Questionnaire d'exploration de Nachos

1. Indiquer ce qui doit être sauvegardé lors d'un changement de contexte.

L'objectif du changement de contexte est de sauvegarder le contexte du thread appelant dans l'objet thread correspondant, et de restaurer celui du nouveau thread élu.

Le contexte du thread est constitué de l'état des registres de la machine MIPS : `thread_context.int_registers` et `thread_context.float_registers`.

```
/*! \brief Defines the thread context (MIPS virtual machine)
*/
typedef struct
{
//! Integer CPU register state (value of
//  MIPS registers)
int32_t int_registers[NUM_INT_REGS];

//! Floating point general purpose registers
int32_t float_registers[NUM_FP_REGS];

//! Condition code register.
int8_t cc;
} threadContextT;
```

2. Quelle variable est utilisée pour mémoriser la liste des threads prêts à s'exécuter ? Est-ce que le thread élu (actif) appartient à cette liste ? Comment accéder à ce thread ?

La variable est `readyList`, créé par le scheduler lors de sa construction :

```
Scheduler::Scheduler()
{
    readyList = new Listint;
}
```

`Listint` est de type :

```
typedef List<int> Listint;
```

List est une classe template défini dans `list.hpp/cpp`.

Le thread actif élu est supprimé de la liste.

On y accès via cette méthode :

```
//----------------------------------------------------------------------
// Scheduler::FindNextToRun
/*!     Return the next thread to be scheduled onto the CPU.
 // If there are no ready threads, return NULL.
 // Side effect:
 // Thread is removed from the ready list.
 // \return Thread to be scheduled on the CPU
 */
//----------------------------------------------------------------------
Thread*
Scheduler::FindNextToRun()
{
    Thread *thread = (Thread*)readyList->Remove();
    return thread;
}
```

3. A quoi sert la variable `g_alive` ? Quelle est la différence avec le champ `readyList` de l'objet g scheduler ?

`g_alive` est la liste des threads en vie à un moment donné, mais pas forcément en train d'être exécuté. `readyList` est la liste des threads pouvant être exécuté.

4. Comment se comportent les routines de gestion de listes vis à vis de l'allocation de mémoire ? Est-ce qu'elles se chargent d'allouer/désallouer les objets chaînés dans la liste ? Pourquoi ?

Je ne comprends pas la question.

5. A quel endroit est placé un objet thread quand il est bloqué sur un sémaphore ?

Le thread est replacé dans la liste des thread prêt à l'exécution.

6. Comment faire en sorte qu'on ne soit pas interrompu lors de la manipulation des structures de données du noyau ?

Pour cela, il faut faire en sorte que les interruptions ne soient pas prises en compte en utilisant :

```
g_machine->interrupt->SetStatus(INTERRUPTS_OFF);
```

7. A quoi sert la méthode `SwitchTo` de l'objet `g_scheduler` ? Quel est le rôle des variables `thread_context` et `simulator_context` de l'objet thread ? Que font les méthodes `SaveSimulatorState` et `RestoreSimulatorState` ? Que

devront (à terme) faire les méthodes `SaveProcessorState` et `RestoreProcessorState` de l'objet thread ?

Permet de passer la main du thread passé en paramètre au CPU. Sauvegarde l'état du thread courrant et l'état du simulateur Nachos.

Ces deux états sont ensuite envoyé au thread passé en paramètre.

`thread_context` contient le contexte du thread, les registres du thread.

`simulator_context` contient le contexte ainsi qu'un pointer sur le bas de la pile et sa taille, du CPU non simulé au moment où le swap est fait.

```
/*! \brief Defines the context of the Nachos simulator
 */
typedef struct
{
    ucontext_t buf;
    int8_t *stackBottom;
    int stackSize;
} simulatorContextT;
```

`SaveSimulatorState` appelle la méthode `getcontext()` pour remplire la structure pointée, avec le contexte actuellement actif du simulateur.

`RestoreSimulatorState` appelle la méthode `setcontext()` pour restaurer le contexte utilisateur du simulateur dans la structure pointé. Un appel réussi ne revient pas. Le contexte doit avoir été obtenu par un appel `getcontext()` ou `makecontext()`.

`SaveProcessorState` appelle la méthode `getcontext()` pour remplire la structure pointée, avec le contexte actuellement actif du CPU non simulé.

`RestoreProcessorState` appelle la méthode `setcontext()` pour restaurer le contexte utilisateur du CPU non simulé dans la structure pointé. Un appel réussi ne revient pas. Le contexte doit avoir été obtenu par un appel `getcontext()` ou `makecontext()`.

8. Expliquer l'utilité du champ type de tous les objets manipulés par le noyau (sémaphores, tâches, threads, etc.).

Permet de définir le type de l'objet. Les syscalls font en sorte que les objets passés sont du type attendu.

```
/*! Each syscall makes sure that the object that the user passes to it
 * are of the expected type, by checking the typeId field against
 * these identifiers
 */
typedef enum
```

```
{
    SEMAPHORE_TYPE = 0xdeefeaea,
    LOCK_TYPE = 0xdeefcccc,
    CONDITION_TYPE = 0xdeefcdcd,
    FILE_TYPE = 0xdeadbeef,
    THREAD_TYPE = 0xbadcafe,
    INVALID_TYPE = 0xf0f0f0f
} ObjectType;
```

Environnement de développement

1. Lister les outils offerts par Nachos pour la mise au point des programmes utilisateur. Comment par exemple visualiser toutes les opérations effectuées par la machine MIPS émulée ?

Nachos possède des outils de déboguage, de gestion de listes et de gestion de bitmaps.

Nachos offre certaines routines permettant d'afficher des messages facilitant le déboguage des fonctions et méthodes système lors de leur implantation. Il est possible de sélectionner le 'type' de message de déboguage que l'on désire systématiquement afficher. Chaque type de message est identifié par un drapeau (flag).

Pour afficher toutes les opérations il devra alors falloir passer en paramètre d'appel de Nachos un `+`.

```
//----------------------------------------------------------------------
// DEBUG
/*!      Print a debug message, if flag is enabled.  Like printf,
 // only with an extra argument on the front.
 */
//----------------------------------------------------------------------
void DEBUG(char flag, const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    if (DebugIsEnabled(flag))
    {
        // You will get an unused variable message here -- ignore it.
        vfprintf(stdout, format, ap);
        fflush (stdout);
    }

    va_end(ap);
}
```

Nachos propose une structure de listes à simple chaînage, ainsi que les accès classiques aux listes et à leurs eléments.

Nachos propose aussi une libc maison pour avoir accès aux fonctions les plus nécessaires.

2. Peut-on utiliser l'utilitaire `gdb` pour mettre au point le code de Nachos ? Le lancer et visualiser le contenu de différentes variables du noyau.

Oui on peut utiliser `gdb` pour mettre au point le code de Nachos.

```
...
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from nachos...
(gdb) break Thread::Start
Breakpoint 1 at 0x6512: file thread.cpp, line 102.
(gdb) run
Starting program: /home/yberion/Bureau/GitHub/M1-Nachos/nachos
/usr/lib/../share/gcc-9.2.0/python/libstdcxx/v6/xmethods.py:731: SyntaxWarning: list indices m
  refcounts = ['_M_refcount']['_M_pi']

Breakpoint 1, Thread::Start (this=0x55555559b7c0, owner=0x55555559b5c0, func=0, arg=-1) at thr
102         ASSERT(process == NULL);
(gdb) print owner
$1 = (Process *) 0x55555559b5c0
(gdb) print this
$2 = (Thread * const) 0x55555559b7c0
(gdb) info frame
Stack level 0, frame at 0x7fffffffdce0:
 rip = 0x55555555a512 in Thread::Start (thread.cpp:102); saved rip = 0x555555559a38
 called by frame at 0x7fffffffdda0
 source language c++.
 Arglist at 0x7fffffffdcd0, args: this=0x55555559b7c0, owner=0x55555559b5c0, func=0, arg=-1
 Locals at 0x7fffffffdcd0, Previous frame's sp is 0x7fffffffdce0
 Saved registers:
  rbp at 0x7fffffffdcd0, rip at 0x7fffffffdcd8
(gdb) bt
#0  Thread::Start (this=0x55555559b7c0, owner=0x55555559b5c0, func=0, arg=-1) at thread.cpp:10
#1  0x0000555555559a38 in Initialize (argc=0, argv=0x7fffffffdee0) at system.cpp:190
#2  0x000055555555756b in main (argc=1, argv=0x7fffffffded8) at main.cpp:54
(gdb) jump *0x000055555555756b
Line 55 is not in `Thread::Start(Process*, int, int)'.  Jump anyway? (y or n) y
Continuing at 0x55555555756b.

Program received signal SIGSEGV, Segmentation fault.
0x000055555555759a in main (argc=32766, argv=0x8) at main.cpp:62
62              if (!strcmp(*argv, "-z"))
```

```
(gdb) print argv
$3 = (char **) 0x8
```

3. Peut-on utiliser `gdb` pour mettre au point les programmes utilisateur ? Expliquer.

Oui, il existe `gdb-multiarch` qui permet de pouvoir déboguer n'importe quel programme (avec architectures différentes) sous n'importe quel contexte, dans notre cas le noyau et le simulateur sont une seul et même entité donc il est plus aisé pour le débogage.