Introduction, outils de base, premiers appels système

©A. Dragut
Université Aix-Marseille I.U.T.d'Aix en Provence - Département Informatique
Dernière mise à jour :27/09/2012

Compilation, outil `make`, `makefile`

Attention: à la fin de ce TP vous devrez rendre une feuille avec des réponses à des questions du TP.

1. Avant-propos -- Outil make -- nécessaire pour la compilation séparée

Lorsque le code source d'un projet de programmation se trouve dans plusieurs fichiers, on doit pouvoir les compiler séparément. De même, si on en modifie un ou deux, on ne doit recompiler que ceux qui en dépendent. L'outil make permet de faire ceci de manière simple et flexible: on lui donne dans un fichier spécial la commande pour compiler chaque fichier source, et quand on doit le recompiler. Donc, avant de commencer avec la programmation système, nous apprendrons à nous servir de make, afin de mieux gérer un nombre plus grand de fichiers interdépendants.
Travail à faire

- *allez dans votre **home directory***
- *allez dans sous-répertoire **TP** (s'il n'existe pas, créez-le)*
- *créez le sous-répertoire **tpsystem***
- *allez dedans*
- *créez le sous-répertoire **tpfile**, correspondant à cette unité de TP.*
- *allez dans **tpfile***
- *créez le répertoire **essai***
- *allez dans **essai***
- *créez les répertoires **include**, **dirfile***

2. Code source

Travail à faire

- *Créez ces trois fichiers ainsi, chacun dans le répertoire qu'il faut, comme indiqué dans le commentaire :*

```
// debut fichier file1.cxx, a mettre dans le repertoire dirfile
  #include <iostream>
  #include "notreEntete.h"

  int main() {
  int a,b;
  std::cin >> a >> b;
  int c (calculer(a,b));
  std::cout << c;
  return(0);
  }
// fin fichier file1.cxx
```

```
// debut fichier file2.cxx, a mettre dans le repertoire dirfile
  #include "notreEntete.h"
  int calculer(int a, int b) {
  return (a+b);
  }
// fin fichier file2.cxx
```

```
// debut fichier notreEntete.h, a mettre dans le repertoire include
  int calculer(int a, int b);
// fin fichier notreEntete.h
```

- *Allez dans le répertoire dirfile.*
- *Essayez de compiler "normalement": g++ -o file1.run file1.cxx file2.cxx*
- *Ceci ne fonctionnera pas, car le fichier en-tête n'est pas trouvable au même endroit que les sources l'incluant.*
- *Compilez alors ainsi :*

```
g++ -o file1.run file1.cxx file2.cxx -I../include
```

- *Lancez le programme en lui donnant deux nombres et constatez qu'il en affiche la somme.*

3. Makefile

En regardant l'arbre de dépendances de l'exécutable file1.run on voit que

- *si on modifie par exemple notreEntete.h ainsi, en lançant dans le shell 'touch notreEntete.h' c'est normal qu'on doit tout recompiler*
- *si on modifie file1.cxx on ne devrait pas tout recompiler. Il suffirait de mettre à jour file1.o et file1.run*

Heureusement nous disposons d'un outil astucieux -- make. On lui décrit ces règles de dépendances dans un fichier nommé makefile ou Makefile, et il s'en occupe. Le principe de l'exécution du make est d'évaluer d'abord la premiere règle rencontrée, ou celle dont le nom est specifié en argument de l'appel de make. L'évaluation d'une règle se fait recursivement. Si un fichier de dépendance est lui même un fichier cible d'une autre règle, cette règle est à son tour évalué.
Travail à faire

- ***Dans un fichier appellé makefile, mettez les dépendances** pour créer l'exécutable file1.run et les fichiers objet, file1.o, file2.o . La syntaxe d'une règle est la suivante*

```
cible : dependDeFic1 dependDeFic2 dependDeFic3
<TAB> commande de compilation g++ -c (ou edition de liens g++ -o)
```

- *Lancez la commande make et observez ce qui se passe.*
- *Maintenant faites un ls -lct dans le répertoire, pour examiner les dates de modification des fichiers.*
- *Maintenant faites un touch file1.cxx dans le shell, suivi d'un autre ls -lct. Vous allez constater que la date de file1.cxx est ultérieure à celle de prog.run*
- *Relancez alors la commande*

```
make
```

dans le shell -- vous devez constater que seulement les lignes

```
g++ -c -I../include file1.cxx
g++ -o prog.run file1.o file2.o
```

sont exécutées par l'outil make, car seulement les cibles file1.o, file.r1.run avait besoin d'être à jour.
- *Le fichier makefile peut contenir des commandes du shell sans dépendances -- ceci peut être util pour le nettoyage par exemple, avec rm *.o file1.run*
- ***Rajoutez dans le makefile la cible 'clean'** avec cette commande-ci (n'oubliez pas le TAB), et sans dépendances (mettez bien les ':' après le nom de la cible). Dans le shell, lancez la commande*

```
make clean
```

suivi d'un 'ls -l' pour constater ce qui s'est passé, et puis un autre 'make', suivi d'un autre 'ls -l', et une exécution de programme.

4. Makefile plus générique

L'outil make comprend également la notion de variable pour rendre les choses plus génériques et donc plus flexibles.
De même, on peut imaginer que file2.cxx grandira pour offrir une bibliothèque de fonctions. Alors on peut l'archiver avec l'outil ar et le donner au compilateur/linker avec -lSys.
Travail à faire

- *Remplacez 'g++ -c -I../include' partout dans le makefile avec la variable $(COMPILER), et définissez-la en haut du makefile ainsi*

  ```
  COMPILER = g++ -c -I../include
  ```

  (cette ligne est toute seule, il n'y a pas une seconde ligne, avec TAB, etc.)
- *Refaites*

  ```
  make clean
  ```

  et

  ```
  make
  ```

  pour constater que cela fonctionne comme avant.
- *On veut aussi pouvoir donner à make le nom du programme avec une variable, depuis le shell*

  ```
  make nom=file1
  ```

  pour obtenir file1.run. Il faut alors décrire les cibles et dépendances utilisant la variable nom. Pour les fichiers objets, on fait ainsi:

  ```
  $(nom).o : $(nom).cxx ../include/notreEntete.h
  <TAB> $(COMPILER) $(nom).cxx
  ```

- *Remplacez la dépendance de file2.o avec la dépendance d'une bibliothèque archivée libSys.a. Pour obtenir la cible libSys.a*
  - *d'abord on compile file2.cxx avec $(COMPILER) et puis*
  - *on utilise la commande d'archivage ar*

    ```
    ar -cqs libSys.a file2.o; rm file2.o
    ```

  (on peut mettre plusieurs commandes sur la même ligne mais avec ';').
- *Après remplacez file2.o avec libSys.a dans la dépendance de la cible $(nom).run. Changez aussi la commande d'après le TAB*

  ```
  g++ -s -o $(nom).run $(nom).o -L. -lSys
  ```

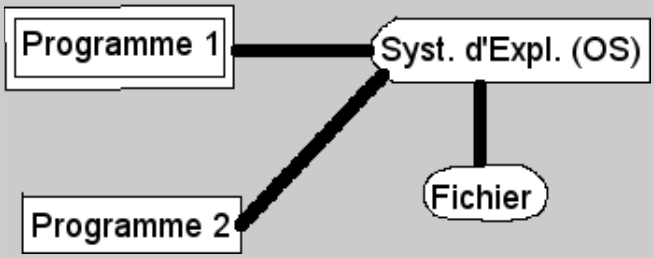
- *Faites les mêmes tests (make clean, make nom=file1, etc.), des 'ls -l' pour voir ce qui est apparu, etc. Cette fois, l'exécutable s'appelle donc file1.run.*
- *QUESTION: Faites une liste avec les fichiers/répertoires apparus. Dessinez l'arbre de dépendances de l'exécutable file1.run .*

Conclusion

Le code source de projets téléchargés pour Unix/Linux comporte un outil de configuration s'adaptant au type de Unix et installation concrets, générant ensuite un makefile pour la compilation et édition de liens pour le projet.
Donc make est incontournable: maîtrisez rapidement ses fonctionnalités de base, et jetez un coup d'oeil sur sa page de man.

Appels système, `errno`, `strerror()`

*Que fait un appel système ? Les appels système offrent aux programmes des **services** de base : lecture depuis un fichier, écriture dans un fichier, renseignements sur un fichier (taille, dates de modification, etc.), démarrage d'un programme, arrêt d'un programme, signalisation ("coup de fil") d'un programme vers un autre, envoi de données par réseau, et ainsi de suite. Ceci est exactement le but du système d'exploitation (Operating System): fournir aux programmes de l'utilisateur un cadre pour "vivre" "en paix" et intéragir efficacement avec l'utilisateur ou avec d'autres programmes.*



5. Les erreurs des appels système

Vous connaissez la commande shell `ls`, avec son paramètre `-l` -- elle vous affiche les noms des fichiers et répertoires, leur taille ainsi que d'autres renseignements. Tout ceci est obtenu à l'aide de l'appel système `stat()`, qui prend deux arguments. Voyons comment cela fonctionne :

- *A. avec votre éditeur de texte (emacs, etc.) pour la programmation créez un fichier nommé `exempleStat.cxx` (ou bien téléchargez-le), et mettez dedans ceci :*

```
#include <iostream>
#include <string>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int main (int argc, char * argv []) {
    if(argc < 2) {
        cerr << "Usage: " << argv[0] << " <nomDeFicher>\n";
        return(1);
    }
    struct stat statStruct;
    stat(argv[1],&statStruct);
    cout << argv[1] << " taille " << statStruct . st_size << "\n";
    return(0);
}
```

- *B. sauvegardez, et dans le shell, compilez ainsi*

```
g++ -o exempleStat.run exempleStat.cxx
```

- *C. lancez le programme ainsi, avec un argument*

```
./exempleStat.run exempleStat.cxx
```

- *D. faites un `ls -l` pour confirmer que votre programme vous donne la bonne réponse*
- ***QUESTION: Quel est le code de retour de l'exécution du programme en ayant en argument un nom de fichier qui n'existe pas?***

```
./exempleStat.run blah
```

Et si on lance le programme sans aucun argument?

```
./exempleStat.run
```

Pourquoi ?

- **CONCLUSION Les appels système font leur travail "en silence", sont "tolérants" et sans tests specifiques on peut continuer sans se rendre compte de l'existence d'une erreur.**
- **MAIS ALORS COMMENT FAIRE pour savoir vraiment si un appel a réussi ? Avec une procédure "standard": récupérer la valeur de retour, et si elle n'est pas zéro, regarder dans une variable spéciale (appelée `errno`) pour savoir ce qui est arrivé:**

> **Un appel système ÉCHOUÉ positionne la variable globale `errno` (qui est déclarée dans `errno.h` en tant qu'`extern`, donc à l'édition de liens elle sera mise en correspondance avec la bonne variable du code système). De plus, la fonction de bibliothèque `strerror()` fournit une description lisible pour les humains associée à chaque valeur d'errno.**

- **Mettez ce qui suit dans un autre fichier, appelé `exempleStatErrno.cxx`, et refaites le tout dessus comme aux points B, C et D (compilation, tests, `echo $?`) -- vous devez constater une différence pour la valeur de retour et pour la "réaction" du programme.**

```
#include <iostream>
#include <string>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <cerrno>
#include <cstring>

using namespace std;

int main (int argc, char * argv []) {
    if(argc < 2) {
        cerr << "Usage: " << argv[0] << " <nomDeFicher>\n";
        return(1);
    }
    struct stat statStruct;
    const int valRet = stat(argv[1],&statStruct);
    if ( -1==valRet) {
        std :: cerr << " Erreur : " << argv[1] << " "
                    << strerror (errno) << "\n" ;
        return(errno);
    }
    cout << argv[1] << " taille " << statStruct . st_size << "\n";
    return(0);
}
```

Faire cela après chaque appel système est fastidieux, donc on verra de quelle manière nous nous allégerons cette tâche pour tout ce cours et TD/TP.

- **QUESTION: Quel est le code de retour de l'exécution du programme en ayant comme argument un fichier qui n'existe pas?**

```
./exempleStat.run blah
```

Utilisez le `man` dans le shell, en lançant `man 2 stat`, pour trouver une explication du code d'erreur obtenu, plus détaillée que celle fournie par `strerror()`. Ecrivez-la.

Structure des répertoires pour les TP

- **allez dans votre home directory**
- **allez dans sous-répertoire TP (s'il n'existe pas, créez-le)**
- **créez le sous-répertoire tpsystem**
- **allez dedans**

- ***créez le sous-répertoire tpfile, correspondant à cette unité de TP.***
- ***allez dans tpfile***
- ***créez le répertoire exo_01***
- ***allez dans exo_01***
- ***créez les répertoires include, dirfile***
- ***copiez (avec l'option -r), le répertoire exo_01 pour créer encore cinq autre répertoires nommés exo_02, exo_03, etc.***

Vous mettrez tous les fichiers créés au cours de chaque exercice des TP dans le répertoire dirfile de l'exercice courant, qui sera considéré comme répertoire courant. En passant d'un exercice à l'autre vous allez recopier le contenu de include et, souvent, y rajouter également des fonctions. Vous mettrez dans le répertoire

- ***include tous les .h contenant des en-têtes avec des déclarations et des définitions inline, pour les***
  - ***wrappers***
  - ***exceptions***
  - ***quelques fonctions utilitaires***
- ***et un fichier de définitions pour make, appelé INCLUDE_H.***
- ***Pour faire de la place dans vos répertoires, lorsque vous finirez ce TP et passerez au suivant***
  - ***Effacez bien tous les exécutables et les objets, soit***
    - ***avec le make clean (que vous mettrez en place) dans le dir* de chaque exo_*, soit***
    - ***avec une simple commande rm */*/*.{o,a,run}***
      dans le répertoire tpfile. La même procédure devra bien entendu être appliquée pour la suite.
  - ***Compressez le contenu de chaque répertoire d'exercice avec tar et gzip, avec la commande suivante (exemple pour exo_01) :***

    ```
    tar cvf - exo_01 | gzip -9 - > exo_01.tar.gz
    ```

  - ***Attention aux espaces autour des tirets seuls, ces tirets demandent l'utilisation de l'entrée, respectivement sortie standard (au lieu d'utiliser des noms de fichiers). Ainsi on peut enchaîner ces commandes et tout déposer directement dans exo_01.tar.gz Par contre, le -9 est bien ensemble, cette option demande la meilleure compression.***

- ***Respectez l'ordre des exercices, car il y a une progression déterminée, qu'il est impératif de suivre.***

Premiers appels système (fichiers)

Sommaire

- ***Liste des fonctions et concepts système étudiés***
- ***Liste des exercices***
- ***Pour chaque exercice (exo_01, exo_02)***
  - ***Présentation courte du but et des sous-points***
  - ***Éventuelles explications nécessaires pour les outils***
  - ***Travail à faire***
  - ***Renvois vers des remarques, rappels du cours, petites astuces, etc.***

Attention: à la fin de ce TP vous devrez rendre une feuille avec des réponses à des questions du TP.

Liste des fonctions et concepts système étudiés

Dans ces deux exercices on met d'abord le tout en place (premier exercice), et ensuite on utilise stat() pour obtenir des renseignements sur les fichiers.
Ne pas oublier de consulter les pages de man 2 nom_fonction, man 3 nom_fonction.

Liste des exercices

- ***exo_01 mise en place de l'espace de travail***
- ***exo_02 fonction stat(), erreurs et exceptions***

Bon courage !

Exo_01 structure de travail, fonction stat(), erreurs et exceptions

Dans cet exercice

- ***vous étudiez la manière dont on utilise les exceptions pour gérer les erreurs éventuelles des appels système.***
- ***La mise en place de la structure générale du main(), qui appelle toutes les fonctions faisant le travail de l'exercice***
- ***Toutes les exceptions échappées de ces fonctions sont captées par le try-catch du main() et un message est affiché***
- ***Les outils (wrappers, exceptions) compilés sont rassemblés dans notre bibliothèque libSys.a (logée dans le répertoire lib), et le tout est mis ensemble avec make.***

Quoi de neuf : wrappers (enrobeurs) et exceptions

- ***Les erreurs système modifient la variable globale errno. Donc si une seconde erreur apparaît avant de pouvoir lire l'errno, on perd le code de la première***
- ***On veut s'interrompre magiquement à l'endroit où une erreur surgit, pour transmettre le maximum de renseignements là-dessus au code appelant.***
- ***On utilise pour ce faire une classe d'exception CExc derivée de exception (qui vient avec C++), mais qui a un affichage facile***
- ***Les exceptions de C++ interrompent le flot d'exécution du programme, remontant à la première partie catch() qui peut la traiter:***

```
int main (int argc, char * argv []) {
  // ...
  try {
  // le corps de l'exercice appellant des fonctions qui peuvent lever des exceptions
  }
  catch (const CExc & Exc) {
   // on affiche les renseignements de l'erreur

  }
  ....
  }
```

- ***Une telle fonction qui peut lever des exceptions (instruction throw) est par exemple un wrapper d'appel système levant une exception de type CExc***
- ***La classe CExc peut afficher ces renseignements (donnés à son constructeur) avec***

l'opérateur << :

```
class CExc: public std::exception
{
protected :
  std::string m_info;
  std::string m_nomf;
  int m_descrfic;
  bool m_qdescrfic;
protected :
  std::ostream & _Edit (std::ostream & os) const;
public :
  CExc (const std::string & NomFonction,
        const std::string & Info) throw ();
  CExc (const std::string & NomFonction,
       int Descrfic) throw ();
  virtual ~CExc (void)throw ();
  friend std::ostream & operator << (std::ostream & os,
  const CExc & Item);
};

nsSysteme::CExc::CExc (const std::string & NomFonction,
                       const std::string & Info) throw ()
                      : m_info (Info), m_nomf(NomFonction), m_descrfic(-1), m_qdescrfic(false) {}
...
```

Elle a deux constructeurs car tous les appels système n'ont pas toujours un nom de fichier disponible. On verra par la suite comment on s'en sert de l'autre.

Travail à faire

- Copiez le fichier CExc.h et mettez -le dans le répertoire 'include'
- Copiez dans le répertoire 'include' le fichier nsSysteme.h. Il contient le profile du wrapper (enrobeur). Rajoutez sa définition et les bons #include (utilisez man 2 stat).
- Dans le fichier nsSysteme.cxx du répertoire 'dirfile' mettez seulement la ligne

```
#include "nsSysteme.h"
```

- Enfin, donc, dans le fichier exo_01.cxx du répertoire 'dirfile' mettez un main () ayant la structure imposé suivante :

```
#include <string>
#include <exception>


#include "string.h"

#include "CExc.h"
#include "nsSysteme.h"  // wrappers système

// a verifier avec le man 2 ou man 3  de la fonction systeme les #include a rajouter

using namespace nsSysteme; // wrappers système
using namespace std;

int main (int argc, char * argv []) {
  try {

  // ecrivez le corps de l'exercice courant

  }
  catch (const CExc & Exc) {
    cerr << Exc << endl;
```

```
        return errno;
      }
      catch (const exception & Exc) {
        cerr << "Exception : " << Exc.what () << endl;
        return 1;
      }
      catch (...) {
         cerr << "Exception inconnue recue dans la fonction main()" << endl;
         return 1;
      }
    }// main()
```

- ***Récupérez le fichier [Makefile](#) et placez-le dans le répertoire 'dirfile'.***
- ***Récupérez le fichier [INCLUDE_H](#) et placez-le dans le répertoire 'include'.***
- ***Lancez dans le shell la commande***

```
make general
```

pour constater que le wrapper de stat() (l'espace de noms nsSysteme) est bien déclaré et défini. On finalise la mise en place, afin de pouvoir s'en servir tout du long des TP.

- ***QUESTION: Quel est le premier répertoire visité pendant l'exécution du make? Quels sont les fichiers et/ou répertoires crées par l'exécution du make?***

Attention: ne compilez pas encore plus que ce qui est dit plus haut!

Exo_02. Les structures C , compilation, édition de liens, le makefile et l'exemple revisité

Dans cet exercice

vous prenez contact avec un premier appel système (pour trouver la taille d'un fichier),

1. ***Fonction stat()***
2. ***Le type C struct***
3. ***Un main() pour attraper les exceptions, compilation, édition de liens, le makefile et le programme du 1. revisité***

Quoi de neuf

Le type structure struct

Datant d'avant l'apparition des classes, les structures permettent de rassembler des éléments de type différent au sein d'une entité repérée par un seul nom de variable. Conditions: deux champs ne peuvent avoir le même nom, les champs peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent. Une structure est composée d'éléments de taille fixe, et donc on peut créer des tableaux contenant des éléments de type structure.
La définition d'une structure:

```
struct Point {
int x;
int y;
char etiquette[7];
};

/*déclaration et utilisation: */

struct Point p = {3,5,"A(x,y)"};
p.x=4;
p.y=3;
```

```
...

/* et avec des pointeurs */

struct Point *q;

... /* allocation mémoire */

q->x = 3;
q->y = 5;
q->etiquette[0]= 'A';
q->etiquette[1]= '(';
...
```

Travail à faire

- *Créez le répertoire exo_02 avec ses sous-répertoires et récupérez '/dirfile/nsSysteme.cxx', le makefile et le répertoire 'include'*
- *Dans le fichier exo_02.cxx*
  - *rajoutez les #include nécessaires*
  - *déclarez une struct stat : struct stat S;*
  - *dans le try ... catch faites un appel à l'enrobeur/wrapper Stat() qui a le même profile que la fonction stat() du main (le système remplit la variable S avec des renseignements sur le fichier donné comme paramètre à Stat()*
  - *afficher la taille et le nombre de liens en dur envers le fichier donné en tant que argument d'entrée (man 2 stat indique comment récupérer ces renseignements depuis les champs de la variable)*
  - *OPTIONNEL: affichez également la date de la dernière modification. Indication: les systèmes Unix utilisent le nombre de secondes écoulées depuis le 1er janvier 1970 pour indiquer la date. Pour traduire ce nombre en format usuel pour les humains, on peut utiliser par exemple la fonction ctime(), qui prend un pointeur vers un time_t *. Le champ st_mtime de la structure struct stat S est de type `time_t *` .*
- *Compilez le tout avec make nom=exo_02 et testez-le.*
- *Utilisez la commande*

  `strace`

  pour avoir la trace des appels système effectués par votre programme. Si on est en bash, il faut faire par exemple

  ```
  strace ./exo_02.run exo_02.run
  ```

  Pour filtrer la sortie, on peut faire par exemple

  ```
  strace ./exo_02.run exo_02.run 2>&1 | grep stat
  ```

- ***QUESTION: Quels sont les appels système effectués par votre programme?***
- *OPTIONNEL: Écrire un programme C++ qui fait (une partie de ce que fait) la commande stat (vous pouvez jeter un coup d'oeil à man 1 stat). Ce programme prendra un ou plusieurs fichiers en argument et devra afficher pour chacun d'entre eux*
  - *son type (fichier, répertoire, lien symbolique, périphérique, fifo, socket)*
  - *son numéro d'inode*
  - *le nombre de liens durs pointant vers cet inode*
  - *sa taille*
  - *ses permissions, dans le style de ls -l*
- *Pour ce faire, servez-vous bien entendu du wrapper de l'appel système stat(). Regardez dans le man comment se servir des informations stockées dans la structure de type struct stat pointée par buff. Un des champs de cette structure s'appelle st_mode et il décrit à la fois le type du fichier et les droits d'accès.*

Solutions

Solution exo_01

exo_01.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h

Solution exo_02

exo_02.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h