Project 8: Designing Virtual Memory Manager

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

1 Designing Virtual Memory Manager

1.1 Requirements

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65536$ bytes. The program will read from a file containing logical address and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. It requires us to use simulation to understand the steps involved in translating logical address to physical address, and this will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

The program will read a file containing several 32-bit integer numbers that represent logical addresses. However, the 16-bit addresses is the only thing that needs to be concerned, so we must mask the rightmost 16 bits of each logical addresses. These 16 bits are divided into an 8-bit page number and an 8-bit page offset. Hence, the addresses are structured as shown as Fig. 1.

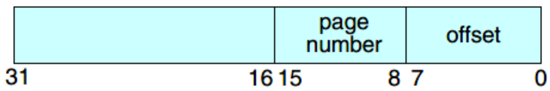

Figure 1: The address structure

Other specifics includes the following:

- 2^8 entries in the page table;
- Page size of 2^8 bytes;
- 16 entries in the TLB;
- Frame size of 2^8 bytes;
- 256 frames;
- Physical memory of 65536 bytes (256 frames $\times$ 256-byte frame size).

The program is to output the following values:

- The logical address being translated (the integer value being read from `addresses.txt`).
- The corresponding physical address (what your program translates the logical address to).
- The signed byte value stored in physical memory at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if the program is correctly translating logical to physical addresses.

After completion, the program is to report the following statistics:

- Page-fault rate - The percentage of address references that resulted in page faults.
- TLB hit rate - The percentage of address references that were resolved in the TLB.

Since the logical addresses in `addresses.txt` were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

After that, we need to change the frame number to 128 to implement the page replacement algorithm.

1.2 Methods

Here are some specific methods of the virtual memory manager.

- We implement a free frame list using the linked list, and at first all the frames are free. Every time we need a free frame, we can look it up in the free frame list using `get_empty_frame()` function. If the function returns -1, then there is no empty frame and the memory will need a page replacement; otherwise the return value of the function is the empty frame number.
- We implement the memory using the LRU page replacement strategy. Every time we need to add a page into the memory, we will call the `get_empty_frame()` function to get the free frame number. If there is a free frame, we will use it to store the page; otherwise we will use the LRU page replacement algorithm to find the victim page and replace it with the requested page, and we will use the `delete_page_table_item()` function to delete the corresponding page table item and TLB item.
- We implement the TLB also using the LRU replacement strategy. Every time we need to add an entry into the TLB, we will find out whether there is a free entry. If so, we can directly put it into the entry; otherwise we must replace one entry using the LRU algorithm. Every time TLB hits, we will also update the LRU value of each entry.
- We also implement the page table using the normal methods. Every time we need to transform the virtual address into the physical address, we will first look it up in TLB by calling `get_TLB_frame_num()` function provided by TLB. If TLB miss, we will refer to the page table. If page table miss, we will using the demand paging algorithm to put the page in the backing store into the memory. So the transforming process is as follows (Fig. 2).

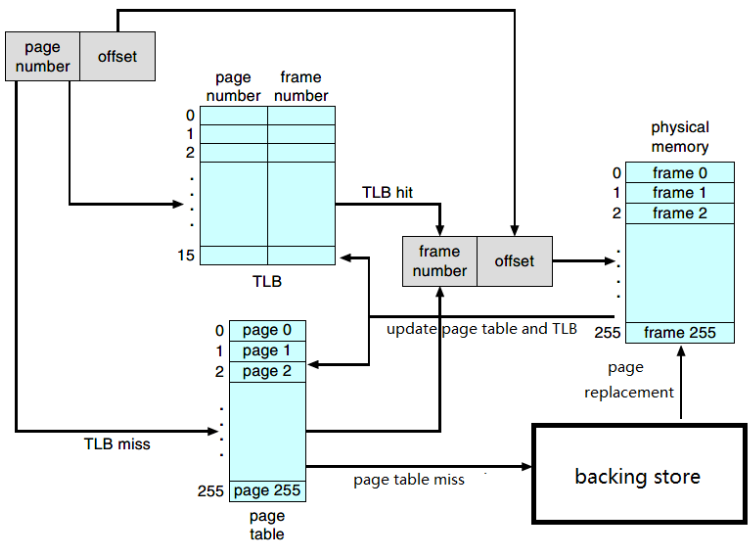

Figure 2: The transforming process

1.3 Implementation

Here is the specific implementation of the virtual memory manager (`vmm.c`). Note: we directly implement the page-replacement algorithm here.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# define PAGE_NUM 256
# define PAGE_SIZE 256
# define FRAME_NUM 128
# define FRAME_SIZE 256
# define TLB_SIZE 16

// ================= Empty Frame List ================== //
struct empty_frame_list_node {
  int frame_num;
  struct empty_frame_list_node *nxt;
};

struct empty_frame_list_node *head = NULL;
struct empty_frame_list_node *tail = NULL;

// Add the empty frame to the empty frame list.
void add_empty_frame(int frame_num) {
  if (head == NULL && tail == NULL) {
    tail = (struct empty_frame_list_node *) malloc (sizeof(struct empty_frame_list_node));
    tail -> frame_num = frame_num;
    tail -> nxt = NULL;
    head = tail;
  } else {
    tail -> nxt = (struct empty_frame_list_node *) malloc (sizeof(struct empty_frame_list_node));
    tail -> nxt -> frame_num = frame_num;
    tail -> nxt -> nxt = NULL;
    tail = tail -> nxt;
  }
}

// Get an empty frame from the empty frame list.
// If success, return frame_num; otherwise, return -1.
int get_empty_frame() {
  if (head == NULL && tail == NULL) return -1;

  int frame_num;
  if (head == tail) {
    frame_num = head -> frame_num;
    free(head);
    head = tail = NULL;
```

```
    return frame_num;
  }

  struct empty_frame_list_node *tmp;
  frame_num = head -> frame_num;
  tmp = head;
  head = head -> nxt;
  free(tmp);
  return frame_num;
}

// Initialize the empty frame list.
void initialize_empty_frame_list() {
  for (int i = 0; i < FRAME_NUM; ++ i)
    add_empty_frame(i);
}

// Clean the empty frame list.
void clean_empty_frame_list() {
  if (head == NULL && tail == NULL) return;
  struct empty_frame_list_node *tmp;
  while (head != tail) {
    tmp = head;
    head = head -> nxt;
    free(tmp);
  }
  free(head);
  head = tail = NULL;
}
// ============== End of Empty Frame List ============== //


// ===================== Memory ====================== //
char memory[FRAME_NUM * FRAME_SIZE];
int frame_LRU[FRAME_NUM];
char buf[FRAME_SIZE];
FILE *fp_backing_store;

void initialize_memory() {
  fp_backing_store = fopen("BACKING_STORE.bin", "rb");
  if (fp_backing_store == NULL) {
    fprintf(stderr, "[Err] Open backing store file error!\n");
    exit(1);
  }
  initialize_empty_frame_list();
  for (int i = 0; i < FRAME_NUM; ++ i)
    frame_LRU[i] = 0;
}

void delete_page_table_item(int frame_num);
```

```c
int add_page_into_memory(int page_num) {
  fseek(fp_backing_store, page_num * FRAME_SIZE, SEEK_SET);
  fread(buf, sizeof(char), FRAME_SIZE, fp_backing_store);

  int frame_num = get_empty_frame();
  if (frame_num == -1) {
    // LRU replacement.
    for (int i = 0; i < FRAME_NUM; ++ i)
      if (frame_LRU[i] == FRAME_NUM) {
        frame_num = i;
        break;
      }
    delete_page_table_item(frame_num);
  }

  for (int i = 0; i < FRAME_SIZE; ++ i)
    memory[frame_num * FRAME_SIZE + i] = buf[i];
  for (int i = 0; i < FRAME_NUM; ++ i)
    if (frame_LRU[i] > 0) ++ frame_LRU[i];
  frame_LRU[frame_num] = 1;
  return frame_num;
}

char access_memory(int frame_num, int offset) {
  char res = memory[frame_num * FRAME_SIZE + offset];
  for (int i = 0; i < FRAME_NUM; ++ i)
    if (frame_LRU[i] > 0 && frame_LRU[i] < frame_LRU[frame_num])
      ++ frame_LRU[i];
  frame_LRU[frame_num] = 1;
  return res;
}

void clean_memory() {
  clean_empty_frame_list();
  fclose(fp_backing_store);
}
// ================== End of Memory =================== //


// ======================= TLB ======================= //
int TLB_page[TLB_SIZE], TLB_frame[TLB_SIZE];
int TLB_LRU[TLB_SIZE];
int TLB_hit_count;

void initialize_TLB() {
  TLB_hit_count = 0;
  for (int i = 0; i < TLB_SIZE; ++ i) {
    TLB_page[i] = 0;
    TLB_frame[i] = 0;
    TLB_LRU[i] = 0;
```

```
  }
}

// Get the corresponding frame number from TLB.
//   Return non-negative number for the corresponding frame number;
//   Return -1 for TLB miss.
//   Note: it's needless to check the validation of page_num again.
int get_TLB_frame_num(int page_num) {
  int pos = -1;
  for (int i = 0; i < TLB_SIZE; ++ i)
    if (TLB_LRU[i] > 0 && TLB_page[i] == page_num) {
      pos = i;
      break;
    }

  if (pos == -1) return -1;

  // TLB hit.
  ++ TLB_hit_count;
  for (int i = 0; i < TLB_SIZE; ++ i)
    if (TLB_LRU[i] > 0 && TLB_LRU[i] < TLB_LRU[pos])
      ++ TLB_LRU[i];
  TLB_LRU[pos] = 1;
  return TLB_frame[pos];
}

// Update TLB entry
void update_TLB(int page_num, int frame_num) {
  int pos = -1;
  for (int i = 0; i < TLB_SIZE; ++ i)
    if(TLB_LRU[i] == 0) {
      pos = i;
      break;
    }

  if (pos == -1) {
    // LRU replacement.
    for (int i = 0; i < TLB_SIZE; ++ i)
      if(TLB_LRU[i] == TLB_SIZE) {
        pos = i;
        break;
      }
  }

  TLB_page[pos] = page_num;
  TLB_frame[pos] = frame_num;
  for (int i = 0; i < TLB_SIZE; ++ i)
    if (TLB_LRU[i] > 0) ++ TLB_LRU[i];
  TLB_LRU[pos] = 1;
}
```

```

// Delete TLB item.
void delete_TLB_item(int page_num, int frame_num) {
  int pos = -1;
  for (int i = 0; i < TLB_SIZE; ++ i)
    if(TLB_LRU[i] && TLB_page[i] == page_num && TLB_frame[i] == frame_num) {
      pos = i;
      break;
    }

  if (pos == -1) return;

  for (int i = 0; i < TLB_SIZE; ++ i)
    if (TLB_LRU[i] > TLB_LRU[pos]) -- TLB_LRU[i];
  TLB_LRU[pos] = 0;
}
// ==================== End of TLB =================== //


// =================== Page Table ==================== //
int page_table[PAGE_NUM];
int vi_page_table[PAGE_NUM]; // vi: valid-invalid
int page_fault_count;

void initialize_page_table() {
  page_fault_count = 0;
  for (int i = 0; i < PAGE_NUM; ++ i) {
    page_table[i] = 0;
    vi_page_table[i] = 0;
  }
}

// Get the corresponding frame number.
//   Return non-negative number for the corresponding frame number;
//   Return -1 for invalid page number.
int get_frame_num(int page_num) {
  if (page_num < 0 || page_num >= PAGE_NUM) return -1;

  int TLB_res = get_TLB_frame_num(page_num);
  if (TLB_res != -1) return TLB_res;

  if (vi_page_table[page_num] == 1) {
    update_TLB(page_num, page_table[page_num]);
    return page_table[page_num];
  } else {
    // Page fault.
    ++ page_fault_count;
    page_table[page_num] = add_page_into_memory(page_num);
    vi_page_table[page_num] = 1;
    update_TLB(page_num, page_table[page_num]);
```

```
    return page_table[page_num];
  }
}

// Delete page table item
void delete_page_table_item(int frame_num) {
  int page_num = -1;
  for (int i = 0; i < PAGE_NUM; ++ i)
    if(vi_page_table[i] && page_table[i] == frame_num) {
      page_num = i;
      break;
    }
  if (page_num == -1) {
    fprintf(stderr, "[Err] Unexpected Error!\n");
    exit(1);
  }
  vi_page_table[page_num] = 0;
  delete_TLB_item(page_num, frame_num);
}
// ================= End of Page Table ================= //


void initialize() {
  initialize_page_table();
  initialize_TLB();
  initialize_memory();
}

void clean() {
  clean_memory();
}


int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "[Err] Invalid input!\n");
    return 1;
  }

  FILE *fp_in = fopen(argv[1], "r");
  if(fp_in == NULL) {
    fprintf(stderr, "[Err] File Error!\n");
    return 1;
  }

  FILE *fp_out = fopen("output.txt", "w");
  if (fp_out == NULL) {
    fprintf(stderr, "[Err] File Error!\n");
    return 1;
  }
```

```
295
296   initialize();
297
298   int addr, page_num, offset, frame_num, res, cnt = 0;
299   while(~fscanf(fp_in, "%d", &addr)) {
300     ++ cnt;
301     addr = addr & 0x0000ffff;
302     offset = addr & 0x000000ff;
303     page_num = (addr >> 8) & 0x000000ff;
304     frame_num = get_frame_num(page_num);
305     res = (int) access_memory(frame_num, offset);
306     fprintf(fp_out, "Virtual address: %d Physical address: %d Value: %d\n", addr,
          (frame_num << 8) + offset, res);
307   }
308
309   fprintf(stdout, "[Statistics]\n TLB hit rate: %.4f %%\n Page fault rate: %.4f
        %%\n", 100.0 * TLB_hit_count / cnt, 100.0 * page_fault_count / cnt);
310
311   clean();
312   fclose(fp_in);
313   fclose(fp_out);
314   return 0;
315 }
```

1.4 Testing

I write a `Makefile` file and a `checker.c` program to help testing the program. You can modify Line 7 in `vmm.c` to change the total frame number to test the page replacement algorithm, and the default number is 128. Since the physical address may be different because of the different replacement strategy, we only check the value stored in the virtual address. The `checker.c` will output `Accept` if the program output matches the correct output, and the checker will output `Wrong Answer` if the outputs do not match. Here is the specific implementation of the checker (`checker.c`).

```
# include <stdio.h>
# include <string.h>

int main() {
  FILE *fp_out = fopen("output.txt", "r");
  FILE *fp_ans = fopen("correct.txt", "r");

  int accept = 1;
  int right_val, out_val, cnt = 0;
  while (~fscanf(fp_ans, "Virtual address: %*d Physical address: %*d Value: %d\n",
       &right_val)) {
    ++ cnt;
    if (fscanf(fp_out, "Virtual address: %*d Physical address: %*d Value: %d\n", &
         out_val) == EOF) {
      printf("File length dismatch!\n");
      accept = 0;
      break;
```

```
    }
    if (right_val != out_val) {
      printf("Error on line %d.\n", ++ cnt);
      accept = 0;
    }
  }

  if (accept == 0) printf("Wrong answer.\n");
  else printf("Accept.\n");

  fclose(fp_out);
  fclose(fp_ans);
  return 0;
}
```

We only need to enter the following instructions in the terminal and we can begin testing.

```
make
./vmm addresses.txt
./checker
```

When the memory frame number is 128, the execution result of the virtual memory manager is as follows (Fig. 3).

```
galaxies@ubuntu:~/CS307-Projects/Project8$ make
gcc -Wall -c vmm.c
gcc -Wall -c checker.c
gcc -Wall -o vmm vmm.o
gcc -Wall -o checker checker.o
galaxies@ubuntu:~/CS307-Projects/Project8$ ./vmm addresses.txt
[Statistics]
  TLB hit rate: 5.5000 %
  Page fault rate: 53.9000 %
galaxies@ubuntu:~/CS307-Projects/Project8$ ./checker
Accept.
```

Figure 3: The execution result of the virtual memory manager when memory frame number is 128

When the memory frame number is 256, the execution result of the virtual memory manager is as follows (Fig. 4).

```
galaxies@ubuntu:~/CS307-Projects/Project8$ make
gcc -Wall -c vmm.c
gcc -Wall -c checker.c
gcc -Wall -o vmm vmm.o
gcc -Wall -o checker checker.o
galaxies@ubuntu:~/CS307-Projects/Project8$ ./vmm addresses.txt
[Statistics]
  TLB hit rate: 5.5000 %
  Page fault rate: 24.4000 %
galaxies@ubuntu:~/CS307-Projects/Project8$ ./checker
Accept.
```

Figure 4: The execution result of the virtual memory manager when memory frame number is 256

When the memory frame number decreases, the page fault rate increases, which is correct since the memory can contain fewer frames.

2 Personal Thoughts

The project helps me understand the virtual memory manager better, including the mapping between virtual memory and physical memory, the replacement algorithm in TLB and memory and the backing store. The project also improves my understanding of page table and TLB - which is the cache of the page table. The project trains our coding skills of designing appropriate cache. The implementation of the algorithms is quite simple but it needs patience, and I enjoy the process of writing this program.

By the way, you can find all the source codes in the "src" folder. You can also refer to my github to see my codes of this project, and they are in the `Project8` folder.