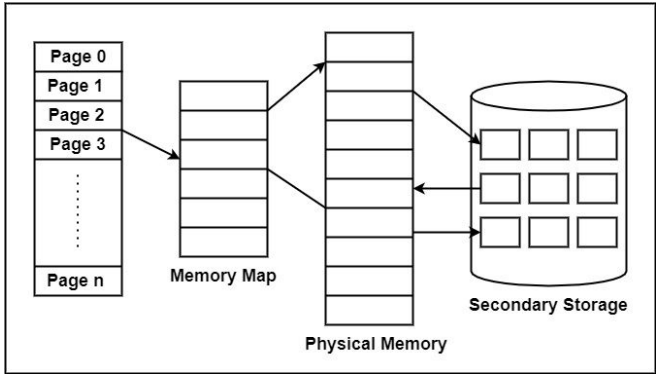
Virtual Memory Manager

Alex Moses

What is virtual memory?

- An abstraction of how much physical memory a machine has
- Allows for easier sharing of resources between processes
- Techniques such as paging and segmentation allow processes to use more “memory” than actually available



Why is virtual memory important?

- Allows for dynamic allocation of memory resources to processes
- Hides segmentation of physical memory
- Separation of process memory from operating system memory

Why did I choose to simulate virtual memory?

- I understood the lecture material fairly well, but not 100%
- I figured implementing it myself would allow me to understand the inner workings a little better
- Also a fun challenge for my C++ skills

Why C++ over C

- Object orientation - Works well for a memory manager with different instructions/methods
- Try-Catch blocks - Easy error handling
- Extensive standard library - I ❤️ Vectors
- I just kind of like it more

MemoryManager class: Page Table Entry

- Struct for easy compatibility with vectors
- Boolean variables for entry status
  - `validBit`: Is this page in use?
  - `presentBit`: Is this page in physical memory?
  - `referenceBit`: Has this page been used recently?
  - `modifyBit`: Does this page contain data?
- Mapping for physical frame number
  - `-1` if not present/initialized

```
struct pageTableEntry {
    bool validBit = false;
    bool presentBit = false;
    bool referenceBit = false;
    bool modifyBit = false;
    int pageFrameNum = -1;
};
```

MemoryManager class members

- Vectors for the page table, a mapping for physical memory, simulated disk storage, and a bitmap for which frames are free
- Parameters for how the memory is setup
  - Helpful for later calculations
- A secret for later...
  - (for our page replacement algorithm)

```
class MemoryManager {
private:
    std::vector<pageTableEntry> pageTable;
    std::vector<uint8_t> physicalMemory;
    std::vector<bool> freeFrames;

    std::vector<uint8_t> diskStorage; // simulated disk storage

    int PAGE_SIZE;
    int PAGE_COUNT; // entries in the page table
    int PHYSICAL_SIZE; // # of bytes of physical memory

    int clockPointer = 0; // for CLOCK page replacement
```

MemoryManager class: Public methods

- Constructor that allows customization of MemoryManager parameters
- Simple page allocation, returns address
- Methods for reading and writing to a virtual address
- A way to delete PTEs
- A debugging method that prints the current status of a given page

```
class MemoryManager {
public:
    MemoryManager();
    MemoryManager(int page_size, int num_pages, int num_frames);

    // allocate a page in the table.returns virtual memory address
    int allocateAnyPage();

    // write to a virtual memory address
    void writeVirtualMemory(int virtualAddress, uint8_t data);
    // read from a virtual memory address. returns data (uint8_t)
    uint8_t readVirtualMemory(int virtualAddress);

    // delete a page table entry and free its memory/disk usage
    void deletePageTableEntry(int virtualAddress);

    // print stats for a page table entry at an address to std::cout
    void printPageTableEntry(int virtualAddress);
```

MemoryManager class: Translation

1. We calculate the address offset (where within the page) and find the VPN
2. Then, we check if the entry being accessed is valid. Segmentation fault if not
3. Next, we check if the page is present. If not, we handle the page fault
4. Finally, we calculate and return the physical address and note that the page has been referenced (and modified, if applicable)

```
class MemoryManager {
private:
    _virtualToPhysicalAddress(int virtualAddress, bool writeOperation) {
        int offset = virtualAddress & (PAGE_SIZE - 1);
        int virtualPageNumber = virtualAddress / PAGE_SIZE;

        if (virtualPageNumber ≥ PAGE_COUNT || virtualPageNumber < 0)
            throw std::out_of_range("Attempted to access out-of-bound virtual address");

        pageTableEntry& entry = pageTable[virtualPageNumber];

        if (!entry.validBit)
            throw std::runtime_error("Segmentation fault occurred: Invalid page accessed");

        if (!entry.presentBit) _handlePageFault(virtualPageNumber);

        int pageFrameNum = entry.pageFrameNum;
        int physicalAddress = (pageFrameNum * PAGE_SIZE) + offset;

        if (physicalAddress ≥ PHYSICAL_SIZE || physicalAddress < 0)
            throw std::out_of_range("Physical address out of bounds");

        entry.referenceBit = true;
        if (writeOperation) entry.modifyBit = true;

        return physicalAddress;
    }
```

MemoryManager class: Page fault

1. First, we look for a free physical memory frame
2. If we don’t find one, we use page replacement to store a used page to the disk and free up a frame
3. Finally, we read the page we’re trying to access into physical memory and mark it present.

```
void MemoryManager:: _handlePageFault(int virtualPageNumber){
    std::cout << "Page fault at VPN: " <<virtualPageNumber << std::endl;

    pageTableEntry& entry = pageTable[virtualPageNumber];

    int freeFrame = -1;
    for (int i = 0; i < freeFrames.size(); i++) {
        if (freeFrames[i]) {
            freeFrame = i;
            break;
        }
    }
    if (freeFrame == -1) {
        entry.pageFrameNum = _replacePage();
    } else
        entry.pageFrameNum = freeFrame;

    _readPageFromDisk(virtualPageNumber, entry.pageFrameNum);
    freeFrames[entry.pageFrameNum] = false;
    entry.presentBit = true;
}
```

MemoryManager class: Page replacement

Using the CLOCK algorithm

1. Check if the page at the clockPointer has been referenced
    a. If it has, we set the `referenceBit` to false and continue through the page table
2. Increment pointer until page with false `referenceBit` is found
3. Write false `referenceBit` page to disk and mark it not present.
4. Return freed `frameNumber`

This allows pages used frequently between page replacements to stay in memory (gives them a second chance)

```
int MemoryManager:: _replacePage() {
    std::cout << "No free frame found, replacing page" << std::endl;

    int frameNumber = -1;
    int replacedVPN = -1;
    bool entryWasModified = false;

    bool ticking = true;
    while (ticking) {
        if (clockPointer == PAGE_COUNT) clockPointer = 0;

        pageTableEntry& entry = pageTable[clockPointer];

        if (entry.presentBit) {
            if (entry.referenceBit) {
                entry.referenceBit = false;
            } else {
                entry.presentBit = false;
                frameNumber = entry.pageFrameNum;
                replacedVPN = clockPointer;
                entryWasModified = entry.modifyBit;
                ticking = false;
            }
        }

        clockPointer++;
    }

    if (entryWasModified) _writePageToDisk(replacedVPN);

    freeFrames[frameNumber] = true;
    pageTable[replacedVPN].pageFrameNum = -1;
    _wipeMemoryFrame(frameNumber);

    return frameNumber;
}
```

MemoryManager class: other internal methods

- Initialization method for setting size of vectors
- Methods for reading/writing to physical memory
- Internal page allocation method
- Method to zero out physical frame
- Disk operations for reading, writing, and zeroing page contents on the disk

```
// initalizes vectors
void _initializeMemory();

// write data to physical address
void _writeMemory(int physicalAddress, uint8_t data);
// read data from physical address. returns data read  (uint8_t)
uint8_t _readMemory(int physicalAddress);

// allocate a page
void _allocatePage(int virtualPageNumber, int frameNumber);

// set all data in a frame to 0
void _wipeMemoryFrame(int frameNumber);

// write pages frame to "disk"
void _writePageToDisk(int virtualPageNumber);
// read pages frame from "disk" back to memory
void _readPageFromDisk(int virtualPageNumber, int frameNumber);
// erase pages data from disk
void _deletePageFromDisk(int virtualPageNumber);
```

MemoryManager class: Usage

1. Initialization: Configure `MemoryManager()` with default or desired parameters
2. Allocation: Request virtual pages via `allocateAnyPage()`
3. Access: Read/write to virtual addresses (triggers page faults as needed)
4. Management: Delete pages when no longer needed

```
main() {
    MemoryManager mm;

    int pageAddress = mm.allocateAnyPage();

    mm.writeVirtualMemory(pageAddress, 0x67);

    int data = mm.readVirtualMemory(pageAddress);

    mm.deletePageTableEntry(pageAddress);

    return 0;
}
```

Live demo? Sure!

Thank you for your time!