de `[var]`
- `p/t [var]` affiche la valeur binaire de `[var]`
- `p/a [var]` affiche la valeur `[var]` sous forme d'adresse
- `p/c [var]` affiche la valeur `[var]` sous forme de caractère
- `p/f [var]` affiche la valeur `[var]` sous forme de flottant

`gdb` peut également afficher la valeur d'un registre. Par exemple `p $eax` affiche la valeur du registre `eax`.

6.2.5 Exécution pas à pas

Une fois le programme lancé, possibilité d'exécuter les instructions une par une:

- `n` (ou `next`) : exécute la prochaine instruction, puis arrête le programme.
- `s` (ou `step`) : idem. Si l'instruction est un appel de fonction, ne descend pas dans la fonction.
- `c` (ou `continue`) : continue l'exécution du programme (arrête le mode pas à pas)

6.2.6 Points d'arrêt

- *"Arrête le programme dès qu'il atteint cette ligne de code"*
  - `b fichier.c:123`
- Permet d'examiner l'état du programme à certains points (exemple: entrée de la fonction buggée)

Après avoir définis les points d'arrêt, on laisse le programme s'exécuter (avec la commande `continue`). Lorsque le programme atteint un des points d'arrêt, le débugger le met en pause et donne la main au développeur afin qu'il puisse examiner l'état du programme.

Par exemple:

```
$ gdb ./programme
[...]
(gdb) b bar
Breakpoint 1 at 0x400569: file programme.c, line 13.
(gdb) b backtrace.c:9
Breakpoint 2 at 0x40054c: file programme.c, line 9.
(gdb) r
Starting program: programme
Debut du programme

Breakpoint 1, bar (n=11, m=9) at backtrace.c:13
```

```
13    if(m<2)
(gdb) p n
$1 = 11
(gdb) p m
$2 = 9
(gdb) c
Continuing.

Breakpoint 2, baz (a=8) at backtrace.c:9
9     return foo(a-1);
(gdb) p a
$3 = 8
(gdb) c
Continuing.

Breakpoint 1, bar (n=8, m=6) at backtrace.c:13
13    if(m<2)
(gdb)
```

Il est également possible de définir des points d'arrêt conditionnels. Par exemple la commande

```
(gdb) b bar if n == 0
```

n'arrêtera l'exécution du programme en entrant dans la fonction `bar` que si `n` est égal à `0`.

6.2.7 Surveiller une variable

- *"Arrête toi dès qu'on modifie la variable [x]"*
  - `watch x`
- Permet de trouver les endroits où une variable est modifiée
  - *"Mais `ptr` ne devrait pas être `NULL` ! Qui a fait ça ?"*

Voici un exemple d'utilisation de la commande `watch`

```
$ gdb ./watch
  [...]
(gdb) watch n
 Hardware watchpoint 2: n
(gdb) c
Continuing.

Hardware watchpoint 2: n

Old value = 0
New value = 1
main (argc=1, argv=0x7fffffffdd68) at watch.c:7
7     for(i=0; i<1000; i++) {
```

```
(gdb) c
Continuing.

Hardware watchpoint 2: n

Old value = 1
New value = 2
main (argc=1, argv=0x7fffffffdd68) at watch.c:7
7     for(i=0; i<1000; i++) {
(gdb) p i
$1 = 17
[...]
```

6.3 Valgrind

- Outils de débugging et de profilage
  - Détection de fuites mémoire
  - Utilisation de variables non initialisées
- `valgrind [programme]`

Valgrind peut détecter l'utilisation de variables non initialisées. Par exemple, la non initialisation de `n` dans instructions suivantes est détectée par valgrind:

```
int n;
printf("%d$\n", n);
```

```
  $ valgrind ./exemple_valgrind
==1148== Memcheck, a memory error detector
==1148== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1148== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==1148== Command: ./exemple_valgrind
==1148==
==1148== Conditional jump or move depends on uninitialised value(s)
==1148==    at 0x4E7F2D3: vfprintf (vfprintf.c:1631)
==1148==    by 0x4E86AC8: printf (printf.c:33)
==1148==    by 0x400504: foo (exemple_valgrind.c:6)
==1148==    by 0x400529: main (exemple_valgrind.c:13)
==1148==
==1148== Use of uninitialised value of size 8
==1148==    at 0x4E7C06B: _itoa_word (_itoa.c:179)
==1148==    by 0x4E7F87C: vfprintf (vfprintf.c:1631)
==1148==    by 0x4E86AC8: printf (printf.c:33)
==1148==    by 0x400504: foo (exemple_valgrind.c:6)
==1148==    by 0x400529: main (exemple_valgrind.c:13)
==1148==
[...]
```

```
==1148==
5
==1148==
==1148== HEAP SUMMARY:
==1148==     in use at exit: 0 bytes in 0 blocks
==1148==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==1148==
==1148== All heap blocks were freed -- no leaks are possible
==1148==
==1148== For counts of detected and suppressed errors, rerun with: -v
==1148== Use --track-origins=yes to see where uninitialised values come from
==1148== ERROR SUMMARY: 8 errors from 8 contexts (suppressed: 0 from 0)
```

Valgrind détecte également les *fuites mémoire*. Lorsqu'une zone mémoire allouée avec `malloc()` n'est pas libérée (avec `free()`), la zone mémoire peut être "perdue". L'effet peut être grave si la fuite mémoire survient fréquemment. Par exemple, un serveur web qui perdrait quelques octets lors du traitement d'une requête web, pourrait perdre plusieurs gigaoctets de mémoire après le traitement de millions de requêtes.

Valgrind détecte ce type de fuites mémoire. Pour obtenir des informations sur l'origine de la fuite, on peut utiliser l'option `--leak-check=full` :

```
$ valgrind  --leak-check=full ./exemple_valgrind2
==1572== Memcheck, a memory error detector
==1572== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1572== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==1572== Command: ./exemple_valgrind2
==1572==
85823552
==1572==
==1572== HEAP SUMMARY:
==1572==     in use at exit: 1,024 bytes in 1 blocks
==1572==   total heap usage: 2 allocs, 1 frees, 2,048 bytes allocated
==1572==
==1572== 1,024 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1572==    at 0x4C2BBCF: malloc (vg_replace_malloc.c:299)
==1572==    by 0x40054E: main (exemple_valgrind2.c:7)
==1572==
==1572== LEAK SUMMARY:
==1572==    definitely lost: 1,024 bytes in 1 blocks
==1572==    indirectly lost: 0 bytes in 0 blocks
==1572==      possibly lost: 0 bytes in 0 blocks
==1572==    still reachable: 0 bytes in 0 blocks
==1572==         suppressed: 0 bytes in 0 blocks
==1572==
==1572== For counts of detected and suppressed errors, rerun with: -v
==1572== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

6.4 Pointeurs de fonction

- Toute fonction a une adresse
  - `foo` désigne l'adresse de la fonction `int foo(int a, char b);`
    - l'adresse correspond à l'endroit où est situé le code dans la mémoire
- Un pointeur de fonction[1] désigne l'adresse d'une fonction
- Exemple de déclaration:
  - `int (*function_ptr)(int a, char b) = foo;`
- Utilisation:
  - `function_ptr(12, 'f'); // appelle la fonction foo`

[1] Oui, c'est sans rapport avec le debugging, mais pour équilibrer les séances, nous avons préféré ne pas aborder cette notion lors du cours sur les pointeurs :-)

- La déclaration d'un pointeur de fonction ressemble à la déclaration du prototype d'une fonction dont le nom serait `(*nom_pointeur)`
- Comme un pointeur "normal", un pointeur de fonction peut être initialisé à `NULL`
- Comme pour un pointeur sur `int` qui ne peut pointer que sur une valeur de type `int`, un pointeur sur `int (*f)(double, char, int)` ne peut pointer que sur une fonction dont le prototype est `int f(double, char, int);`
- Une fois initialisé, un pointeur de fonction peut s'utiliser comme une fonction "normale".

Exemple:

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

double add(double a, double b) {
  return a+b;
}

double substract(double a, double b) {
  return a-b;
}

int main(int argc, char**argv) {
  double n, m;
  scanf("%lf", &n);
  scanf("%lf", &m);
  // declare a function pointer named "operation"
  double (*operation)(double, double) = NULL;

  if(n < m) {
    /* operation points to the add function */
    operation = add;
  } else {
    /* operation points to the substract function */
    operation = substract;
```

```
  }

  /* call the function pointed to by operation */
  double result = operation(n, m);

  printf("Result of the operation: %lf\n", result);

  return EXIT_SUCCESS;
}
```

On peut définir un type (à l'aide du mot-clé `typedef`) correspondant à un pointeur de fonction. Par exemple:

```
typedef double (*op_function)(double, double);
```

définit le type `op_function`. On peut donc ensuite déclarer un pointeur de fonction de ce type en faisant:

```
op_function operation;
```

- Les pointeurs de fonctions sont utilisés pour faire de composants logiciels ou des plugins. Une interface est définie, par exemple:
  - il faut une fonction qui initialise la structure x
  - il faut une fonction qui calcule la structure x
  - il faut une fonction qui affiche la structure x Cela prend généralement la forme d'une structure contenant des pointeurs de fonction nommés *callbacks*:

```
struct component {
  char* plugin_name;
  void (*init_value)(struct value*v);
  void (*compute_value)(struct value*v);
  void (*print_value)(struct value*v);
};
```

Un plugin implémentant ce service allouera la structure et désignera ses fonctions comme callback pour le service.

Chapter 7

Processus

7.1 Caractéristiques d'un processus

- **PID** (*Process IDentifier*) : identifiant unique du processus
  - `pid_t getpid();`
  - retourne le PID du processus courant
- **PPID** (*Parent PID*) : identifiant du processus père
  - `pid_t getppid();`
  - retourne le PPID du processus courant

Voici un exemple de programme affichant son PID et son PPID:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char**argv) {
  printf("Current process ID: %d\n", getpid());
  printf("Current parent process ID: %d\n", getppid());
  return EXIT_SUCCESS;
}
```

Ce programme donne pour résultat:

```
$ ./print_id
Current process ID: 17174
Current parent process ID: 25275
```

Lorsque le processus parent (`P1`) d'un processus (`P2`) meurt, le processus fils est rattaché au processus au processus initial de PID 1. Le PPID de `P2` devient donc 1.

7.2 Création de processus

- `int system(const char* cmd);`
  - Crée un processus shell qui exécute `cmd`
  - Retourne le code de retour de la commande (0 si tout s'est bien passé)

Voici un exemple de programme utilisant la fonction `system`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char**argv) {
  int ret_val;
  char* cmd="ps f";
  printf("Running command '%s'\n", cmd);
  printf("-----------\n");
  ret_val = system(cmd);
  printf("-----------\n");
  printf("system returned %d\n", ret_val);
  return EXIT_SUCCESS;
}
```

Ce programme donne pour résultat:

```
$ ./system
Running command 'ps f'
-----------
  PID TTY      STAT   TIME COMMAND
16847 pts/1    Ss+    0:00 bash
17076 pts/1    S      0:01  _ okular chap.pdf
 8199 pts/0    Ss+    0:00 bash
25275 pts/7    Ss     0:00 bash
 8174 pts/7    Sl     0:14  _ emacs contenu.tex
17540 pts/7    S+     0:00  _ ./system
17541 pts/7    S+     0:00      _ sh -c ps f
17542 pts/7    R+     0:00          _ ps f
-----------
system returned 0
```

7.2.1 fork

- `pid_t fork();`
- duplique le processus courant
  - duplication de la mémoire, des fichiers ouverts, etc.
- le processus créé est le fils du processus courant
- le processus fils est une copie exacte du père sauf

  - le PID du fils est différent
  - le PPID du fils est le PID du père
- seule la valeur retournée par fork permet de différencier le père et le fils
  - le père reçoit le PID du fils
  - le fils reçoit 0

Voici un exemple d'utilisation de `fork`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char**argv) {
  printf("I am process %d. My PPID: %d\n", getpid(), getppid());

  pid_t ret_val = fork();
  if(ret_val == 0) {
    printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1);
  } else if (ret_val>0) {
    printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
  } else {
    printf("Fork failed\n");
  }
  return EXIT_SUCCESS;
}
```

Lors du `fork`, le processus est intégralement dupliqué. Le processus fils possède donc la même mémoire que le processus père: on retrouve donc les variables affectées dans le processus père avant le fork.

Après le fork, les espaces mémoire des deux processus deviennent dissociés: si le processus fils (resp. le père) modifie la variable `value`, il ne modifie que sa copie de la variable, et le processus père (resp. le fils) ne voit pas la modification. Par exemple, le programme suivant:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char**argv) {
  int value = 1;
  printf("[%d] Before forking. value = %d\n", getpid(), value);
  pid_t ret_val = fork();
  if(ret_val == 0) {
    value++;
    printf("[%d] After forking. value = %d\n", getpid(), value);
  } else if (ret_val>0) {
    sleep(1);
    printf("[%d] After forking. value = %d\n", getpid(), value);
```

```
  } else {
    printf("Fork failed\n");
  }
  return EXIT_SUCCESS;
}
```

donnera, lors de son exécution, le résultat suivant:

```
$ ./fork_variable
[13229] Before forking. value = 1
[13230] After forking. value = 2
[13229] After forking. value = 1
```

7.2.2 La classe de fonctions `exec`

- ensemble de fonctions permettant d'exécuter une commande
  - `execlp`, `execvp`, `execve`, `execle`, `execlp`, etc.
- remplace le programme appelant par un nouveau
  - donc, on ne "sort" jamais de la fonction (sauf erreur)

Voici un exemple d'utilisation de `execlp`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char**argv) {
  printf("I am process %d. My PPID: %d\n", getpid(), getppid());

  pid_t ret_val = fork();
  if(ret_val == 0) {
    printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
    execlp("ps", "ps", "-l", NULL);
    printf("This is printed only if execlp fails\n");
    abort();
  } else if (ret_val>0) {
    printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1);
  }

  return EXIT_SUCCESS;
}
```

`exec` est une famille de fonction permettant de remplacer l'image du processus courant: l'ensemble de l'espace mémoire est effacé et remplacé par l'image du procegramme exécuté.

Les paramètres peuvent être passés sous la forme d'une liste de paramètres (avec les fonctions `execl*`), ou sous la forme d'un tableau de chaînes de caractères (avec les fonctions `execv*`).

7.2.3 Terminaison d'un processus

- `pid_t wait(int *status);`
  - Attend la terminaison d'un processus fils
  - Le champs `status` permet de connaître la cause du décès.
- Variante: `pid_t waitpid(pid_t pid, int *wstatus, int options);`

Voici un exemple d'utilisation de la fonction `wait` :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char**argv) {
  printf("I am process %d. My PPID: %d\n", getpid(), getppid());

  pid_t ret_val = fork();
  if(ret_val == 0) {
    printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1);
    return 17;
  } else if (ret_val>0) {
    printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
    int status;
    pid_t pid = wait(&status);

    int exit_status = WEXITSTATUS(status);
    printf("The child process %d ended with exit status %d\n", pid, exit_status);

  } else {
    printf("Fork failed\n");
  }
  return EXIT_SUCCESS;
}
```

La fonction `wait` retourne le PID du processus fils qui s'est terminé et remplit la variable entière `status`. Cette variable peut ensuite être utilisée pour obtenir des informations sur la terminaison du processus fils.

Par exemple, la macro `WEXITSTATUS(status)` retourne le code de retour du processus fils. Ce programme donnera donc :

```
$ ./exemple_wait
I am process 21088. My PPID: 20960
I'm the parent process. PID=21088, PPID=20960
I'm the child process. PID=21089, PPID=21088
```

```
The child process 21089 ended with exit status 17
```

La fonction `waitpid` est une variante de `wait`. Elle permet d'attendre un processus fils précis, ou de tester (sans bloquer le processus appelant) la terminaison d'un processus.

Chapter 8

Appels systèmes

8.1 Qu'est ce qu'un système d'exploitation ?

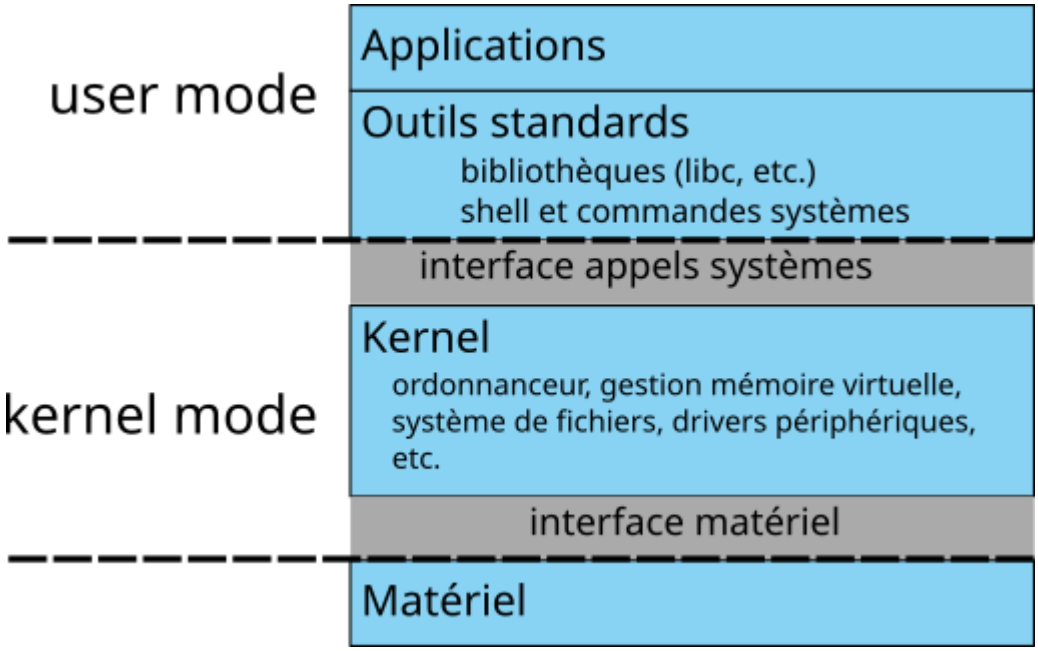
Rôles d'un système d'exploitation:

- Abstraire le matériel pour le programmeur
  - Cacher la complexité du matériel
  - Fournir une interface virtuelle de la machine
- Protéger
  - Protection entre utilisateurs (droits d'accès aux fichiers, espaces mémoires des processus séparés)
  - Protection du matériel
- Partager les ressources
  - Partage du CPU (ordonnancement des processus)
  - Accès concurrents à un périphérique

8.1.1 User mode vs. Kernel mode

Cloisonnement entre le mode utilisateur et le mode noyau

- *User mode*
  - certaines instructions sont interdites
  - pas d'accès aux périphériques
  - accès à l'espace mémoire virtuel du processus
- *Kernel mode*
  - accès aux périphériques
  - accès à la mémoire physique



8.2 Comment passer en mode noyau ?

2 méthodes:

- interruption
  - interruption logicielle
    - Générée par le processeur en exécutant une instruction
    - division par zéro, accès mémoire illicite
  - interruption générée par le matériel (IRQ)
    - “Je viens de recevoir un message.” – la carte réseau
    - “J’ai fini de copier les données sur le disque dur.” – le moteur DMA
- appel système
  - l’utilisateur demande à l’OS un service

Lorsqu’une interruption est reçue, le processeur suspend l’exécution du thread, bascule en mode noyau, et appelle la routine traitant l’interruption. Lorsque la routine se termine, le processeur rebascule en mode utilisateur est reprend l’exécution du thread.

Un appel système consiste à appeler une fonction exécuté en mode noyau. Le passage du mode utilisateur au mode noyau peut se faire en générant une interruption logicielle particulière (par exemple sur les processeurs ARM ou les processeurs x86 32 bits), ou en exécutant une instruction particulière (par exemple, l’instruction `syscall` sur les processeurs x86 64 bits) qui a un effet équivalent. Le kernel exécute alors la fonction correspondant au numéro de l’appel système demandé. Lorsque l’appel système se termine, on sort du traitant d’interruption, et le processeur rebascule en

mode utilisateur.

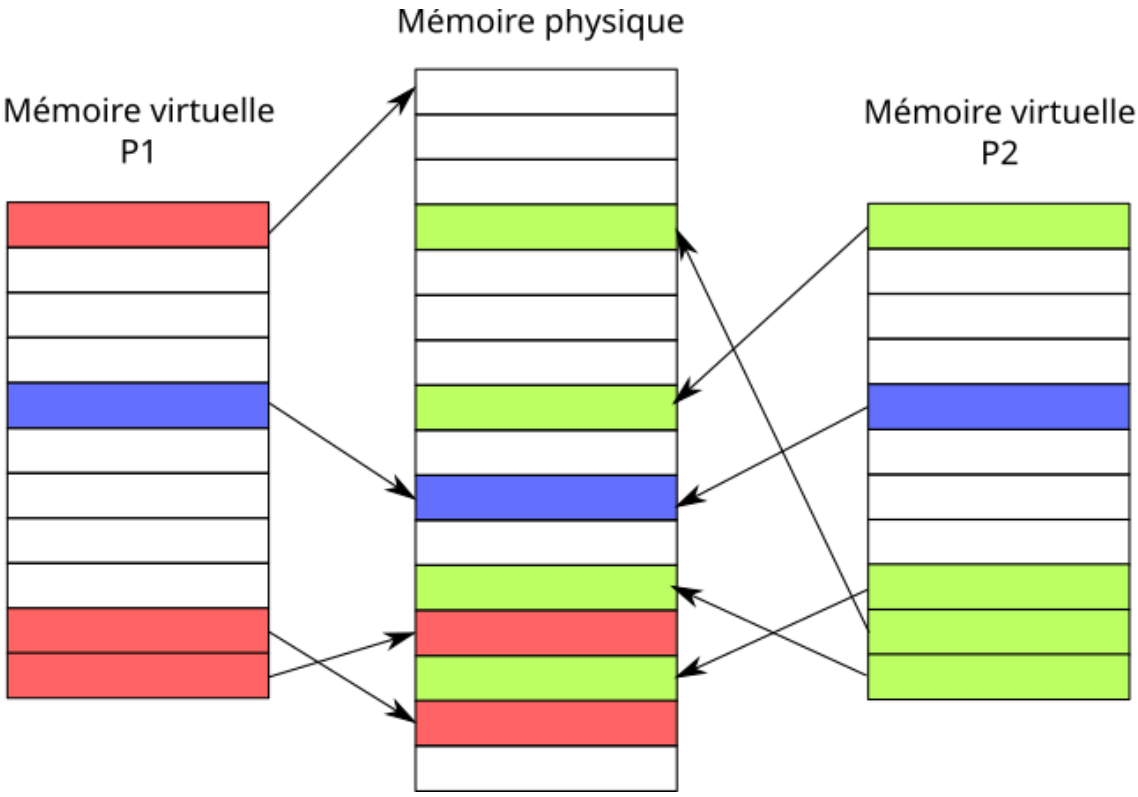
8.2.1 Observer les appels systèmes

La commande `strace` intercepte et affiche les appels systèmes d'un programme:

```
$ strace echo "coucou"
execve("/bin/echo", ["echo", "coucou"], [/* 54 vars */]) = 0
brk(NULL)                               = 0x25d2000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f619cc01000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("tls/x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("libc.so.6", O_RDONLY|O_CLOEXEC)   = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[...]
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
write(1, "coucou\n", 7coucou
)                 = 7
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

8.2.2 Gestion de la mémoire

- mémoire virtuelle des processus découpées en *pages*
- mémoire physique (RAM) découpée en *cadres de pages*
- *pages* projetées sur des *cadres de pages*



Certains cadres de pages sont référencés par plusieurs processus. Cela est possible par exemple si les processus ne font que des accès en lecture à la page. Il s'agit typiquement du code des bibliothèques partagées (`libc.so` par exemple) qui peuvent être chargées par plusieurs processus.

8.2.3 Primitives de synchronisation: les sémaphores

Sémaphore:

- distributeur de "*jetons*"
- 2 opérations:
  - **P** (*"Puis-je"*): prendre un jeton (et attendre si pas de jeton)
  - **V** (*"Vas-y"*): ajouter un jeton (et débloquer un processus)
- Exemple d'utilisation: exclusion mutuelle entre processus

8.2.3.1 Sémaphore: mise en oeuvre

- Création: `sem_open("/CLE", O_CREAT, S_IRWXU, nb_jetons);`
  - retourne un `sem_t*`
  - CLE est une chaîne commençant par `/`

- Ouverture: `sem_open("/CLE", 0);`
  - retourne un `sem_t`
- Destruction : `sem_unlink(const char *name)`
- Opération **P** : `sem_wait(sem_t* sem)`
- Opération **V** : `sem_post(sem_t* sem)`

L'opération `sem_wait` est bloquante tant qu'il n'y a pas de jeton. L'opération `sem_post` est, elle, non-bloquante, et permet de débloquer un des processus en attente d'un jeton.

Voici un exemple d'utilisation d'un sémaphore. Le progamme `exemple_sem_init` crée un sémaphore, qui peut ensuite être utilisé par le programme `exemple_sem`. Le programme `exemple_sem_unlink` détruit un sémaphore.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char**argv) {
  sem_t *sem;

  if (argc != 3) {
    fprintf(stderr, "USAGE = %s cle valeur\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  char*cle=argv[1];
  int valeur = atoi(argv[2]);

  /* Creation et initialisation du semaphore */
  sem = sem_open(cle, O_CREAT, S_IRWXU, valeur);
  if (sem == SEM_FAILED) {
    perror("sem_open failed");
    exit(EXIT_FAILURE);
  }

  printf("Initialisation OK\n");
  int sval = -1;
  /* recupere le nombre de jetons */
  if(sem_getvalue(sem, &sval) <0) {
    perror("sem_getvalue failed");
    exit(EXIT_FAILURE);
  }
  printf("sval = %d\n", sval);

  return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>

int main(int argc, char**argv) {
  sem_t *sem;

  if (argc != 2) {
    fprintf(stderr, "USAGE = %s cle\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  char*cle=argv[1];

  /* Creation et initialisation du semaphore */
  sem = sem_open(cle, 0);
  if (sem == SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
  }

  printf("Ouverture OK\n");

  printf("On prend un jeton...\n");
  sem_wait(sem);
  printf("Jeton obtenu.\n");
  sleep(5);

  printf("On relache le jeton\n");
  sem_post(sem);
  printf("Jeton relache\n");

  return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char**argv) {
  sem_t *sem;

  if (argc != 2) {
    fprintf(stderr, "USAGE = %s cle\n", argv[0]);
```

```
    exit(EXIT_FAILURE);
  }

  char*cle=argv[1];

  if(sem_unlink(cle) < 0){
    perror("sem unlink failed");
    abort();
  }

  return EXIT_SUCCESS;
}
```

Chapter 9

Signaux

9.1 Signaux

Rappel (CSC3102)

- Signal: mécanisme de communication inter-processus
- Message: un entier entre 1 et 31
- Ordre de réception aléatoire (différent de l'ordre d'émission)
- Une routine de réception est automatiquement invoquée chez le récepteur dès que le signal arrive

---

9.1.1 Envoyer un signal

- `int kill(pid_t pid, int sig);`
  - Envoie le signal `sig` au processus `pid`
  - Quelle valeur pour `sig`?
    - valeur entière (par ex: `9`): pas portable (dépend de l'OS)
    - constante (par ex: `SIGKILL`) définie dans `signal.h`

Voici un exemple de programme utilisant la fonction `kill`:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>

int main(int argc, char**argv) {
  if(argc != 2) {
    fprintf(stderr, "usage: %s PID\n", argv[0]);
    return EXIT_FAILURE;
  }
```

```
  pid_t pid = atoi(argv[1]);
  int signo = SIGKILL;

  printf("Sending signal %d to %d\n", signo, pid);
  kill(pid, signo);

  return EXIT_SUCCESS;
}
```

9.1.2 Recevoir un signal

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Spécifie le comportant lors de la réception du signal `signum`
  - `struct sigaction` est une structure de la forme:

```
struct sigaction {
        void     (*sa_handler)(int); // pointeur sur la fonction à appeler
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

9.1.3 Signaux interceptables

Il est possible d'utiliser `sigaction` pour "intercepter" tout signal sauf les signaux `SIGKILL` et `SIGSTOP`

9.1.4 `struct sigaction`

La valeur prise par `sa_handler` est: * l'adresse d'une fonction (par ex: `void signal_handler(int signo)` * Le paramètre `signo` est le numéro du signal reçu * la valeur `SIG_DFL` pour restaurer l'action par défaut (tuer le processus) * la valeur `SIG_IGN` pour ignorer le signal: à la réception de ce signal, aucune action ne sera effectuée

Sauf cas d'usages particuliers, les autres champs de la structure `sigaction` sont à mettre à `0`.

9.1.5 `oldact`

La fonction `sigaction` modifie le comportant du processus lorsqu'il reçoit le signal `signum`. Si `oldact` n'est pas `NULL`, l'ancien comportement y est stocké.

9.1.6 Variables globales

Si la fonction traitant le signal manipule des variables globales, il est conseillé de les déclarer `volatile`. Par exemple:

```
volatile int var;
```

Lorsqu'une variable est déclarée `volatile`, le compilateur limite les optimisations faites sur cette variable. Par exemple, le compilateur ne met pas en cache (dans un registre) la variable.

Si une fonction (par exemple `foo`) qui manipule la variable `var` non `volatile` est interrompue par un traitant de signal (`sig_handler`) modifiant `var`, la modification de la variable risque de ne pas être "vue" par `foo` qui travaille sur une copie en cache de la variable. La fonction `foo` risque donc de travailler sur une version obsolète de la variable.

9.1.7 Exemple

Voici un exemple de programme utilisant `sigaction` pour intercepter un signal:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>

/* Function to call upon signal reception */
void signal_handler(int signo) {
  printf("Received: signal %d\n", signo);
}

int main(int argc, char**argv) {
  if(argc != 2) {
    fprintf(stderr, "usage: %s signo\n", argv[0]);
    return EXIT_FAILURE;
  }

  /* Initialize the sigaction structure */
  int signo = atoi(argv[1]);
  struct sigaction s;
  s.sa_handler = signal_handler;

  /* Install the signal handler */
  printf("Installing signal handler for signal %d\n", signo);
  int retval = sigaction(signo, &s, NULL);
  if(retval<0) {
    perror("sigaction failed");
    abort();
  }
```

```
  /* Wait to receive signals */
  while(1) {
    printf("[%d] Sleeping...\n", getpid());
    sleep(1);
  }

  return EXIT_SUCCESS;
}
```

9.1.8 Attendre un signal

- `int pause( );`
  - Attend qu'un signal (non ignoré) soit reçu

9.1.9 Programmer une alarme

- `int alarm(unsigned int s);`
  - Programme l'envoi de `SIGALRM` après `s` secondes

Pour programmer une alarme avec une granularité plus fine, utilisez la fonction `setitimer`. Cette fonction permet de programmer des alarmes périodiques (qui se répètent) avec une granularité de l'ordre de la microseconde.

Chapter 10

Bibliography

Chapter 11

Annexes

11.1 Les 15 commandes à connaître pour tout padawan

11.1.1 Lancer/quitter GDB

Commande	Description
`gdb [prog]`	Charge le programme `[prog]` dans GDB.
`gdb --args [prog] [arg1 arg2 ...]`	Charge le programme `[prog]` dans GDB avec les paramètres `[arg1 arg2 ...]`
`gdb [prog] [core-file]`	Charge le core-dump du programme `[prog]` dans GDB.
`q` ou `quit`	Quitte GDB.

11.1.2 Lancer l'exécution d'un programme

Commande	Description
`r` ou `run`	Lance l'exécution du programme chargé dans gdb.
`r [arg1 arg2 ...]`	Lance l'exécution du programme chargé dans gdb avec les paramètres `[arg1 arg2 ...]`.
`set [args arg1 arg2 ...]`	Sélectionne la liste des arguments (`[arg1 arg2 ...]`) pour le prochain programme à démarrer.

11.1.3 Examiner l'état d'un processus

Commande	Description
`p [var]` ou `print [var]`	Affiche la valeur de la variable `[var]`.
`p/x [var]`	Affiche la valeur hexadécimale de la variable `[var]`.

Commande	Description
`display [var]`	Affiche la valeur de la variable `[var]` à chaque arrêt du programme.
`bt` ou `backtrace`	Affiche la pile d'appel.
`frame`	Affiche la stack frame courante.
`frame [x]`	Sélectionne la stack frame `[x]`.
`l` ou `list`	Affiche la portion de code autour de la stack frame sélectionnée.

11.1.4 Surveiller l'exécution d'un processus

Commande	Description
`b [pos]` ou `break [pos]`	Positionne un breakpoint à l'endroit `[pos]`. `[pos]` peut être un nom de fonction, un numéro de ligne (du fichier courant), un nom de fichier+numéro de ligne (`break mon_fichier:57`), etc.
`clear [pos]`	Supprime le breakpoint positionné à l'endroit `[pos]`.
`d [num]` ou `delete [num]`	Supprime le breakpoint numéro `[num]`.
`w [var]` ou `watch [var]`	Surveille l'état de la variable `[var]`.

11.1.5 Contrôler l'exécution d'un processus

Commande	Description
`n` ou `next`	Avance d'un pas.
`s` ou `step`	Avance d'un pas. Si l'instruction à exécuter est un appel de fonction, ne descend pas dans la fonction.
`c` ou `continue`	Continue l'exécution jusqu'au prochain breakpoint.

11.2 Pour devenir un maître jedi de GDB

11.2.1 Assembleur

Commande	Description
`disassemble [function]`	Affiche le code assembleur de la fonction `[function]`.
`info registers`	Affiche la valeur des registres.
`p \$[register]`	Affiche la valeur d'un registre. Exemple: `print \$eax`.

11.2.2 Reverse debugging

Commande	Description
`record`	Démarre l'enregistrement du comportement du processus.
`record stop`	Arrête l'enregistrement.
`rs` ou `reverse-step`	Comme step, mais en arrière.
`rn` ou `reverse-next`	Comme next, mais en arrière.
`rc` ou `reverse-continue`	Comme continue, mais en arrière.
`set can-use-hw-watchpoints 0`	Permet d'utiliser des watchpoints en reverse debugging.

11.2.3 Autre

Commande	Description
`break [pos] if [cond]`	Arrête le processus à `[pos]` si la condition `[cond]` est vérifiée.
`up`	Remonte d'une stack frame dans la pile.
`down`	Descend d'une stack frame dans la pile.
`attach [pid]`	Attache GDB au processus de pid `[pid]`.
`detach`	Détache le processus de GDB.
`set env [var]=[value]`	Affecte la valeur `[value]` à la variable d'environnment `[var]`.

11.3 Exemple de fichier Makefile

```
# In order to reuse this Makefile
#    Modify the project name => modify the binary name
#    Add all the .o dependencies in OBJ
#    That's all folk :)

PROJECT=main

BIN=$(PROJECT)
OBJ=main.o

# compilation flags
CFLAGS=-Wall -Werror -g
# link flags
LDFLAGS=
CC=gcc

Echo=@echo [$(PROJECT)]:
```

```
ifndef VERBOSE
  Verb := @
endif

# Tells make that 'all' and 'clean' are "virtual" targets (that does not
# generate a file)
.PHONY: all clean

all: $(BIN)

$(BIN): $(OBJ)
 $(Echo) Linking $@
 $(Verb) $(CC) $(LDFLAGS) -o $@ $^

# generic rule: to generate foo.o, we need foo.c
%.o: %.c
 $(Echo) Compiling $<
 $(Verb) $(CC) $(CFLAGS) -c "$<" -o "$@"

# you can add specific compilation rules here


# you can invoke "make clean" to delete all the generated files
clean:
 $(Echo) Cleaning compilation files
 $(Verb) rm -f $(OBJ) $(BIN)
```
