< Operating Systems Project />

} /> [

Aytan Novruzlu
Bahruz Gurbanli
Chilanay Hajisoy
Ilhama Novruzova
Natavan Hasanova

</ Table of contents

</>

Introduction

01

} /> [

</>

Overview

02

} /> [

</ Overview

Key Components

- Virtual Memory: Manages system memory to provide an efficient and isolated virtual address space for processes.
- Page Table: Maps virtual addresses to physical addresses.
- Physical Memory: Simulated RAM where data is stored temporarily.
- Translation Lookaside Buffer (TLB): Cache used to reduce time needed to access memory locations.
- Disk: simulates a hard disk drive, serving as the backing store for pages not currently held in physical memory, supporting random-access retrieval during page faults.

</>

03

Implementation

</>

} /> [

</ Implementation of Page Table, Physical Memory, and TLB

- TLB Entry: struct representing page number, frame number, validity of entry, and counter for LRU replacement.
- TLB: Array of TLB Entries
- Page Table Entry: struct representing frame numbers, validity of entry, and whether it has been referenced or not.
- Page Table: Array of page table entries representing frame numbers.
- Physical Memory: simulated 2D array of number of frames and frame size

```
typedef struct {
    unsigned int page_number;
    unsigned int frame_number;
    bool valid;
    int counter; // Add counter field for LRU replacement
} TLBEntry;

typedef struct {
    unsigned int frame_number;
    bool valid;
    bool referenced;  // adding R bit for tracking usage of pages
} PageTableEntry;

extern unsigned char physical_memory[NUM_FRAMES][FRAME_SIZE];
```

</ TLB Miss (FIFO & Second Chance Replacement Strategy)

- **On a TLB Miss:**
  - In the main program and Extra 1, the system uses a simple circular buffer method to manage the TLB entries. This approach does not specifically assess the usage or age of each entry. When a TLB miss occurs and a new entry needs to be added, the system replaces the entry at the current tlb_next_index, which automatically moves to the next position after each insertion. This ensures all entries are used and replaced in a sequential, rotating manner.

```c
frame_number = search_TLB(tlb, page_number, &total_tlb_hits);

if (frame_number == -1) { // TLB Miss
    if (page_table[page_number].valid) {
        frame_number = page_table[page_number].frame_number;
        update_frame_queue(frame_number);
        page_table[page_number].referenced = true;
    } else {
        frame_number = handle_page_fault(page_table, tlb, page_number, frame_occupied, &total_page_faults);
    }
}
```

```c
int tlb_next_index = 0;
void update_TLB(TLBEntry *tlb, unsigned int page_number, unsigned int frame_number) {
    for (int i = 0; i < TLB_SIZE; i++) {
        if (tlb[i].valid && tlb[i].page_number == page_number) {
            tlb[i].frame_number = frame_number;
            return;
        }
    }

    tlb[tlb_next_index].page_number = page_number;
    tlb[tlb_next_index].frame_number = frame_number;
    tlb[tlb_next_index].valid = true;
    tlb_next_index = (tlb_next_index + 1) % TLB_SIZE;
}

int search_TLB(const TLBEntry *tlb, unsigned int page_number, int *total_tlb_hits) {
    for (int i = 0; i < TLB_SIZE; i++) {
        if (tlb[i].valid && tlb[i].page_number == page_number) {
            (*total_tlb_hits)++;
            return tlb[i].frame_number;
        }
    }

    return -1;
}
```

</ TLB Miss (Least Recently Used Replacement Strategy)

- **On a TLB Miss:**
  - In Extra 2, the system uses the Least Recently Used (LRU) strategy for TLB management. This strategy involves tracking how long each TLB entry has not been used. When a TLB miss occurs, the system searches for the least recently used entry based on a usage counter array (tlb_usage_counter). The entry with the highest count (indicating it was used least recently) is replaced with the new page information. This method is effective for maintaining the most frequently accessed pages in the TLB.

```c
int search_TLB(const TLBEntry *tlb, unsigned int page_number, int *total_tlb_hits) {
    // Find the least recently used entry
    int least_used_index = 0;
    int least_used_count = tlb_usage_counter[0];
    for (int i = 1; i < TLB_SIZE; i++) {
        if (tlb_usage_counter[i] < least_used_count) {
            least_used_index = i;
            least_used_count = tlb_usage_counter[i];
        }
    }

    // Search for the page in the TLB
    for (int i = 0; i < TLB_SIZE; i++) {
        if (tlb[i].valid && tlb[i].page_number == page_number) {
            (*total_tlb_hits)++;
            // Update usage counters for all entries
            for (int j = 0; j < TLB_SIZE; j++) {
                if (j != i && tlb[j].valid) {
                    tlb_usage_counter[j]++;
                }
            }
            // Reset usage counter for the hit entry
            tlb_usage_counter[i] = 0;
            return tlb[i].frame_number;
        }
    }

    // If the page is not found in the TLB, update least recently used entry
    tlb_usage_counter[least_used_index]++;
    return -1;
}
```

```c
void update_TLB(TLBEntry *tlb, unsigned int page_number, unsigned int frame_number) {
    // Update TLB with new entry
    tlb[tlb_next_index].page_number = page_number;
    tlb[tlb_next_index].frame_number = frame_number;
    tlb[tlb_next_index].valid = true;

    // Increment usage counter for the updated entry
    tlb_usage_counter[tlb_next_index] = 0;

    // Increment usage counters for all other entries
    for (int i = 0; i < TLB_SIZE; i++) {
        if (i != tlb_next_index && tlb[i].valid) {
            tlb_usage_counter[i]++;
        }
    }

    // Move to the next index for next update
    tlb_next_index = (tlb_next_index + 1) % TLB_SIZE;
}
```

</ Page Fault Handling (FIFO & LRU)

```c
int handle_page_fault(PageTableEntry *page_table, TLBEntry *tlb, unsigned int page_number, bool *frame_occupied, int *total_page_faults) {
    (*total_page_faults)++;
    int frame_number = read_page_from_disk(page_number, physical_memory, frame_occupied);
    if (frame_number != -1) {
        update_page_table(page_table, page_number, frame_number);
        update_TLB(tlb, page_number, frame_number);
    }
    return frame_number;
}
```

</ Page Fault Handling (Second Chance Replacement Strategy)

```c
int handle_page_fault_second_chance(PageTableEntry *page_table, TLBEntry *tlb, unsigned int page_number, bool *frame_occupied, int *total_page_faults) {
    (*total_page_faults)++;
    int frame_number = -1;

    while (true) {
        int idx = frame_queue[front];
        if (!page_table[idx].referenced) {
            // If R bit is 0, replace this page.
            frame_number = idx;
            break;
        } else {
            // If R bit is 1, give it a second chance.
            page_table[idx].referenced = false;
            // Move this page to the end of the queue.
            update_frame_queue(idx);
        }
    }

    // After finding, we make a place for that frame
    frame_occupied[frame_number] = true;

    // Now read the frame into the selected page
    frame_number = read_page_from_disk(page_number, physical_memory, frame_occupied);
    if (frame_number != -1) {
        update_page_table(page_table, page_number, frame_number);
        update_TLB(tlb, page_number, frame_number);
    }
    return frame_number;
}
```

</ Configurability of System Parameters

- Parameters like TLB size and page size are configurable through constant definitions in the source code.
- Changes can be made easily by altering these definitions, demonstrating flexibility in the system’s design.

```
#define TLB_SIZE 16
#define PAGE_SIZE 256
#define FRAME_SIZE 256
#define NUM_FRAMES 256
#define MAX_PAGES 256
```

</ Experiments and Techniques

- **Performance analysis** of different page replacement algorithms (like FIFO, LRU, or Second Chance).
- **Testing the effectiveness** of the TLB and its hit rate under various workload scenarios.

</ Experiment Results

Main Program (64 KB Memory)

```
Page Fault Rate: 8.533%
TLB Hit Rate: 6.200%
```

Least Recently Used Program (64 KB Memory)

```
Page Fault Rate: 8.533%
TLB Hit Rate: 14.733%
```

Second-Chance Replacement Program (16 KB Memory)

```
Page Fault Rate: 75.267%
TLB Hit Rate: 7.333%
```

Second-Chance Replacement Program (64 KB Memory)

```
Page Fault Rate: 8.700%
TLB Hit Rate: 6.433%
```

Least Recently Used Program (16 KB Memory)

```
Page Fault Rate: 75.533%
TLB Hit Rate: 82.867%
```

</ Page Loading from the Backing Store

- In our simulated virtual memory manager, pages are loaded from the backing store (a simulated disk) only when they are needed, which is determined during a page fault.

```c
int read_page_from_disk(unsigned int page_number, unsigned char physical_memory[NUM_FRAMES][FRAME_SIZE], bool *frame_occupied) {
    const char *disk_filename = "disk_sim";
    FILE *disk = fopen(disk_filename, "rb");
    if (disk == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }

    unsigned char BUFFER[PAGE_SIZE];
    int frame_number = -1;

    long offset = (long) page_number * PAGE_SIZE;
    fseek(disk, offset, SEEK_SET);

    if(fread(BUFFER, 1, PAGE_SIZE, disk) == PAGE_SIZE) {
        for (int i = 0; i < NUM_FRAMES; i++) {
            if (!frame_occupied[i]) {
                memcpy(physical_memory[i], BUFFER, PAGE_SIZE);
                frame_occupied[i] = true;
                frame_number = i;
                update_frame_queue(frame_number);
                break;
            }
        }
    }

    fclose(disk);
    return frame_number;
}
```

</ Physical Address Space Smaller than Virtual Address Space

- For the extra task, a smaller physical address space than the virtual address space was implemented, which is typical in virtual memory systems to use physical memory efficiently.
- Context in Project:
- Virtual address space: 64KB (calculated based on 16-bit address space: 216216 addresses)
- Physical memory configured: 16KB (for implementation of the Second Chance algorithm and to demonstrate handling with less physical memory than virtual)

</ Code Style and Practices

- Our code is divided into well-organized files, each handling specific parts of the memory management process such as TLB management, page fault handling, and initial setups.
- Functions are kept short and focused, each performing a single task, making the code easier to read and maintain.
- Consistency is maintained in naming styles across all files, aiding in predictability and understandability.
- The code is not overloaded with comments, but sufficient documentation is provided where necessary to understand the workings of complex logic.
- Systematic error checks are in place, particularly in file operations and memory access, to ensure the system behaves reliably under different scenarios.

/>

< Thanks! />

Link to GitHub

} /> [