Lab 3

Letong Han, 2021533062

1 Kernel

```
// * kernel code
auto fin = std::ifstream("kernel");
uint32_t inst;
int offset = 0;
while (fin >> std::hex >> inst) {
  simulator.memory->setInt(kernel_code_base + offset, inst);
  offset += 4;
}
```

the kernel code has to be placed alongside the Simulator executable.

1.1 Kernel Stack

To execute a syscall, we have to make `sp` point to the kernel stack

Kernel stack base address: 0x9000000

Kernel stack maximum size: 0x400000

```
if ((!kernel_mode &&
     this->reg[REG_SP] < this->stackBase - this->maximumStackSize) ||
    (kernel_mode &&
     this->reg[REG_SP] < this->kernelStackBase - this->kernelStackSize)) {
  this->panic("Stack Overflow!\n");
}
```

The stack overflow detection code has to be modified, too. (Although our kernel code does not use kernel stack)

1.2 Kernel Code

The instructions of syscall are stored in the following position

Kernel code base address: 0xA0000000

Kernel code maximum size: 0x100000

The kernel code is translated from the following c code

```
int syscall(int *a, int len) {
  a[0] = 114514;
  a[1] = 1919;
  a[2] = 13;
  a[3] = 998244353;
  a[4] = 1000000007;
  a[5] = 810;
  a[6] = 2341;
  a[7] = 17;
```

```c
  int max = -2147483647 - 1;
  for (int i = 0; i < len; ++i) {
    if (a[i] > max) {
      max = a[i];
    }
  }
  return max;
}
```

which is

```
0000000000010268 <syscall>:
   10268:	0001c737          	lui	a4,0x1c
   1026c:	f5270713          	add	a4,a4,-174
   10270:	00e52023          	sw	a4,0(a0)
   10274:	77f00713          	li	a4,1919
   10278:	00e52223          	sw	a4,4(a0)
   1027c:	1d100713          	li	a4,465
   10280:	00e52423          	sw	a4,8(a0)
   10284:	3b800737          	lui	a4,0x3b800
   10288:	00170713          	add	a4,a4,1
   1028c:	00e52623          	sw	a4,12(a0)
   10290:	3b9ad737          	lui	a4,0x3b9ad
   10294:	a0770713          	add	a4,a4,-1529
   10298:	00e52823          	sw	a4,16(a0)
   1029c:	32a00713          	li	a4,810
   102a0:	00e52a23          	sw	a4,20(a0)
   102a4:	00001737          	lui	a4,0x1
   102a8:	92570713          	add	a4,a4,-1755
   102ac:	00e52c23          	sw	a4,24(a0)
   102b0:	01100713          	li	a4,17
   102b4:	00e52e23          	sw	a4,28(a0)
   102b8:	00050793          	mv	a5,a0
   102bc:	02b05863          	blez	a1,102ec <syscall+0x84>
   102c0:	00259593          	sll	a1,a1,0x2
   102c4:	00b50633          	add	a2,a0,a1
   102c8:	80000537          	lui	a0,0x80000
   102cc:	0007a703          	lw	a4,0(a5)
   102d0:	00478793          	add	a5,a5,4
   102d4:	00070693          	mv	a3,a4
   102d8:	00a75463          	bge	a4,a0,102e0 <syscall+0x78>
   102dc:	00050693          	mv	a3,a0
   102e0:	0006851b          	sext.w	a0,a3
   102e4:	fef614e3          	bne	a2,a5,102cc <syscall+0x64>
   102e8:	00008067          	ret
   102ec:	80000537          	lui	a0,0x80000
   102f0:	00008067          	ret
```

1.3 Save Context

Before executing a syscall, we have to save the context to the following position:

Context save base address: 0x9000000

Context maximum size: 0x100000

The context includes:

```
x1-x31
f0-f32
pc
```

1.4 Restore Context

Since we have to use `ret` to return from syscall, we used a special address to notify the system to restore context.

Kernel restore addr: 0xC0000000

After saving context and before executing the instructions in syscall, we modify `ra` to kernel restore addr.
Thus, when we return from syscall, `pc` will be set to 0xC0000000 and this special `pc` will trigger the system to restore context.

2 Trap Handler

We implemented this lab with scoreboard (based on lab1).

2.0 Use the ecall as a trigger for the syscall.

Yes, to trigger this syscall, `a7` should be 7.
The parameters are stored in `a0` and `a1`, where `a0` holds the address of array and `a1` holds the length of array.

2.1 The program state can be saved in one of the known manners.

Yes, as stated in 1.3, it will be save to a dedicated place in memory.

2.3 Make sure that no instruction beyond the instruction that triggers the syscall is factually committed.

Yes, when `ecall` is issued, instruction fetch & decode are stalled. So no instruction beyond the instruction that triggers the syscall is issued, let alone committed.

2.2 Make sure that all instructions before the instruction that triggers the syscall have successfully executed and committed.

Yes, since we prevent instruction issuing beyong `ecall`, to make sure that all instructions before the instruction that triggers the syscall have successfully executed and committed, we do not execute `ecall` until all other issued instructions haved committed.

The combination of 2.3 and 2.4 ensures precise trap.

2.4 You should track and save the state before executing the instruction that triggers the syscall.

Yes, it will be saved before executing syscall (1.3) and restored after syscall returned using the mechanism mentioned in 1.4.

2.5 (Optional) One bonus point would be given to students who implement and run tests with out-of-order execution (scoreboard).

Yes, it is out-of-order execution (scoreboard).

3 Tests

3.1 test1.c

```
#include "lib.h"

int a[20];

int main() {
  int len = 20;
  for (int i = 0; i < len; ++i) {
    a[i] = i;
  }
  for (int i = 0; i < len; ++i) {
    print_d(a[i]);
    print_c(' ');
  }
  print_c('\n');

  asm("add a0, zero,%[a];"
      "add a1, zero,%[len];"
      "li a7, 7;"
      "ecall;"
      "li a7, 2;"
      "ecall;" ::[a] "r"(a),
      [len] "r"(len));

  print_c('\n');
  for (int i = 0; i < len; ++i) {
    print_d(a[i]);
    print_c(' ');
  }
  print_c('\n');

  exit_proc();
}
```

its output is

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1000000007
114514 1919 465 998244353 1000000007 810 2341 17 8 9 10 11 12 13 14 15 16 17 18
19
```

3.2 test2.c

```c
#include "lib.h"

int a[20];

int main() {
  int len = 20;
  for (int i = 0; i < len; ++i) {
    a[i] = i;
  }
  a[13] = 2147483647;
  for (int i = 0; i < len; ++i) {
    print_d(a[i]);
    print_c(' ');
  }
  print_c('\n');

  asm("add a0, zero,%[a];"
      "add a1, zero,%[len];"
      "li a7, 7;"
      "ecall;"
      "li a7, 2;"
      "ecall;" ::[a] "r"(a),
      [len] "r"(len));

  print_c('\n');
  for (int i = 0; i < len; ++i) {
    print_d(a[i]);
    print_c(' ');
  }
  print_c('\n');

  exit_proc();
}
```

its output is

```
0 1 2 3 4 5 6 7 8 9 10 11 12 2147483647 14 15 16 17 18 19
2147483647
114514 1919 465 998244353 1000000007 810 2341 17 8 9 10 11 12 2147483647 14 15 16
17 18 19
```

3.3 Save&Load Context

Use test2.c as an example, we will show that we correctly saved & restored context.

syscall instruction is at 1018c

```
                 ......
   10180:   01300533          add a0,zero,s3
   10184:   00f005b3          add a1,zero,a5
   10188:   00700893          li  a7,7
   1018c:   00000073          ecall
                 ......
```

Thanks to the -v option of the simulator, we can inspect cpu's state before and after the syscall.

Before:

```
Fetched instruction 0x00200893 at address 0x10190
Decoded instruction 0x00000073 as ecall
Issued
------------ CPU STATE ------------
PC: 0x10194
zero: 0x00000000(0) ra: 0x0001017c(65916) sp: 0x7fffffd0(2147483600) gp:
0x00011f20(73504)
tp: 0x00000000(0) t0: 0x000103f0(66544) t1: 0x0000000f(15) t2: 0x00000000(0)
s0: 0x00011eb8(73400) s1: 0x00011f08(73480) a0: 0x00011eb8(73400) a1:
0x00000014(20)
a2: 0x00000000(0) a3: 0x00000014(20) a4: 0x00011f08(73480) a5: 0x00000014(20)
a6: 0x0000001f(31) a7: 0x00000007(7) s2: 0x00011f08(73480) s3: 0x00011eb8(73400)
s4: 0x00000000(0) s5: 0x00000000(0) s6: 0x00000000(0) s7: 0x00000000(0)
s8: 0x00000000(0) s9: 0x00000000(0) s10: 0x00000000(0) s11: 0x00000000(0)
t3: 0x00000000(0) t4: 0x00000000(0) t5: 0x00000000(0) t6: 0x00000000(0)
ft0: (0.000000) ft1: (0.000000) ft2: (0.000000) ft3: (0.000000)
ft4: (0.000000) ft5: (0.000000) ft6: (0.000000) ft7: (0.000000)
fs0: (0.000000) fs1: (0.000000) fa0: (0.000000) fa1: (0.000000)
fa2: (0.000000) fa3: (0.000000) fa4: (0.000000) fa5: (0.000000)
fa6: (0.000000) fa7: (0.000000) fs2: (0.000000) fs3: (0.000000)
fs4: (0.000000) fs5: (0.000000) fs6: (0.000000) fs7: (0.000000)
fs8: (0.000000) fs9: (0.000000) fs10: (0.000000) fs11: (0.000000)
ft8: (0.000000) ft9: (0.000000) ft10: (0.000000) ft11: (0.000000)
-----------------------------------
```

Context saved, `ra` points to kernel restore addr, `sp` points to kernel stack, control flow redirected to kernel (0xa0000000):

```
Fetched instruction 0x00000073 at address 0x10194
Decode: Stall
Execute: ecall
------------ CPU STATE ------------
PC: 0xa0000000
zero: 0x00000000(0) ra: 0xc0000000(3221225472) sp: 0x09000000(150994944) gp:
0x00011f20(73504)
tp: 0x00000000(0) t0: 0x000103f0(66544) t1: 0x0000000f(15) t2: 0x00000000(0)
```

```
s0: 0x00011eb8(73400) s1: 0x00011f08(73480) a0: 0x00011eb8(73400) a1:
0x00000014(20)
a2: 0x00000000(0) a3: 0x00000014(20) a4: 0x00011f08(73480) a5: 0x00000014(20)
a6: 0x0000001f(31) a7: 0x00000007(7) s2: 0x00011f08(73480) s3: 0x00011eb8(73400)
s4: 0x00000000(0) s5: 0x00000000(0) s6: 0x00000000(0) s7: 0x00000000(0)
s8: 0x00000000(0) s9: 0x00000000(0) s10: 0x00000000(0) s11: 0x00000000(0)
t3: 0x00000000(0) t4: 0x00000000(0) t5: 0x00000000(0) t6: 0x00000000(0)
ft0: (0.000000) ft1: (0.000000) ft2: (0.000000) ft3: (0.000000)
ft4: (0.000000) ft5: (0.000000) ft6: (0.000000) ft7: (0.000000)
fs0: (0.000000) fs1: (0.000000) fa0: (0.000000) fa1: (0.000000)
fa2: (0.000000) fa3: (0.000000) fa4: (0.000000) fa5: (0.000000)
fa6: (0.000000) fa7: (0.000000) fs2: (0.000000) fs3: (0.000000)
fs4: (0.000000) fs5: (0.000000) fs6: (0.000000) fs7: (0.000000)
fs8: (0.000000) fs9: (0.000000) fs10: (0.000000) fs11: (0.000000)
ft8: (0.000000) ft9: (0.000000) ft10: (0.000000) ft11: (0.000000)
-----------------------------------
```

......

......

......

syscall returns to 0xc0000000, our special address for context restore:

```
Fetched instruction 0x00008067 at address 0xa0000088
Decode: Stall
Execute: jalr
------------ CPU STATE ------------
PC: 0xc0000000
zero: 0x00000000(0) ra: 0xc0000000(3221225472) sp: 0x09000000(150994944) gp:
0x00011f20(73504)
tp: 0x00000000(0) t0: 0x000103f0(66544) t1: 0x0000000f(15) t2: 0x00000000(0)
s0: 0x00011eb8(73400) s1: 0x00011f08(73480) a0: 0x3b9aca07(1000000007) a1:
0x00000050(80)
a2: 0x00011f08(73480) a3: 0x3b9aca07(1000000007) a4: 0x00000013(19) a5:
0x00011f08(73480)
a6: 0x0000001f(31) a7: 0x00000007(7) s2: 0x00011f08(73480) s3: 0x00011eb8(73400)
s4: 0x00000000(0) s5: 0x00000000(0) s6: 0x00000000(0) s7: 0x00000000(0)
s8: 0x00000000(0) s9: 0x00000000(0) s10: 0x00000000(0) s11: 0x00000000(0)
t3: 0x00000000(0) t4: 0x00000000(0) t5: 0x00000000(0) t6: 0x00000000(0)
ft0: (0.000000) ft1: (0.000000) ft2: (0.000000) ft3: (0.000000)
ft4: (0.000000) ft5: (0.000000) ft6: (0.000000) ft7: (0.000000)
fs0: (0.000000) fs1: (0.000000) fa0: (0.000000) fa1: (0.000000)
fa2: (0.000000) fa3: (0.000000) fa4: (0.000000) fa5: (0.000000)
fa6: (0.000000) fa7: (0.000000) fs2: (0.000000) fs3: (0.000000)
fs4: (0.000000) fs5: (0.000000) fs6: (0.000000) fs7: (0.000000)
fs8: (0.000000) fs9: (0.000000) fs10: (0.000000) fs11: (0.000000)
ft8: (0.000000) ft9: (0.000000) ft10: (0.000000) ft11: (0.000000)
-----------------------------------
```

Context restored:
Note that `a0` holds the return value.

PC points to 0x10194 since 0x10194=0x1018c+4+4, the PC was first set to the next instruction after the syscall, which is 0x10190, and then +=4 in Simulator::Fetch(). The instruction fetched in IF stage is still 0x10190.

```
Fetched instruction 0x00200893 at address 0x10190
Decode: Bubble
WriteBack: jalr
0: -1610612604 0.000000
------------ CPU STATE ------------
PC: 0x10194
zero: 0x00000000(0) ra: 0x0001017c(65916) sp: 0x7fffffd0(2147483600) gp:
0x00011f20(73504)
tp: 0x00000000(0) t0: 0x000103f0(66544) t1: 0x0000000f(15) t2: 0x00000000(0)
s0: 0x00011eb8(73400) s1: 0x00011f08(73480) a0: 0x3b9aca07(1000000007) a1:
0x00000014(20)
a2: 0x00000000(0) a3: 0x00000014(20) a4: 0x00011f08(73480) a5: 0x00000014(20)
a6: 0x0000001f(31) a7: 0x00000007(7) s2: 0x00011f08(73480) s3: 0x00011eb8(73400)
s4: 0x00000000(0) s5: 0x00000000(0) s6: 0x00000000(0) s7: 0x00000000(0)
s8: 0x00000000(0) s9: 0x00000000(0) s10: 0x00000000(0) s11: 0x00000000(0)
t3: 0x00000000(0) t4: 0x00000000(0) t5: 0x00000000(0) t6: 0x00000000(0)
ft0: (0.000000) ft1: (0.000000) ft2: (0.000000) ft3: (0.000000)
ft4: (0.000000) ft5: (0.000000) ft6: (0.000000) ft7: (0.000000)
fs0: (0.000000) fs1: (0.000000) fa0: (0.000000) fa1: (0.000000)
fa2: (0.000000) fa3: (0.000000) fa4: (0.000000) fa5: (0.000000)
fa6: (0.000000) fa7: (0.000000) fs2: (0.000000) fs3: (0.000000)
fs4: (0.000000) fs5: (0.000000) fs6: (0.000000) fs7: (0.000000)
fs8: (0.000000) fs9: (0.000000) fs10: (0.000000) fs11: (0.000000)
ft8: (0.000000) ft9: (0.000000) ft10: (0.000000) ft11: (0.000000)
-----------------------------------
```