计算机组成原理第二次实验报告

2013012245 白可

June 12, 2016

1 第一部分

1.1 题目

增加IADDL（iaddl 立即数，目标寄存器）与LEAVE指令。

1.2 思路

IADDL:

icode:fun ←M_1[PC]
rA:rB ←M_1[PC+1]
valC ←M_4[PC+2]
valP ← $PC + 6$
$valB \leftarrow R[rB]$
$valE \leftarrow valB + valC$
$setCC$
$R[rB] = valE$
$PC \leftarrow valP$

LEAVE:

等价于： movl %ebp %esp
pop %ebp
icode:fun ←M_1[PC]
valP ← $PC + 1$
$valA \leftarrow R[\%ebp]$
$valB \leftarrow R[\%ebp]$
$valE \leftarrow valB + 4$
$valM$ ←M_4[valA]
R[%esp] ← $valA$
$R[\%ebp] \leftarrow valE$
$PC \leftarrow valP$

1.3 实现

具体的内容为修改pipe-full.hcl中的内容。观察流水线的五个阶段，根据上面两个的论述，修改在流水线每个阶段需要的内容。

2 第二部分

2.1 题目

修改Y86源代码，可同时运行两个Y86流水线处理器，这两个处理器共享memory且分别具有私有的L1 cache。 每个处理器分别运行同一个测试程序，也即同一个测试程序的两个实例分别在这两个处理器上运行，互不冲突。同时要求这两个程序实例能够通过共享内存相互通信。具体的通信模式为ping / pong通信，也即实例A“发送”一个数据给实例B，同时等待B的“回复”数据，B收到后给A回复数据，再继续这个过程，收发过程至少需要100次；同时每次“发送”、“回复“的数据长度应该有变化（介于1个int到1024个int之间）。

2.2 模拟器思路

获得两个进程可以有三种形式，一种为在进程的开始进行fork。进行父进程与子进程的通讯，但是这种形势由于在同一个command打印信息顺序的混乱，这种方法较为直观，调试起来较为困难；一种为通过第三个进程对两个进程进行连接，这种方法较为直观且调试容易。另一种为直接使用两个进程，由于初始化的原因，实现起来稍微较第二种复杂。

需要先实现L1 cache。 这里我实现的是较为复杂的组相连高速缓存。

为了使两个进程能够通讯，需要将每个进程的运行栈分成不同的区域。这里我将其分成了四个区域：代码段（例如：私有0x0 – 0x7f0），接收数据存储段（私有0x8f0 – 0x1000），进程通讯数据段（映射至共享内存0x1000 – 0x1f00），进程通讯信息段（映射至共享内存0x1f00–0x1fff），划分形式不唯一，保证对齐即可。

上面所说的进程通讯段，使用mmap函数进行栈内存与共享内存的绑定。

为了保证在读写共享内存的操作原子性，在每次读写共享内存的时候，先要得到flock的排斥锁的加锁权限，才能对共享内存进行读写。

在共享内存里面写的内容主要有两部分，一部分是需要传输的数据，另一部分显示了两方中任何一方的信息收、发情况。两个进程通过对共享内存的读取，先通过第二部分获得消息，然后根据第二部分得到的信息的结果对第一部分内存进行操作。

接下来，我将对上面几个部分进行具体的论述。

2.2.1 L1 cache的实现

L1 cahce采用组向量机制，为$8x8x8$, 即有八个组(cache set)，每个组有8个tag(cache line), 每个tag 拥有8个block.

用一个结构体来声明了L1 cache，其中包含flag (valid位)，dirt （dirty位），tags 和 cache block. 由于block有8(2^3)个，set 有8(2^3)个，因此tag 的后10位(16-3-3)有效。

```
typedef struct {
  bool_t flag;
  bool_t dirt;
// last 10 bytes for tags.
  int tags;
// eight blocks
  byte_t contents[8];
}cache_line, *cache_line_t;

typedef struct {
```

```
 cache_line cache_matrix[8][8];
}cache_set, *cache_set_t;
```

2.2.2 内存的划分

一、代码段（例如：私有0x0 – 0x7f0）：存放程序栈。

二、接收数据存储段（私有0x8f0 – 0x1000）：该部分存在的目的是，在测试的时候，为了验证得到的结果的正确性，将最后一次收到的数据放在这个区域里。在进程结束的时候，通过观察该私有区域的数据情况，判断通讯是否正常进行。

三、进程通讯数据段（映射至共享内存0x1000 – 0x1c00）：该部分每次发送数据的时候，会将数据写进这个范围的地址内（但会先经过cache部分）。

四、进程通讯信息段（映射至共享内存0x1c00–0x1fff）：该部分存储两个进程的通讯信息。

消息的内容包含：该进程ID，发送消息的类型（读或写），访问内存的地址。

发送消息：一个进程在发送一条消息（即将信息写入该进程通讯信息段），读类型或写类型的消息无功能区分性（即操作相同，区别于“读取”操作对不同类型的消息的不同反应）。此时另一方无法对内存进行操作，该进程会发生阻塞，等待直到对方回复。

读取消息：当另一进程，试图内存进行读写前，均会被强制检查这个区域是否有未读的消息。如果有未读的消息：若该消息为写消息，则观察该地址是否在cache中，若存在，将该地址位置为dirty（此时缓存已经与内存不一致），若不存在，直接返回；若该消息为读消息，则观察该地址是否在cache中，若存在，且存在dirty位（说明内存需要更新），则将该cacheline写入内存中。若不存在或不存在dirty为，则直接返回。

2.2.3 内存共享机制

这部分使用了mmap函数，使用这个函数，主要包括几部分。先得到文件句柄，将该文件设为共享，于是在任何位置，当打开文件并使用mmap后，均可对该部分内存进行操作。

```
        byte_t *shared;
        int fd;
        if ((fd = open(SHARE_FILE, O_RDWR | O_CREAT, FILE_MODE)) == -1)
            {
             printf("open error: %s\n", strerror(errno));
             exit(EXIT_FAILURE);
        }
        if (( shared = mmap(NULL, SHARE_SIZE,PROT_READ | PROT_WRITE,
            MAP_SHARED| MAP_FILE, fd, 0)) == MAP_FAILED) {
             printf("open error: %s\n", strerror(errno));
             exit(EXIT_FAILURE);
        }
        close(fd);
```

为了将该指向共享内存部分与Memory联系起来。在mem中声明了file_mem_t结构体。通过file_mem_t,将普通内存与共享内存联系起来。

```
typedef struct {
  int len;
  word_t maxaddr;
```

```
  byte_t *contents;
  file_mem_t m_file_mem_t;
} mem_rec, *mem_t;
```

从前程序通过set/get_word_value对内存操作。现在，当程序试图读写共享内存部分的时候，就通过指向共享内存的指针，直接对共享内存进行操作。如下一段代码，携带file的均为对共享内存的操作。

```
bool_t get_byte_val_file(mem_t m,word_t pos, word_t * dest);
bool_t get_byte_val(mem_t m, word_t pos, word_t *dest);

bool_t get_word_val_file(mem_t m,word_t pos, word_t * dest);
bool_t get_word_val(mem_t m, word_t pos, word_t *dest);

bool_t set_byte_val_file(mem_t m,word_t pos, byte_t val);
bool_t set_byte_val(mem_t m, word_t pos, byte_t val);

bool_t set_word_val_file(mem_t m,word_t pos, word_t val);
bool_t set_word_val(mem_t m, word_t pos, word_t val);
```

2.2.4 读写操作原子性

使用flock函数。在每次需要读写内存之前，检查某文件是否加锁，如果没有加锁，那么直接读写。如果加锁，则等待直到锁解除。

```
int fd_lock = open(LOCK_FILE, O_RDWR | O_CREAT |O_TRUNC, S_IRUSR |
    S_IWUSR);
flock(fd_lock, LOCK_EX|LOCK_NB);
```

2.3 消息收发机制

上面讨论的内容都是在模拟器的层次上面，由于cache 的存在，在每次需要读写数据之时，保证两个进程看到的内存不一定是最新更新的数据。模拟器做那些工作的目的是，让两个进程看到的内存是相同的。

在汇编代码层次，仍然有需要考虑的问题。

发送者问题：这里加入一条机器指令rmchange.即将读取某个位置的值并将其替换，该位为共享内存。这样后来的就能够得到消息。

pop %ebp

icode:fun $\leftarrow$M$_1$[PC]

rA:rB $\leftarrow$M$_1$[PC+1]

valC $\leftarrow$M$_4$[PC+2] valP $\leftarrow PC + 6$

$valA \leftarrow R[rA]$

$valB \leftarrow R[rB]$

$valE \leftarrow valB + valC$

$R[rA] \leftarrow$M$_4$[valE]

M_4[valE] $\leftarrow R[rA]$

$PC \leftarrow valP$

2.4 结果说明

运行的时候，先运行 ./connect,实现共享内存的初始化。 然后将 ./misc/yas ./y86-code/filename.ys 然后分别开启两个进程，分别运行./pipe/psim ./test/filename.yo

对于第一部分，filename 为rmchange,第二部分，filename为sendreceive.

2.4.1 rmchange的实现

通过观察不同进程的最后的%edx是否为零，以及%ecx的值，观察程序是否进行了正常的跳转。实验结果正常。

这也一定程度能够说明，通信机制基本正确（因为rmchange利用共享内存部分）。

```
Init:
    irmovl $2, %edx
    irmovl $0x1f00, %eax
    rmchange %edx, (%eax)
    andl %edx, %edx
    je Sender
    jmp Receiver
Sender:
    irmovl $0x1022, %eax
    irmovl $3, %ecx
    rmmovl %ecx, 4(%eax)
    halt
Receiver:
    irmovl $0x1022, %eax
    irmovl $5, %ecx
    rmmovl %ecx, 8(%eax)
    halt
```

```
Changed Register State:                Changed Register State:
%eax:   0x00000000      0x00001022 %eax:   0x00000000      0x00001022
%ecx:   0x00000000      0x00000003 %ecx:   0x00000000      0x00000005
Changed Memory State:              %edx:   0x00000000      0x00000002
```

Figure 1: Test of rmchange

2.4.2 发送字节

这里使用另外一个程序，每次A发送给B， m 个数字，例如1,2,3,4,5，B 则返回m+1 个数字，例如6,7,8,9,10,11 相互交替。由于在y86里面无法直接把信息打印在屏幕上。因此，我将每次接收到的信息都放在了每个进程的私有空间里面。如果需要查看第二十次的结果，则将循环次数设为20.并观察模拟器最后在屏幕上打印出来的结果。

忽略0x07ec 的结果，可以看到，他们的值是连续的。

3 实验总结

在原来代码的基础上，我主要更改了isa.c中的内容，添加了对共享内存的访问部分，并添加了

为使共享内存保持同步的通讯机制，为保证内存的原子性的lock机制。

并在psim.c添加了少部分内容，保证进程离开时，清空所有的cache，保存到主存上。

增加了rmchange命令，修改了yas-grammer.lex， pipe-full.hcl 中的部分内容。

4 感想

抱歉这回作业交迟了。

由于第一次实验较为简单，错误地估计了第二次实验的难度。第十四周周末开始做，也由于自身的能力不足，整整做了一周还是没有做出来。反而耽误了许多复习的时间。

从前没有接触过类似的任务，最初面对这道题目不知所措，过程中请教了谷昱、刘汉鹏同学，从他们那里得到了一些思路和帮助。逐渐构建起了整个体系。虽然花费了非常多的精力，但是也感觉收获很大，从最开始出现的segmental fault一脸茫然，到最后能够较熟练地debug，感觉对Linux C编程的理解上升了一个台阶。

这次实验的一个难点主要在于调试的困难性上面。开始使用的fork进行主父进程区分。然而打印出来的值十分混乱。在理清逻辑上花费了很多的精力，最后还是使用了第三个进程对这两个进程进行调度。思路清晰之后，debug的速度有所加快。

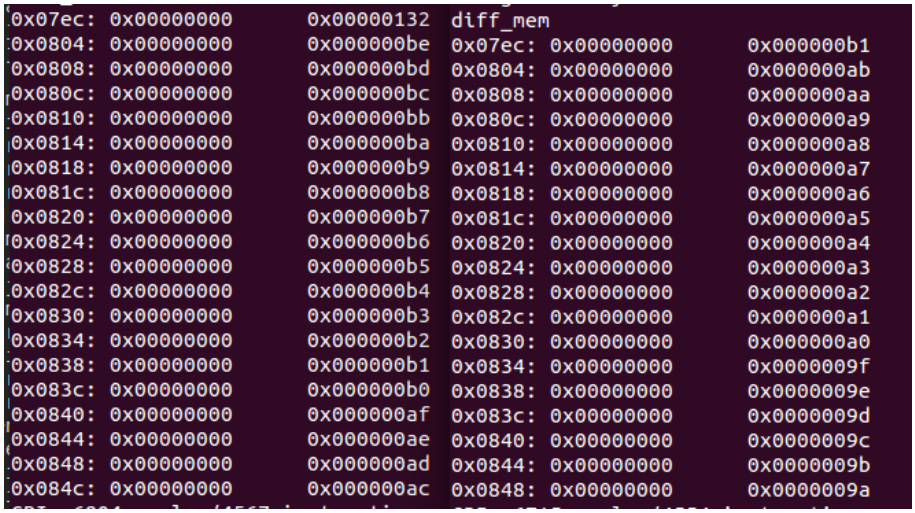

Figure 2: Test of messages send and receive