操作系统(D)

项目 8

李子龙 518070910095

2021 年 4 月 11 日

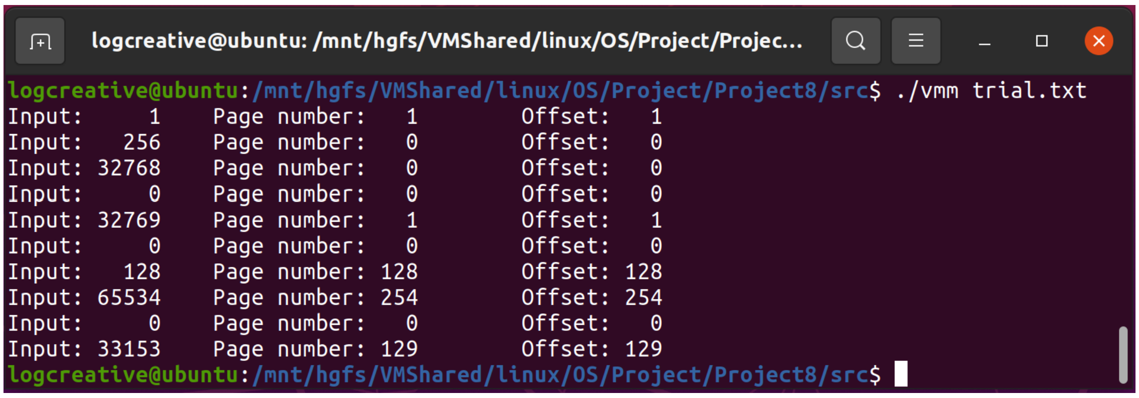
设计虚拟内存管理器

1. 起步

首先将测试用的地址写入 trial.txt，以测试 `addext` 内存地址分析模块的正确性。

Listing 1: src/trial.txt

```
1
256
32768
32769
128
65534
33153
```



logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...

```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ ./vmm trial.txt
Input:      1     Page number:   1         Offset:   1
Input:    256     Page number:   0         Offset:   0
Input: 32768      Page number:   0         Offset:   0
Input:      0     Page number:   0         Offset:   0
Input: 32769      Page number:   1         Offset:   1
Input:      0     Page number:   0         Offset:   0
Input:    128     Page number: 128         Offset: 128
Input: 65534      Page number: 254         Offset: 254
Input:      0     Page number:   0         Offset:   0
Input: 33153      Page number: 129         Offset: 129
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$
```

定义地址结构和地址提取器如下：

Listing 2: src/addext.h

```c
#include <stdlib.h>

typedef struct address
{
    int number;
    int offset;
} add;
```

```
add addext(int _rline);
int getAdd(add _addin);
```

Listing 3: src/addext.c

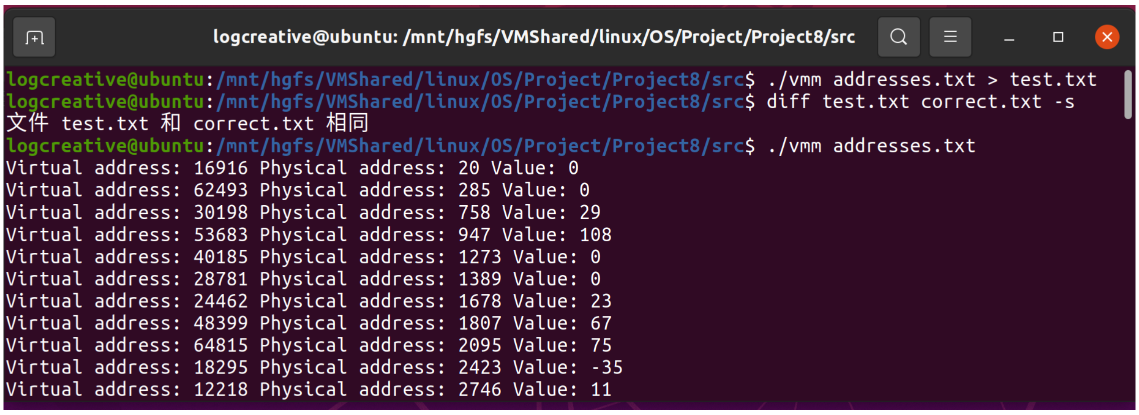
```
#include "addext.h"

add addext(int _rline) {
    add add_;
    _rline = _rline & 0x0000FFFF;
    add_.number = (_rline & 0x0000FF00) >> 8;
    add_.offset = _rline & 0x000000FF;
    return add_;
}

int getAdd(add _addin) {
    return (_addin.number << 8) + _addin.offset;
}
```

2. 处理页面错误

接着，先不考虑 TLB，只使用页表。将输出结果与正确参考比较，结论是正确：



```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ ./vmm addresses.txt > test.txt
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ diff test.txt correct.txt -s
文件 test.txt 和 correct.txt 相同
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ ./vmm addresses.txt
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
```

首先对输入流分析，在 main 函数里的情形如下：

```
    while(fgets(addline, MAXLINE, addfile)!=NULL){
        int rline = atoi(addline);
        add viradd = addext(rline);
        add phyadd = getPhyAdd(viradd);
        fprintf(stdout, "Virtual address: %d Physical address: %d Value: %d\n",
            getAdd(viradd), getAdd(phyadd), getValue(phyadd));
    }
```

获取值是直接从内存中获得对应位置的值：

```
int getValue(add _phyadd) {
    return mem[_phyadd.number][_phyadd.offset];
}
```

其中 mem 是用 char 存储的：

Listing 4: src/memory.h

```
#ifndef MEMORY
#define MEMORY 1

#include <stdio.h>

#define MEMSIZE 256
#define FRAMESIZE 256

char mem[MEMSIZE][FRAMESIZE];

int read_frame(int page_number);

#endif
```

当前只使用页表是不需要考虑 TLB 的获取物理地址的函数如下：

```
add getPhyAdd(add _inadd) {
    add phyadd;

    if (!page_table[_inadd.number][1])
        handle_pagefault(_inadd.number);

    phyadd.number = page_table[_inadd.number][0];
    phyadd.offset = _inadd.offset;
    return phyadd;
}
```

一旦有页面错误就会触发对应的函数，将内容存放到内存中去：

```
void handle_pagefault(int page_number) {
    int frame_number = read_frame(page_number);
    page_table[page_number][0] = frame_number;
    page_table[page_number][1] = 1;
}
```

由于现在的内存充足，帧码直接用静态变量 `frame_number` 递增存储。

```
int read_frame(int page_number) {
    static int frame_number = 0;

    FILE* backstore;
    if ((backstore = fopen("BACKING_STORE.bin", "rb")) == NULL) {
        fprintf(stderr, "Empty file storage!\n");
        return -1;
    }

    int frame_number_ = frame_number++;
    long pos = page_number * FRAMESIZE;
    fseek(backstore, pos, SEEK_SET);
    fread(mem[frame_number_], sizeof(char), FRAMESIZE, backstore);
    fclose(backstore);

    return frame_number_;
}
```

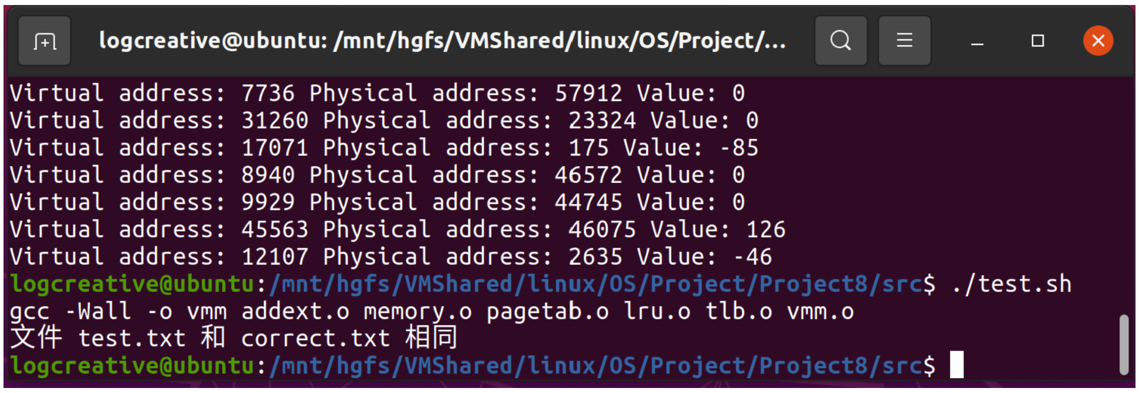
这里使用了二进制文件读取的方式复制到内存中去。

3. 使用 TLB

使用 `test.sh` 脚本进行相同的测试后，结果仍然是一致的。

Listing 5: src/test.sh

```
make
./vmm addresses.txt > test.txt
diff test.txt correct.txt -s
```


logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/...

```
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ ./test.sh
gcc -Wall -o vmm addext.o memory.o pagetab.o lru.o tlb.o vmm.o
文件 test.txt 和 correct.txt 相同
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project8/src$ 
```

首先，输入流分析被重定向到 TLB 对应的接口。

```
add getPhyAdd(add _inadd) {
    add phyadd;

    phyadd.number = tlb_search(_inadd.number);
    phyadd.offset = _inadd.offset;

    return phyadd;
}
```

TLB 由 16 个内存块组成。

Listing 6: src/tlb.h

```
#ifndef TLB_GUARD
#define TLB_GUARD 1

#include "pagetab.h"
#include "lru.h"

#define TLBSIZE 16

// TLB[i][2] - isOccupied:
//      0 - empty
//      1 - occupied
int TLB[TLBSIZE][3];

int tlb_search(int page_number);

#endif // !TLB_GUARD
```

而最主要的 tlb_search 函数首先会尝试 TLB 命中，接着如果是 TLB 未命中，就看需不需要触发页面缺失，然后看是否由空余 TLB 空间用于存储到 TLB 中。如果 TLB 满，就会使用 LRU 算法进行 TLB 置换。

Listing 7: src/tlb.c

```c
#include "tlb.h"

int tlb_search(int page_number) {
    static struct used_node* tlb_head = NULL;
    static struct used_node* tlb_tail = NULL;

    int result = -1;
    for (int i = 0; i < TLBSIZE; ++i) {
        if (TLB[i][2]          // is occupied
            && TLB[i][0] == page_number) {
            result = i;
            break;
        }
    }
    if (result >= 0) {         // TLB hit
        search_pop(&tlb_head, &tlb_tail, page_number);
        push(&tlb_head, &tlb_tail, page_number);
        return TLB[result][1];
    }
    else {                     // TLB miss

        if (!page_table[page_number][1]) // page fault
            handle_pagefault(page_number);

        int frame_number = page_table[page_number][0];

        int hole = -1;

        for (int i = 0; i < TLBSIZE; ++i)
            if (!TLB[i][2]) { // is empty
                hole = i;
                break;
            }

        if (hole >= 0) {
            TLB[hole][0] = page_number;
            TLB[hole][1] = frame_number;
            TLB[hole][2] = 1;
            push(&tlb_head, &tlb_tail, page_number);
        } else {               // full TLB
            // LRU Algorithm
            int least_used = bottom_pop(&tlb_head, &tlb_tail);
            int least_used_index = 0;
            for (; TLB[least_used_index][0] != least_used; ++least_used_index);
            TLB[least_used_index][0] = page_number;
            TLB[least_used_index][1] = frame_number;
            push(&tlb_head, &tlb_tail, page_number);
        }

        return frame_number;
    }
}
```

对于 TLB 使用状态使用了双向链表式栈存储，记录头 `tlb_head` 和尾 `tlb_tail`。对应的 LRU 操作具有如下定义：

Listing 8: src/lru.h

```c
#ifndef LRU
#define LRU 1

#include <stdlib.h>

struct used_node {
    int page_number;
    struct used_node* prev;
    struct used_node* next;
};

void search_pop(struct used_node** head, struct used_node** tail, int page_number);

void push(struct used_node** head, struct used_node** tail, int page_number);

int bottom_pop(struct used_node** head, struct used_node** tail);

#endif
```

- `search_pop` 将会寻找对应的栈节点，并移除。
- `push` 入栈操作。
- `bottom_pop` 栈底出栈。

Listing 9: src/lru.c

```c
#include "lru.h"

void search_pop(struct used_node** head, struct used_node** tail, int page_number) {
    struct used_node* tmp = *head;
    while (tmp && tmp->page_number != page_number)
        tmp = tmp->next;
    if (!tmp) return;
    if (*head == *tail) {
        *head = *tail = NULL;
        return;
    }
    if (tmp == *head) {
        tmp->next->prev = tmp->prev;
        *head = tmp->next;
    }
    else if (tmp == *tail) {
        tmp->prev->next = tmp->next;
        *tail = tmp->prev;
    }
    else {
        tmp->next->prev = tmp->prev;
        tmp->prev->next = tmp->next;
    }
    free(tmp);
}

void push(struct used_node** head, struct used_node** tail, int page_number) {
    struct used_node* new_node = (struct used_node*)malloc(sizeof(struct used_node));
    new_node->page_number = page_number;
    new_node->prev = NULL;
    new_node->next = NULL;

    if (!*head) {
        *head = *tail = new_node;
        return;
    }

    new_node->next = *head;
    (*head)->prev = new_node;
    *head = new_node;
}

int bottom_pop(struct used_node** head, struct used_node** tail) {
    if (!*tail) return -1;

    int bottom = (*tail)->page_number;
    if (*head == *tail) {
        *head = *tail = NULL;
        return bottom;
    }

    *tail = (*tail)->prev;
    free((*tail)->next);
    (*tail)->next = NULL;
    return bottom;
}
```