Système d'exploitation

L2 Informatique – UVSQ

Sebastien GOUGEAUD

pro.seb.gougeaud@gmail.com

Avant-propos

Liste des topics abordés

1. Rappels C/Shell
2. Systèmes de fichiers
3. Processus
4. Threads
5. Communication
6. Mémoire
7. Ordonnancement
8. *HPC*

Évaluation

- 60% examen, 40% contrôle continu
- Examen :
  - traditionnel : séries d'exercices portant sur questions de cours, exécution d'algorithmes et écriture de code
- Contrôle continu : 60% TD, 40% CM
  - TD : 3 exercices de TD répartis sur le semestre, commencés durant la séance de TD et à rendre dans la semaine qui suit
  - CM : 1 QCM d'1h, corrigé dans la demi-heure de cours restante

Travaux dirigés

Chaque groupe a sa salle :

- Gr.1 : G207, le mercredi à 9h40
- Gr.2 : D122, le mercredi à 9h40
- Gr.3 : G107, le mardi à 9h40 (+ DL BI)
- Gr.4 : Jungle, le mardi à 9h40

Les TDs notés se font en binôme intra-groupe, mais peuvent changer entre chaque TP

Rappels C

Pourquoi ce rappel ?

- Le C est le langage utilisé dans ce cours
- Pour vous rafraîchir la mémoire sur ce que vous savez déjà
- Pour vous introduire ce que vous ne connaissez pas encore et ce qui sera utilisé dans ce cours

- Le premier TD vous permet de vous refaire la main sur les mécanismes de base du C, il vous permettra de voir où sont vos lacunes pour pouvoir les combler

Liste des thèmes abordés

1. Variables
2. Blocs de contrôle
3. Fonctions
4. Structures
5. Pointeurs
6. Pointeurs de fonction
7. Tableaux
8. Directives de pré-compilation
9. ’Bien coder ?’

Variables

```
1  char *s = "Hello world!";
2  double f = 42.01;
3  int a = 7;
4
5  printf("%d - %f - '%s'\n",
6         a, f, s);
```

- Déclaration sous la forme `type nom;`
- Types basiques : `int`, `float`, `double`, `char`
- Modificateurs : `unsigned`, `short`, `long`

Blocs de contrôle

```c
const int N = 100;
int tot = 0;
int i = 0;

if (N < 0) {
    printf("Total = 0\n");
} else {
    while (i != N) {
        tot += i;
        ++i;
    }

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

Structures conditionnelles :

- `if .. else if .. else`
- opérateur ternaire
  `test ? true : false`
- `switch`

Structures itératives :

- `while`
- `for`
- `do .. while`

Fonctions

```
1  int _topic_3(const char *s,
2               const char delim) {
3      int res = 0;
4      int i;
5
6      for (i = 0;
7           i < strlen(s) &&
8               s[i] != delim;
9           ++i)
10         ++res;
11
12     return res;
13 }
```

- Déclaration sous la forme `type nom (parametres)`
- Définition avec un bloc d'instructions (entre accolades)
- Utilisation de `return` pour sortir de la fonction

Structures de données

```c
struct cplx {
    double reel;
    double imgn;
};

struct cplx r;

r.reel = a.reel + b.reel;
r.imgn = a.imgn + b.imgn;

printf("(%f,%f)\n", r.reel, r.imgn);
```

- Structures, énumérations et unions
- Utilisation de `typedef` pour créer un nouveau type à partir d'un autre (renommage)
- Initialisation de la forme `c = {.5, 2};`

Pointeurs

Variable contenant l'adresse d'une autre variable

```
1  void _topic_5(struct cplx a,
2                struct cplx b,
3                struct cplx *r) {
4      if (r == NULL)
5          return;
6
7      r->reel = a.reel + b.reel;
8      r->imgn = a.imgn + b.imgn;
9  }
```

- Si donné en paramètre de fonction, permet de modifier le contenu de la cible
- Adresse d'une variable
  `p = &v;`
- Valeur de la cible d'un pointeur
  `v = *p;`
- Structure :
  `p->f` $\leftrightarrow$ `(*p).f`

Pointeurs de fonction

```c
void dire_oui(const int id)
    { printf("Oui %d\n", id); }
void dire_non(const int id)
    { printf("Non %d\n", id); }

void topic_6() {
    void (*f_pair)(const int) =
        dire_oui;
    void (*f_impr)(const int) =
        dire_non;
    int i;

    for (i = 0; i < 10; ++i)
        if (i % 2)
            f_impr(i);
        else
            f_pair(i);
}
```

- Déclaration sous la forme `type (*nom)(param);`
- Manipulation de la fonction en tant que variable (en paramètre de fonction, dans une structure ou dans un tableau)
- Utilisation pour faire de la spécification (entre autres)

Tableaux

```
1  void (*func[2])(const int) =
2      { dire_oui, dire_non };
3  int *tab;
4  int i;
5
6  tab = malloc(10 * sizeof(*tab));
7  for (i = 0; i < 10; ++i)
8      tab[i] = i%2;
9
10 for (i = 0; i < 10; ++i)
11     func[tab[i]](i);
12
13 free(tab);
```

- Tableaux fixes (pile/*stack*)
- Taille connue à la compilation
- Tableaux dynamiques (tas/*heap*)
- Taille définie à l'exécution
- Utilisation de `malloc()` et `free()`
- `T[i]` $\leftrightarrow$ `*(T+i)`

Directives de pré-compilation

```
1  #define N 30
2      int tab[N];
3
4  #ifndef N
5      printf("N n'est pas défini\n");
6  #elif N < 1
7      printf("Pas d'elements\n");
8  #else
9      int i;
10
11     for (i = 0; i < N; ++i)
12         if (i)
13             tab[i] = i + tab[i - 1];
14         else
15             tab[i] = i;
16
17     printf("Allocation effectuee\n");
18 #endif
```

- Résolution des directives avant la compilation
- Création d'alias
- Ajout conditionnel d'instructions
- Autres...

Quelques bonnes pratiques de développeur

- Indenter
- Donner des noms de variables/fonctions clairs
- Éviter les fonctions trop longues (>50 lignes)
- Éviter les duplications de code
- Commenter le code, à minima les structures de données, les fonctions et les parties de code complexes

Rappels Shell

Qu’est-ce que le Shell ?

Interface utilisateur de base avec l’ordinateur – basée sur l’exécution de commandes données au clavier

```
$ ls ~
```

Utilisateur et administrateur

Deux types d'acteur utilisant le Shell :

- Utilisateur (user) – utilisation des programmes contenus dans /*/bin, manipulation des dossiers/fichiers utilisateur (~), etc.
- Administrateur (sudoer) – utilisation des programmes contenus dans /*/sbin, manipulation des dossiers/fichiers système, manipulation directe des périphériques, etc.

Redirection de flux – concept

- Redirection de l'entrée ou de la sortie standard par défaut vers un fichier ou un périphérique

```
$ ls ~ >fichier
$ sort <fichier1 >fichier2
```

- Utilisation de la sortie standard d'un processus comme entrée standard d'un autre processus

```
$ ls ~ | grep .txt
```

Redirection de flux – exemple

Fusion des deux types de redirection

```
$ cat fichier1 fichier2 | sort >/dev/lp[a]
```

[a] fichier vers un Kernel Printer Device

Script Shell

- Regroupement de commandes au sein d'un fichier
- Exécution du script comme s'il s'agissait d'un programme :

```
$ ./script.sh
```

Commande man

- Affichage de la page de documentation correspondant à la commande entrée en argument

```
$ man ls
```

- Plusieurs catégories de documentation :
  1. Commandes Unix
  2. Appels systèmes
  3. Fonctions de la bibliothèque standard

Commande gcc

- Compilation d'un (ou plusieurs) fichier(s) écrit(s) en langage C

```
$ gcc -o prog main.c
```

- Plusieurs options de compilation et *linkage* disponibles

Commande gdb

- Exécution en mode *debug* d'un programme

```
$ gdb ./prog
```

- Utilisation de commandes pour naviguer dans le programme durant l'exécution : *break*, *run*, *backtrace*, *up*/*down*, *print*, *next*, *continue*
- Informations de *debug* injectées dans le programme à l'aide d'une option de compilation :

```
$ gcc -g -o prog main.c
```

Introduction au système d’exploitation

Qu’est-ce qu’un système d’exploitation ?

$\rightarrow$ Couche logicielle faisant le pont entre les applications et le matériel

Deux rôles :

- Masquer la complexité du matériel géré
- Gérer les ressources disponibles et les faire fonctionner ensemble de manière sure et équitable

Exemples de ressources

- Processeur (CPU) et registres
- Mémoire (RAM, cache)
- Entrée/sortie (disque, imprimante, clavier, écran)
- Processus
- Autres

Système de fichiers

Pourquoi ?

→ Pouvoir stocker des informations de façon pérenne, de manière organisée et abstraite

Besoins :

- Stockage de grande quantité
- Conservation
- Accès simultané

Qu’est ce qu’un système de fichiers ?

→ Partie du système d’exploitation gérant les fichiers

Dans un système UNIX, tout est fichier :

- Fichiers
- Répertoires
- Liens symboliques
- Autres (FIFO, etc.)

Qu'est-ce qu'un fichier ?

→ Suite d'octets (binaire, ASCII, etc.) caractérisée par :

- Nom de fichier – label et extension (optionnel sous UNIX)

```
$ cat fichier-test.txt
```

- Chemin – emplacement dans la hiérarchie (absolu ou relatif)

```
$ ls /home/user1/doc
```

- Inode – noeud d'informations entre le système de fichiers et le périphérique
- Méta-données – attributs, ex : créateur, permissions d'accès, etc.

Appels systèmes – fichiers

```
1  int open(const char *pathname,
2           int flags);
3  int open(const char *pathname,
4           int flags, mode_t mode);
5  int close(int fd);
6  int unlink(const char *pathname);
7
8  ssize_t read(int fd, void *buf,
9               size_t count);
10 ssize_t write(int fd, void *buf,
11              size_t count);
12
13 off_t lseek(int fd, off_t offset,
14             int whence);
```

- Création, ouverture, fermeture et suppression
- Lecture et écriture
- Positionnement

Fonction `createFile()`

```c
void createFile(const char *fname, const int size, const char *data) {
    int fd = -1;
    ssize_t sz;
    int rc;

    fd = open(fname, O_CREAT | O_WRONLY, 0666);
    if (fd == -1) {
        fprintf(stderr, "ERR on file creation: %s\n", strerror(errno));
        return;
    }

    if (size) {
        sz = write(fd, data, size);
        if (sz != size)
            fprintf(stderr, "ERR on file writing: %s\n", strerror(errno));
    }

    rc = close(fd);
    if (rc)
        fprintf(stderr, "ERR on file closure: %s\n", strerror(errno));
}
```

Appels systèmes – répertoires

```
1  int mkdir(const char *pathname,
2            mode_t mode);
3  DIR *opendir(const char* name);
4  int closedir(DIR *dirp);
5  int rmdir(const char *pathname);
6
7  struct dirent *readdir(DIR *dirp);
8
9  void rewinddir(DIR *dirp);
10 long telldir(DIR *dirp);
11 void seekdir(DIR *dirp, long loc);
```

- Création, ouverture, fermeture et suppression
- Lecture
- Positionnement

Fonction `createDir()` 1/2

```
1  int createDir(const char *dname, const int nbEmptyFiles,
2                char ***dirFiles) {
3      int rc;
4      int i;
5
6      rc = mkdir(dname, 0700);
7      if (rc) {
8          fprintf(stderr, "ERR on dir creation: %s\n", strerror(errno));
9          return 1;
10     }
11
12     *dirFiles = malloc(nbEmptyFiles * sizeof(char **));
13     if (nbEmptyFiles && !*dirFiles) {
14         fprintf(stderr, "ERR on dir file names allocation: %s\n",
15                 strerror(errno));
16         return 0;
17     }
```

Fonction `createDir()` 2/2

```
1     for (i = 0; i < nbEmptyFiles; ++i) {
2         char *fname;
3         int fd;
4
5         fname = malloc(16 + strlen(dname));
6         if (!fname) {
7             fprintf(stderr, "ERR on file name allocation");
8             return 1;
9         }
10        snprintf(fname, 16 + strlen(dname), "%s/empty_XXXXXX", dname);
11        fd = mkstemp(fname);
12        if (fd == -1) {
13            fprintf(stderr, "ERR on file name creation (%s): %s\n",
14                    fname, strerror(errno));
15            continue;
16        }
17        close(fd);
18        (*dirFiles)[i] = fname;
19    }
20    return 0;
21 }
```

Gestion d'erreur incomplète en l.8 et l.15, discutée en cours

Fonction deleteDir()

```
1  void deleteDir(const char *dname, const int nbEmptyFiles,
2                 char **dirFiles) {
3      int i;
4
5      for (i = 0; i < nbEmptyFiles; ++i) {
6          unlink(dirFiles[i]);
7          free(dirFiles[i]);
8      }
9
10     free(dirFiles);
11     rmdir(dname);
12 }
```