HEB-ESI
1re année

Année 2006-2007
BEJ, DNA, **HAL**,
*NVS**, PBT

LMI-TD04 : Langage d'assemblage sous Linux avec nasm

1 Matière abordée au long du TD

Manipulation de fichiers, récupération d'arguments par le biais de la ligne de commande, transmission d'argument à une procédure par la pile, valeur en retour d'une procédure, variables locales.

2 Les fichiers

2.1 En général

Le système d'exploitation offre une série d'appels systèmes qui permettent au processus de retrouver, lire ou modifier des données qui se trouvent sur disque :

- `open` permet de retrouver un fichier, de l'ouvrir et de lui associer un descripteur de fichier contenant, entre autre, la position actuelle (initiallement nulle) dans le fichier ;
- `read` permet de transférer en mémoire centrale un ensemble d'octets enregistrés à la position actuelle d'un fichier. La position actuelle est incrémentée du nombre d'octets lus ;
- `write` permet de transférer un ensemble d'octets depuis la mémoire centrale vers la position actuelle d'un fichier. Cette dernière est incrémentée du nombre d'octets écrits ;
- `close` permet de fermer un fichier ;
- `lseek` permet de modifier la position actuelle d'un fichier.

Voici quelques valeurs de constantes utiles, extraites du fichier `/usr/include/bits/fcntl.h` :

- `O_RDONLY` == 0 : pour ouvrir un fichier en lecture seule ;
- `O_WRONLY` == 1 : pour ouvrir un fichier en écriture seule ;
- `O_RDWR` == 2 : pour ouvrir un fichier en lecture / écriture ;
- `O_CREATE` == 64 : pour ouvrir un fichier en le créant s'il n'existe pas ;
- `SEEK_SET` == 0 : pour indiquer un décalage par rapport au début du fichier ;
- `SEEK_CUR` == 1 : pour indiquer un décalage par rapport à la position courante du fichier ;
- `SEEK_END` == 2 : pour indiquer un décalage par rapport à la fin du fichier.

Exercice1. Écrivez un programme, `helloFichier`, qui crée le fichier `hello.dat` de contenu « Hello world ! ».

* Ce TD s'inspire largement de ceux rédigés par JCJ en 2006.

Exercice2. Écrivez un programme, `helloAgainFichier`, qui *ajoute* au fichier `hello.dat` : « En français : "Bonjour le monde" ».

2.2 La console

Linux considère le clavier et l'écran comme des fichiers. De plus, tout processus ouvre automatiquement trois fichiers :
- celui de descripteur 0 : l'entrée standard, associée par défaut au clavier ;
- celui de descripteur 1 : la sortie standard, associée par défaut à l'écran ;
- celui de descripteur 2 : la sortie d'erreur standard, associée également par défaut à l'écran.

Ils sont fermés automatiquement à la fin du processus.

2.3 Retour en général

La fin d'un fichier est détectée lorsque, lors d'un `read`, le nombre d'octets lus est strictement inférieur à celui demandé. Quant au fichier associé au clavier, on indique sa fin en enfonçant les touches Ctrl-D simultanément.

Exercice3. La commande `cat` sans argument est une commande qui écrit à l'écran tout ce qu'on tape au clavier. Écrivez un programme, `monCat`, qui effectue le même boulot que la commande `cat` sans argument. Testez votre programme :
- simplement : `monCat` ;
- en redirigeant la sortie standard vers un fichier : `monCat > fichierSortie` ;
- en redirigeant l'entrée standard depuis un fichier : `monCat < fichierEntree`.

Vous pourriez être étonnés que l'affichage n'est pas synchronisé avec l'appel de `write`. Linux place tous les caractères à écrire dans un tampon (*buffer* en anglais) afin de les afficher en une seule fois. En général, le saut de ligne provoque l'affichage du contenu du tampon à l'écran.

3 Utilitaires Linux et arguments en ligne de commande

Dans la section précédente, le fonctionnement de `cat` *sans* argument a été étudié. Lorsqu'on ne fournit pas d'argument, la plupart des utilitaires utilisent les entrée et sortie standard (la console, donc).

Si on donne des arguments, la plupart des utilitaires considère qu'il s'agit de noms de fichier et que ces derniers doivent être traités séquentiellement comme l'entrée standard.

À titre d'exemple, vérifiez, en ligne de commande, le comportement de `cat` sans, avec un puis avec trois arguments.

Les arguments du point d'entrée du programme sont transmis par la pile. La Fig. 1 explique comment. Notez que sur cette figure, la pile se remplit *vers le bas*. Le registre `esp` pointe, comme toujours, sur la dernière valeur empilée. Toutes les valeurs dans la pile ont comme taille 32 bits (ou 4 octets).

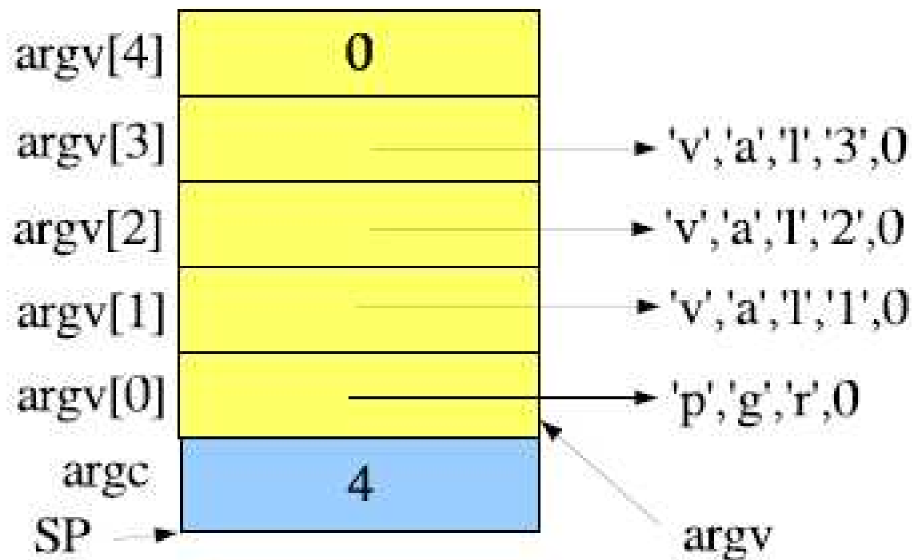

FIG. 1 – État de la pile au commencement de la procédure principale du programme `pgr` invoqué par : `./pgr val1 val2 val3`. *Attention :* remplacer `sp` par `esp`.

Exercice4. Écrivez un programme, `monEcho`, qui effectue la même chose que la commande `echo`. Testez votre programme avec les commandes suivantes :
- `./monEcho Hello World`;
- `./monEcho *.asm`;
- `./monEcho Hello World > fichierSortie`.

Exercice5. Écrivez un programme, `monHead`, qui effectue la même chose que la commande `head`. Testez votre programme avec les commandes suivantes :
- `./monHead`;
- `./monHead monHead.asm`;
- `./monHead *.asm`;
- `./monHead > fichierSortie`.

4 Procédure, passage d'argument, valeur de retour et variable globale

On peut passer les arguments d'une procédure en les plaçant dans des registres. C'est la technique utilisée pour les appels système. Cependant, le nombre et la taille des registres sont limités.

On peut passer des arguments à une procédure en plaçant la valeur de ces arguments sur la pile. Elles sont lues par la procédure mais les éventuelles variables dont elles sont issues ne peuvent être modifiées car la procédure ne connaît pas l'adresse de ces variables. On parle dans

ce cas d'une transmission d'argument par valeur.

On peut passer des arguments à une procédure en plaçant l'adresse de variables dans la pile. Les arguments peuvent être lus par la procédure. Ils peuvent aussi être modifiés car la procédure connaît l'adresse de ces arguments. On parle de passage d'arguments par adresse. Remarquez que l'adresse d'une variable peut également être transmise par registre.

La procédure peut renvoyer une information au programme qui l'appelle. En général, cette information se trouve dans le registre `eax`. C'est la valeur de retour. Si la procédure doit renvoyer plus d'une information, on lui passe, en argument, l'adresse où elle peut fournir ces informations.

La pile sert également, au sein d'une procédure, à y créer des variables locales. Leur création, sous la forme de *trou* dans la pile, et leur destruction en fin de procédure n'est pas automatique en langage d'assemblage.

Dans tous les exercices qui suivent, transmettez les arguments par la pile. De plus, veillez à n'utiliser aucune variable globale, à l'exception des chaînes de caractères.

Exercice6. Écrivez une procédure `afficher` qui affiche à l'écran une chaîne de maximum 80 caractères terminée par un zéro binaire. L'affichage de la chaîne peut être suivi par un saut de ligne, selon les désirs de son utilisateur. Testez votre procédure avec un programme `testAfficher.asm` qui appelle `afficher` comme procédure externe. La fonction `afficher` utilise comme arguments :
- l'adresse de la chaîne à afficher ;
- une valeur numérique stockée sur un double mot : si cette valeur est non nulle, l'affichage de la chaîne est suivi par un passage à la ligne.

Elle renvoie (dans `eax`) :
- 0 : si aucune erreur n'est rencontrée ;
- 1 : si on tente d'afficher une chaîne de plus de 80 caractères.

Exercice7. Écrivez une procédure `entierBinToStr` qui transforme un entier stocké dans un double mot en une chaîne de caractères terminée par un zéro binaire. Testez votre procédure avec un programme `testEBTS` qui appelle `entierBinToStr` et `afficher` comme procédures externes. La fonction `entierBinToStr` a comme arguments :
- la valeur du nombre à transformer ;
- la manière d'interpréter le motif binaire du nombre à transformer : 0 pour un nombre non signé, 1 pour un nombre signé ;
- l'adresse de la chaîne de caractères où stocker le résultat de la conversion.

Elle représente toujours l'entier en base 10. Aucune validation relative à la longueur de la chaîne fournie ne peut être réalisée par `entierBinToStr`. Veillez donc à ce que cette chaîne soit suffisamment grande lors de son test.

Exercice8. Écrivez une procédure `lire` qui lit une ligne au clavier et la transforme en une chaîne de maximum 80 caractères terminée par un zéro binaire. Testez votre procédure avec le programme `testLire.asm` qui appelle `lire` et `afficher` comme procédures externes. La fonction `lire` utilise comme argument l'adresse de la chaîne résultat de la lecture et renvoie (dans `eax`)
- 0 : si tout est ok ;

– 1 : si la chaîne en entrée fait plus de 79 caractères. Dans ce cas, tronquer en sortie la chaîne en entrée à ses 79 premiers caractères.

Aucune validation sur la longueur de la chaîne où `lire` écrit son résultat n'est réalisée. Veillez donc à ce que cette chaîne soit suffisamment longue lors de l'utilisation de `lire`.

Exercice9. Écrivez une procédure `strToEntierBin` qui transforme une chaîne de caractères terminée par un zéro binaire en un nombre binaire entier signé contenu dans un double mot. La chaîne est au format [+/-]digit[digit]. Testez votre procédure avec un programme `testSTEB` qui appelle `lire`, `strToEntierBin`, `entierBinToStr` et `afficher` comme procédures externes. La fonction `strToEntierBin` utilise comme arguments :

– l'adresse de la chaîne de caractères où est représenté le nombre à transformer ;
– l'adresse où le nombre doit être stocké sous forme binaire ;

et renvoie (dans `eax`)

– 0 : quand aucune erreur n'est rencontrée ;
– 1 : lorsque la chaîne contient au moins un caractère non valide sachant que la représentation textuelle attendue est celle d'un entier signé en base 10 ;
– 2 : en cas de dépassement de capacité lors de la conversion.

Exercice10. (Exercice d'évaluation) Utilisez les diverses fonctions dont le développement vous est demandé ci-dessus au sein d'une fonction principale `main` telle que le programme correspondant, `operer`, ait le comportement suivant. `operer` reçoit trois arguments par la ligne de commande. Ces trois arguments doivent représenter une paire d'entiers signés en base 10 suivie d'un caractère associé à une opération arithmétique. Une mise en forme usuelle du calcul puis le résultat sont affichés[†], si les arguments sont valides en nombre et en nature et si aucun problème ne survient à quelque moment que ce soit du traitement. Dans le cas contraire, un message d'erreur *adéquat* est affiché. Les opérations arithmétiques supportées sont : l'addition, la soustraction, la multiplication, la division entière et le modulo.

[†] C'est-à-dire quelque chose du genre : `opérande1 opération opérande2 = résultat`.