Correction
-
TP 1
Langages, techniques et outils

Le générateur de données

Rappel de la déclaration de la fonction fibonacci

```
/**
* Remplit un tableau avec la suite de Fibonnacci.
* @param output_array tableau où la suite est enregistrée
* @param size nombre de valeur à inscrire dans \a output_array
* @param min_value valeur jusqu'à laquelle les nombres ne sont pas enregistrés
*/
void fibonacci(uint32_t output_array[], uint32_t size, uint32_t min_value);
```

Définition de la fonction fibonacci

```
void fibonacci(uint32_t output_array[], uint32_t size, uint32_t min_value) {
	// initialise les variables
	uint32_t n1 = 0;
	uint32_t n2 = 1;
	uint32_t n3;

	// parcourt la suite tant qu'inférieur à min_value
	while(n1 < min_value){
		n3 = n1+n2;
		n1 = n2;
		n2 = n3;
	}

	// renseigne les valeurs déjà parcourues
	if (size > 0) {
		output_array[0] = n1;
	}
	if (size > 1) {
		output_array[1] = n2;
	}

	// parcourt la suite et l'enregistre dans output_array
	if (size > 2) {
		for (unsigned int i = 2; i < size; i++) {
			output_array[i] = output_array[i - 2] + output_array[i - 1];
		}
	}
}
```

Fichier main

```
#include <stdint.h>
#include "fibonacci.h"

#define DATA_SIZE 20

uint32_t input[DATA_SIZE];

void test(unsigned int val){
	static int cnt = 0;
	// la fonction fait quelque chose pour éviter que l'optimisation du compilateur
ne la supprime.
	cnt++;
}

int main(void) {
	// initialise le tableau global
	for (int i = 0; i < DATA_SIZE; i++) {
		input[i] = 0;
	}

	fibonacci((float*) input , DATA_SIZE, 1000);

	// pour appel du point d'arrêt
	test(1);

	return 0 /*EXIT_SUCCESS*/;
}
```

Examinez des données grâce à un script GDB

Voici une version minimale de script GDB nécessaire à l'affichage des tableaux globaux `input` et `output` avant et après le calcul des périmètres. Il faut pour cela appeler la fonction `test()` avant et après cette fonction.

```
set logging file gdb_result_001.txt
set height 0
set print array on

tar extended-remote localhost:1234
load

break test
commands
	p input
	p output
cont
end

start
cont
```

Créez des sections

Le passage suivant est ajouté au linker script par défaut, après la section `bss`, pour définir une section `bss_array`.

```
  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.
      FIXME: Why do we need it? When there is no .bss section, we do not
      pad the .data section.  */
   . = ALIGN(. != 0 ? 8 : 1);
  }
  .bss_array 0x40006000 : {
    . = ALIGN(0x8);
    *(.bss .bss.* )
    *(COMMON)
  }
```

Les tableaux globaux sont déclarés ainsi pour apparaître dans la nouvelle section.

```
#define DATA_SIZE 20

uint32_t input[DATA_SIZE] __attribute__ ((section (".bss_array")));
float output[DATA_SIZE] __attribute__ ((section (".bss_array")));
```

La partie suivante du linker map fait référence à la nouvelle section.

```
.bss_array      0x0000000040006000       0xa0
                0x0000000040006000                . = ALIGN (0x8)
 *(.bss .bss.*)
 *(COMMON)
 .bss_array     0x0000000040006000       0xa0 src/TP1.o
                0x0000000040006000                input
                0x0000000040006050                output
                0x00000000400060a0                . = ALIGN (0x8)
                0x00000000400060a0                . = SEGMENT_START ("ldata-
segment", .)
                0x00000000400060a0                . = ALIGN (0x8)
                0x00000000400060a0                _end = .
                [!provide]                        PROVIDE (end = .)
                0x00000000400060a0                . = DATA_SEGMENT_END (.)
```

On y apprend que le tableau `input` est à l'adresse `0x40006000` et que le tableau `output` est à l'adresse `0x40006050`.

La partie suivante du linker map fait référence aux fonctions développées dans ce TP.

```
.text          0x0000000040001250       0xfc src/TP1.o
               0x0000000040001250               test
               0x0000000040001284               main
.text          0x000000004000134c      0x13c src/fibonacci.o
               0x000000004000134c               fibonacci
.text          0x0000000040001488       0x90 src/perimeter.o
               0x0000000040001488               perimeter
```

Pour la fonction `perimeter`, on apprend qu'elle est à l'adresse `0x40001488` et qu'elle occupe 144 octets (0x90 en hexadécimal).

Générer fichier SREC

Voici le début du fichier SREC généré à partir de mon exécutable.

```
S011000044656275672F5450312E737265632800
S3154000000088100000091000058 1C12118010000078
S3154000001091D020000100000001000000010000016
S3154000002091D020000100000001000000010000006
```

Ça confirme que le programme est bien chargé à l'adresse 0X40000000.

Voici la fin du fichier SREC en question.

```
S3154000608000000000000000000000000000000CA
S3154000609000000000000000000000000000000BA
S705400000000BA
```

On y apprend que le dernier octet chargé l'est à l'adresse 0X4000609F, c'est à dire l'adresse du premier octet (0X40006090) de la dernière séquence de données (S3) plus le nombre d'octets de la ligne (16).

Optimiser et débuguer

En plaçant différents point d'arrêt dans le programme on découvre que l'erreur de segmentation se produit sur l'instruction de `perimeter()` suivante :

```
Thread 1 hit Breakpoint 2, perimeter (in=0x40006000 <input>,
    out=0x40006050 <output>, size=20) at ../src/perimeter.c:7
7                   out[i] = pi * in[i];
$2 = 3.14159274
$3 =   {2.23787365e-42}

Thread 1 received signal SIGSEGV, Segmentation fault.
```

`pi` y vaut 3.14159274 et `in[0]` 2.23787365e-42

Grâce à l'outil objdump, et avec les options -d et -S, on en déduit que l'erreur se produit sur une des instructions assembleur suivantes :

```
      out[i] = pi * in[i];
  2c:   c2 07 bf fc    ld  [ %fp + -4 ], %g1
  30:   83 28 60 02    sll  %g1, 2, %g1
  34:   c4 07 a0 44    ld  [ %fp + 0x44 ], %g2
  38:   82 00 80 01    add  %g2, %g1, %g1
  3c:   d3 00 40 00    ld  [ %g1 ], %f9
  40:   c2 07 bf fc    ld  [ %fp + -4 ], %g1
  44:   83 28 60 02    sll  %g1, 2, %g1
  48:   c4 07 a0 48    ld  [ %fp + 0x48 ], %g2
  4c:   82 00 80 01    add  %g2, %g1, %g1
  50:   d1 07 bf f8    ld  [ %fp + -8 ], %f8
  54:   91 a2 49 28    fmuls  %f9, %f8, %f8
  58:   d1 20 40 00    st  %f8, [ %g1 ]
```

L'erreur de segmentation ne se produisant qu'après avoir activé le GRFPU dans TSIM, on peut en déduire que c'est d'une instruction gérée par ce composant que provient le problème. Or la seule instruction qui corresponde est le `fmuls` chargé de la multiplication de 3.14159274 par 2.23787365e-42.

La norme IEE-754 qui défini l'encodage des nombres flottant, nous indique que les nombres sous $1{,}175\,494\,21 \times 10^{-38}$ appartiennent aux dénormalisés. Or on trouve dans la documentation du GRFPU que :

> « All operations are IEEE-754 compliant, except for denormalized numbers that can rise exceptions or be automatically flushed to zero. »

L'erreur venant des données dénormalisés injectés dans la fonction perimeter(), différents solutions peuvent être envisagées : un autre générateur de données, un contrôle des données dans la fonction perimeter() ou encore configurer le GRFPU pour qu'il fasse un arrondi à 0 plutôt qu'une erreur de segmentation.