Systèmes d'Exploitation

Devoir 1 : Entrées/Sorties dans NACHOS

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/se/

L'objectif de ce devoir est de mettre en place en Nachos quelques appels systèmes de base.
Le devoir est à faire en binôme et à rendre (fichiers sources + rapport de quelques pages) avant

le Lundi 11 Octobre 2021 (8h)

Plus précisément, il vous est demandé de rendre les pièces suivantes :
— une description de la stratégie d'implémentation utilisée, et une discussion des choix que vous avez faits (5 pages maximum). Un guide de rédaction de ce document est disponible sur le site de l'UE : guide.txt ;
— Vos sources, qui comprendront des programmes de test représentatifs présentant les qualités de votre implémentation et ses limites. Chaque test doit contenir un commentaire expliquant comment le programme doit être lancé (options nachos à utiliser,...) et être accompagné d'un court commentaire (5–10 lignes) expliquant son intérêt. Une archive de votre répertoire `code`, dans lequel vous aurez tapé `make clean` au préalable, suffira.
— *Il **n**'est **pas** demandé de répondre aux questions de ce sujet dans l'ordre où elles sont posées.*

L'énoncé est volontairement laissé flou sur plusieurs points. Certains choix de conception *non triviaux* sont donc laissés à votre appréciation.

Avant de commencer à coder, lisez bien chaque partie **en entier** : les sujets de TP contiennent à la fois des passages descriptifs pour expliquer vers où l'on va (et donc il ne s'agit que de lire et comprendre, pas de coder), et des **Actions** qui indiquent précisément comment procéder pour implémenter pas à pas (et là c'est vraiment à vous de jouer).

Note *Attention : ce sujet demande beaucoup de méthode de votre part. Réfléchissez avant de coder, sinon ça sera encore plus dur !*

D'autre part, toutes vos modifications doivent être encadrées avec

```
#ifdef CHANGED
  ...
#endif // CHANGED
```

pour pouvoir être réversibles. Les `Makefile` fournis compilent déjà avec `-DCHANGED`. Les modifications non signalées *sont rigoureusement interdites.*

Partie I. Quel est le but ?

L'objectif de ce devoir encadré est de mettre en place sous Nachos un mécanisme d'entrée-sortie minimal, permettant d'exécuter le petit programme `putchar.c` suivant. Quelle est la sortie raisonnablement attendue ? Notez qu'on ne testera véritablement ce programme qu'une fois que vous aurez implémenté l'appel système `PutChar`. À ce moment-là on supprimera le `#if 0`...

```
#include "syscall.h"

void print(char c, int n)
{
  int i;
#if 0
  for (i = 0; i < n; i++) {
    PutChar(c + i);
  }
  PutChar('\n');
#endif
}

int
main()
{
  print('a',4);
  Halt();
}
```

Il faut donc placer ce programme `test/putchar.c` sous le répertoire `test`. Ajoutez-le à la base SVN en utilisant `svn add putchar.c` puis `svn commit putchar.c`

Il est normal que cela ne compile pas encore sans erreur, vous complèterez plus tard dans le projet et l'erreur de compilation ne vous gênera pas entre-temps.

Partie II. Entrées-sorties asynchrones

Nachos offre une version primitive d'entrées-sorties par la classe `Console` qui se trouve déclarée sous `machine/console.h`. Lisez attentivement les commentaires. Ne lisez pas `machine/console.cc`, puisque c'est la simulation du matériel ; c'est l'interface `machine/console.h` qu'il fournit qui nous intéresse. Les entrées-sorties fonctionnent de manière asynchrone, par interruption :

- Pour écrire un caractère, on "poste" une requête d'écriture vers le matériel grâce à `Console::TX(char ch)`, puis on attend d'être averti de la terminaison de la requête par l'exécution du traitant (anglais : handler) d'interruption `WriteDoneHandler`.
- Pour lire, on attend d'être averti qu'il y a quelque chose à lire par l'exécution du traitant d'interruption `ReadAvailHandler`, puis on réalise la lecture effectivement depuis le matériel par la fonction `Console::RX()`.

C'est une erreur que de chercher à lire un caractère avant d'être averti qu'un caractère est disponible, ou de chercher à écrire avant d'être averti que l'écriture précédente est terminée. Expliquez pourquoi.

Notez que les traitants sont des fonctions C, pas C++, car elles sont partagées par la console et les classes qui l'utilisent.

Notez aussi qu'il n'y a aucune raison de ne pas faire des choses utiles entre le moment où l'on "poste" la requête et le moment où l'on est averti de sa terminaison. On peut tout à fait *recouvrir* les communications par des calculs !

Regardez maintenant la mise en oeuvre sous `userprog/progtest.cc`. On se place d'abord dans un cas simple où l'on se bloque sur l'attente de terminaison grâce à des sémaphores.

```
static Console *console;
static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvailHandler(void *arg) { (void) arg; readAvail->V(); }
static void WriteDoneHandler(void *arg) { (void) arg; writeDone->V(); }
```

Pour attendre, on prend le sémaphore. Les traitants de notification les libèrent. En conséquence, si le caractère est déjà présent lors d'une demande de lecture, on est immédiatement servi !

```
readAvail->P();          // wait for character to arrive
ch = console->RX();      // read it
```

Action II.1. *Examinez* `ConsoleTest` *dans* `userprog/progtest.cc`*. Depuis* `userprog`*, lancer* `./nachos -c` *qui exécute* `ConsoleTest` *(voir* `threads/main.cc`*). Bien comprendre ce qui se passe.*

Action II.2. *Modifiez* `ConsoleTest` *pour afficher « Au revoir » quand on quitte avec le caractère 'q'. Faites également terminer la boucle sur fin de fichier, i.e. quand* `RX` *retourne la valeur* `EOF` *(Note : pour exprimer la fin de fichier au clavier, il suffit de taper sur control-D en début de ligne).*

Action II.3. *Modifiez* `userprog/progtest.cc` *pour faire écrire* `<x>` *au lieu de* `x` *dans le corps de la boucle (quel que soit le caractère* `x`*).*

Action II.4. *Vérifiez que cela fonctionne aussi avec un fichier d'entrée et un de sortie. Par exemple,* `./nachos -c in out` *pour travailler sur les fichiers* `in` *et* `out`*. (Voir* `threads/main.cc`*.)*

Partie III. Entrées-sorties synchrones

L'objectif est d'implémenter au-dessus de la couche `Console` un *driver* de console qui fonctionne de manière *synchrone*, `ConsoleDriver`. L'idée est que le driver encapsule tout le mécanisme des sémaphores pour ne fournir que deux fonctions *synchrones* `PutChar` et `Getchar`.

Action III.1. *Créez le fichier* `userprog/consoledriver.h` *comme suit (on pourra copier/coller depuis le PDF). À noter que le* `#include "console.h"` *fonctionne déjà correctement grâce au chemin de recherche spécifié dans l'appel au compilateur.*

```
#ifdef CHANGED

#ifndef CONSOLEDRIVER_H
#define CONSOLEDRIVER_H

#include "copyright.h"
#include "utility.h"
#include "console.h"

class ConsoleDriver:dontcopythis {
  public:
                                  // initialize the hardware console device
```

```
    ConsoleDriver(const char *readFile, const char *writeFile);
    ~ConsoleDriver();               // clean up

    void PutChar(int ch);           // Behaves like putchar(3S)
    int GetChar();                  // Behaves like getchar(3S)

    void PutString(const char *s);  // Behaves like fputs(3S)
    void GetString(char *s, int n); // Behaves like fgets(3S)
  private:
    Console *console;
};

#endif // CONSOLEDRIVER_H

#endif // CHANGED
```

Notez que les sémaphores doivent être globaux, car lorsque le traitant d'interruption est invoqué par le matériel, il ne sait pas quel objet C++ correspond à votre consoledriver. Le fichier `userprog/consoledriver.cc` doit donc avoir la structure suivante :

```
#ifdef CHANGED

#include "copyright.h"
#include "system.h"
#include "consoledriver.h"
#include "synch.h"

static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvailHandler(void *arg) { (void) arg; readAvail->V(); }
static void WriteDoneHandler(void *arg) { (void) arg; writeDone->V(); }

ConsoleDriver::ConsoleDriver(const char *in, const char *out)
{
  readAvail = new Semaphore("read avail", 0);
  writeDone = new Semaphore("write done", 0);
  console = ...
}

ConsoleDriver::~ConsoleDriver()
{
  delete console;
  delete writeDone;
  delete readAvail;
}

void ConsoleDriver::PutChar(int ch)
{
  // ...
}

int ConsoleDriver::GetChar()
{
  // ...
```

```
}

void ConsoleDriver::PutString(const char s[])
{
  // ...
}

void ConsoleDriver::GetString(char *s, int n)
{
  // ...
}

#endif // CHANGED
```

Action III.2. *En vous inspirant du code de ConsoleTest, complétez* `consoledriver.cc` *en ce qui concerne les opérations sur les caractères (comme documenté dans* `consoledriver.h`*, ils avoir un comportement équivalent aux fonctions Unix* `putchar` *et* `getchar`*). Ne complétez pas encore* `PutString` *et* `GetString`*.*

Action III.3. *Complétez le fichier le fichier* `Makefile.common`*. À chaque fois que* `console` *apparaît,* `consoledriver` *doit aussi apparaître.*

Action III.4. *Modifiez* `threads/main.cc` *pour ajouter une option* `-sc` *de test de la console synchrone qui lance la fonction* `ConsoleDriverTest` *que l'on ajoute dans l'action ci-dessous.*

Action III.5. *Ajoutez à la fin de* `progtest.cc` *la définition de cette fonction. Par exemple :*

```
#ifdef CHANGED

void
ConsoleDriverTest (const char *in, const char *out)
{
  char ch;
  ConsoleDriver *test_consoledriver = new ConsoleDriver(in, out);

  while ((ch = test_consoledriver->GetChar()) != EOF)
      test_consoledriver->PutChar(ch);
  fprintf(stderr, "EOF detected in ConsoleDriver!\n");

  delete test_consoledriver;
}

#endif //CHANGED
```

et pensez à inclure `<consoledriver.h>` *pour obtenir la déclaration de la class* `ConsoleDriver` *Notez que le* `fprintf` *est effectué par Linux, pas par Nachos !*

Action III.6. *Agrémenter la fonction* `ConsoleDriverTest` *avec des < et > comme à la partie précédente.*

Partie IV. Appel système `Putchar`

L'objectif est maintenant de mettre en place un appel système `PutChar(char c)` qui prend en argument un caractère `c` en mode utilisateur puis lève une interruption `SyscallException`. Celle-ci provoque le passage en mode noyau et l'exécution du traitant standard `ExceptionHandler`. Celui-ci doit récupérer le paramètres depuis le monde MIPS, appeler la méthode `PutChar`, puis rendre la main au programme appelant, en ayant soin d'incrémenter le compteur de programme ! Vous comprenez maintenant pourquoi un appel système est si coûteux. C'est pourquoi dans un vrai système les entrées-sorties Unix sont *bufferisées* : `fprintf(3)` est bien moins coûteux que `write(2)` sur chaque caractère, puisqu'il y a un appel système à chaque *ligne*, et non à chaque *caractère* !

De cette manière, le programme *utilisateur* Nachos `putchar` ci-dessus devrait fonctionner !

La première tâche est de créer le driver console lors de l'initialisation du système pour l'exécution de programmes utilisateurs.

Action IV.1. *Éditez le fichier `threads/system.cc`. Ajoutez une définition globale*

```
#ifdef CHANGED
#ifdef USER_PROGRAM
ConsoleDriver *consoledriver;
#endif
#endif
```

et dans `threads/system.h` la déclaration

```
#ifdef CHANGED
#ifdef USER_PROGRAM
extern ConsoleDriver *consoledriver;
#endif
#endif
```

Ensuite, ajoutez la création de l'objet dans `threads/main.cc` dans la fonction `main` juste avant l'appel à `StartProcess()` (passez `NULL, NULL` en paramètre pour simplifier), et sa destruction à la fonction C `Cleanup()` avant la destruction de la machine. Mettez à jour le fichier `system.h` en conséquence pour déclarer cet objet. Notez le `#ifdef USER_PROGRAM` : cette modification n'est faite que lorsque l'on souhaite exécuter un programme utilisateur, c'est-à-dire que l'on compile depuis `userprog`.

Maintenant il s'agit de mettre en place l'appel système.

Action IV.2. *Éditez le fichier `userprog/syscall.h` pour y rajouter un appel système `#define SC_PutChar ...` et la déclaration de la fonction `void PutChar(char c)` correspondante. Il s'agit ici de la fonction* utilisateur *Nachos : en terme Unix, `putchar(3)`.*

Il faut maintenant définir le code de la fonction `PutChar(char c)`. Comme celle-ci doit provoquer un déroutement (*trap*), ce code doit être écrit en assembleur.

Action IV.3. *Éditez le fichier `test/start.S` pour y rajouter la définition en assembleur de `PutChar`. Vous pouvez copier celle de `Halt`, en faisant attention de bien récupérer tout l'ensemble des lignes concernant `Halt`. Notez que l'on place le numéro de l'appel système*

dans le registre `r2` avant d'appeler l'instruction "magique" `syscall`. C'est le compilateur qui s'occupe pour vous de placer le premier argument `char c` dans le registre `r4`. Ce registre est un registre entier 32 bits : le caractère est donc implicitement converti : `r4 = (int)c`.

Vous pouvez désormais activer le bout de code de `putchar.c` appelant `PutChar`, cela devrait compiler !

Il faut maintenant mettre en place le traitant qui est activé par l'interruption `syscall` et utilise simplement notre driver de console.

Action IV.4. *Éditez le fichier `userprog/exception.cc`. Observez dans la fonction*

```
ExceptionHandler (ExceptionType which)
```

le `switch` C/C++, il y a déjà le cas de l'appel système `SC_Halt`, ajoutez à côté le cas `SC_PutChar` (entouré de `#ifdef CHANGED` bien sûr), et implémentez-le en utilisant votre console synchrone. Prenez déjà l'habitude de mettre un `DEBUG('s', "PutChar\n ");` au début du `case`, pour pouvoir facilement afficher quels appels systèmes sont appelés par un programme en passant simplement `-d s` à nachos. Il y aura de nombreuses exceptions possibles, il faudra à chaque fois y récupérer les paramètres depuis le monde MIPS et y écrire les résultats dans le monde MIPS, bien sûr ! Notez la présence d'`UpdatePC` pour incrémenter le compteur d'instruction : par défaut, on réactive l'instruction courante au retour d'une exception (notamment pour les défauts de page !).

Lancez `make` depuis `code`, et vérifiez que `putchar` fonctionne désormais...

Note : il est possible que les tests `./nachos -c` et `./nachos -sc` se conduisent dorénavant de manière bizarre, cela est dû au fait que selon votre implémentation l'entrée standard est dans ce cas utilisée à la fois par la `consoledriver` allouée pour les appels systèmes et la `console` des tests, il n'est pas utile d'essayer de corriger cela.

Partie V. Des caractères aux chaînes

Pour le moment, nous ne pouvons écrire qu'un seul caractère à la fois. Écrire une chaîne se résume à faire une suite d'écritures de caractères, bien sûr ! Le seul problème est que l'on ne dispose que d'un pointeur utilisateur vers la chaîne, et non pas d'un pointeur noyau...

Action V.1. *Occupons-nous déjà de la partie Linux : complétez la méthode `PutString` de la classe `ConsoleDriver`, qui travaille sur une chaîne Linux.*

Action V.2. *Écrivez une procédure similaire à `strcpy`,*

```
int copyStringFromMachine(int from, char *to, unsigned size)
```

qui copie une chaîne caractère par caractère, du monde utilisateur (MIPS) en partant de l'adresse `from`, vers le monde noyau en partant de l'adresse `to`, à l'aide de la méthode `ReadMem`, en s'arrêtant si elle rencontre un `'\0'`. Au plus `size` caractères doivent être écrits, ce sera donc typiquement la taille du tampon noyau. Un `'\0'` doit être forcé à la fin de la copie en dernière position pour garantir la sécurité du système. Le nombre de caractères écrits doit être retourné. Attention à l'argument `int *value` de ReadMem : pourquoi ne peut-on pas simplement passer un pointeur pointant à l'intérieur du tampon `to` ? Le

choix du fichier dans lequel mettre cette fonction est de votre ressort, il y a plusieurs endroits qui sont appropriés, à vous de motiver votre choix dans le rapport ! (note : une telle fonction pourrait reservir pour les prochains TPs).

Action V.3. *Ajoutez l'appel système* `PutString`*, qui utilise votre* `copyStringFromMachine` *puis la méthode* `PutString` *de l'objet* `consoledriver`*. On pourra utiliser un buffer local de taille* `MAX_STRING_SIZE`*, en déclarant cette constante dans le fichier* `threads/system.h`*. Pourquoi ne serait-il pas raisonnable d'allouer un buffer de la même taille que la chaîne MIPS ? Selon la manière dont vous l'avez alloué, veillez à ce que le buffer soit libéré une fois l'appel système terminé !*

Action V.4. *Montrez sur quelques exemples le comportement de votre implémentation, notamment en cas de chaîne trop longue (et essayez de corriger).*

Vérifiez bien que vous ne débordez pas de vos tampons ! Pour vous en assurer, non seulement utilisez valgrind, mais activez AddressSanitizer en décommentant les deux lignes `-fsanitize=address` *dans* `Makefile.dep`[1]

Pour rappel, la convention usuelle en C est que le paramètre `size` *est la taille totale du tampon, les fonctions ne doivent pas en dépasser.*

Partie VI. Mais comment s'arrêter ?

Action VI.1. *Que se passe-t-il si vous enlevez l'appel à* `Halt()` *à la fin de la fonction* `main` *de* `putchar.c` *? Décryptez le message d'erreur et expliquez. Que changer pour éviter cette erreur ? Vous n'aurez ainsi plus besoin d'appeler explicitement* `Halt()` *dans vos programmes. Comment faire pour prendre en compte la valeur de retour* `return n` *de la fonction* `main` *si celle-ci est déclarée à valeur entière ? (cherchez dans* `test` *qui appelle* `main` *et utilisez* `make putstring.s` *dans test/ pour voir où* `main` *met sa valeur de retour)*

Partie VII. Fonctions de lecture

Action VII.1. *Complétez l'appel système* `GetChar`*. Le registre utilisé pour le retour d'une valeur à la fin d'une fonction est le registre* `2` *: c'est là qu'il faut placer la valeur lue à la console. Que faites-vous en cas de fin de fichier ?*

Action VII.2. *Complétez la méthode* `GetString` *de la classe* `ConsoleDriver` *sur le modèle de* `fgets` *(lisez bien son manuel pour la gestion des caractères de fin de ligne et des débordements !)*

Action VII.3. *De même que pour* `GetChar`*, complétez l'appel système* `void GetString(char *s, int n)` *sur le modèle de* `fgets` *en suivant le même principe que* `PutString` *(il faudra donc définir une fonction* `copyStringToMachine`*). Attention : 1) Vous devez absolument garantir qu'il n'y a pas de débordement de tableau au niveau du noyau et au niveau utilisateur. 2) Vous devez au besoin désallouer toutes les structures temporaires allouées pour éviter les fuites mémoire. 3) Même si pour l'instant nous n'avons pas de threads utilisateurs, essayez de prendre en compte les appels concurrents dès maintenant : que se passerait-il si plusieurs threads appellaient en même temps cette fonction ?*

1. Note : cela va aussi activer LeakSanitizer qui vérifie les fuites mémoire. Il est alors possible qu'il râle sur une fuite de l'allocation de la pile du thread principal. Cette pile est effectivement forcément encore allouée à la terminaison du programme.

Action VII.4. *(Bonus) Mettez en place un appel système* `void PutInt(int n)` *qui écrit un entier signé en utilisant la fonction* `snprintf` *pour en obtenir l'écriture décimale prête à afficher sur la console. Idem dans l'autre sens avec* `void GetInt(int *n)` *et la fonction* `sscanf`*.*

Partie VIII. Bonus : un printf

Appeler `PutString` et `PutInt` successivement est fastidieux, on voudrait pouvoir appeler `printf`, comme d'habitude. Pourquoi est-ce une mauvaise idée de définir un appel système `Printf` ? C'est pour cela que l'on va plutôt simplement intégrer une fonction `printf` à l'espace utilisateur.

Action VIII.1. *Téléchargez les sources de Linux 2.2.26 depuis* `http://www.kernel.org/pub/linux/kernel/v2.2/`*, récupérez le fichier* `lib/vsprintf.c`*, commentez les inclusions des fichiers* `linux/*.h` *(*`stdarg.h`*, lui, est bien fourni par le cross-compilateur MIPS). En implémentant les fonctions* `isxdigit`*,* `isdigit`*,* `islower`*,* `toupper` *et* `strnlen`*, et en modifiant* `test/Makefile` *pour inclure automatiquement* `vsprintf.o` *à la liaison de vos programmes MIPS (comme* `start.o`*), mais en l'excluant de la liste PROGS (en utilisant la fonction* `filter-out` *de make), vous devriez pouvoir obtenir un* `vsprintf` *qui fonctionne, et de là implémenter un* `printf`*, voir* `man va_start` *pour initialiser le* `va_list`*.*