CIT 5950

Recitation 9

HW3, Smart Pointers, and Processes

Logistics

- HW3
  - Due Thursday March 30th @ 11:59 PM

Homework 3 Overview

Overview

In HW3, you will be implementing a simplified version of simplevm

There are three core aspects of the simplevm implementation

- Swap file (provided to you)
- Page
- PageTable

Specification provided in the .h files for Page and PageTable

HIGHLY suggest that you follow the recommended approach in the writeup

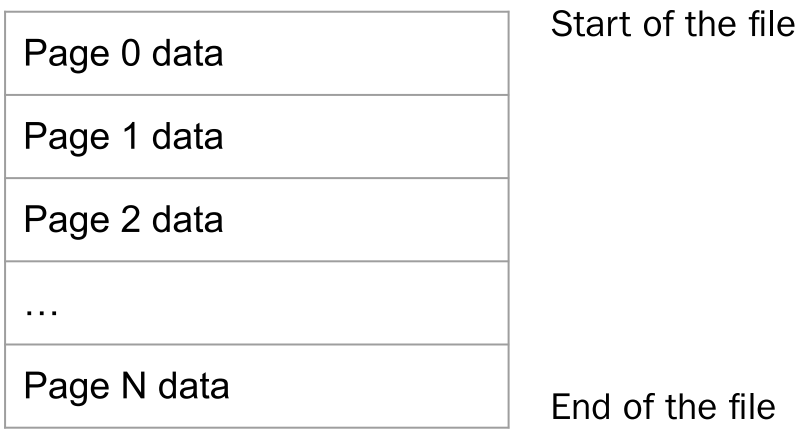
Swap File

A file containing all of the initial page contents and the contents of pages that aren’t loaded in to memory currently.

Swap files are used by “real” OS’s to store data that doesn’t fit into physical memory

(provided for you)

Swap file layout



*each page data is fixed size of 4096 bytes

Page

A page represents a single page of data in virtual memory

A page holds Page::PAGE_SIZE amount of bytes. (**4096 bytes**)

In the constructor, a page should load in its data from the swap file

On flush() a copy of the page's data is written to its location on the swap file

The access() and store() member functions modify the bytes_ and not the swap_file

You MUST **use an initializer list in the ctor and cctor** to initialize the swap_file_ reference.

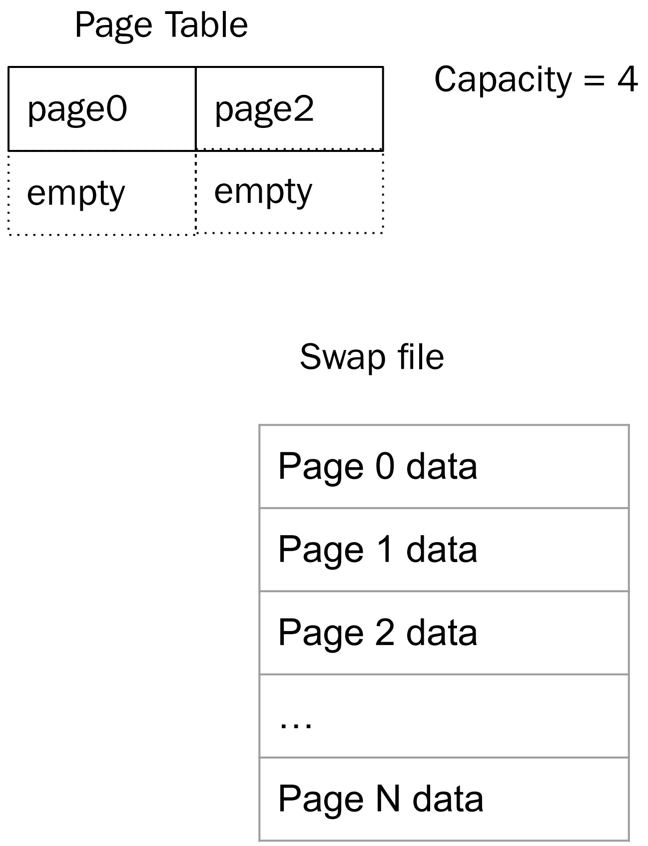
Page Table

Contains an LRU cache of Page's

Pages are considered "loaded into physical memory" when there is a Page object for that page.

Pages that aren't in memory are stored in the swap file

get_page() handles both cases where a page is loaded into memory and where it isn't



LRU Cache Key Properties

- We need to support **quick lookup**.
  - Can I quickly check if a Page is in the PageTable?

- We need to be able to **flexibly rearrange** Pages and maintain **sequential order.**
  - Can I easily move a Page from the middle of a data structure to the end?
  - Can I easily check what the next least used page is?

Alas, no single data structure meets both these requirements.

Think about what **combination** of data structures could fit these needs.

Casting Tips

From the writeup:

- You can assume the type you are reading/writing to the page data will be **primitives types.**

This means you can "build up" the bytes that make up an element of type T.

- Read from bytes_ member variable of Page class.
  - Where in the bytes array should you start reading from?
  - How many bytes should you read?
- Then use static_cast<T> to cast it into the desired type T.

Take a look at reinterpret_cast<T> when reading from the swap file into bytes_.

Any Questions?

Review Smart Pointers

Smart Pointers

- `std::unique_ptr`
  - Unique owner of the managed raw pointer – disabled cctor and op=
  - Used when you want to declare unique ownership of a pointer
- `std::shared_ptr`
  - Similar to `unique_ptr` but can be copied (via cctor or op=), uses reference counting to decide when to call delete on managed raw pointer
  - Most commonly used type of smart pointer in practice
- `std::weak_ptr`
  - Similar to `shared_ptr` but does not contribute to reference count
  - Almost always used with `shared_ptr`

Smart Pointer Usage

```
unique_ptr<int[]> uptr = unique_ptr<int[]>(new int[3])
```

- Main/typical usage:
  - Call ctor with new keyword or existing smart pointer (*e.g.*, unique_ptr<int[]> uptr(new int[3]))
  - Treat like a normal pointer (*i.e.*, use *, ->, [])

- Other methods that may be useful in *some* cases:
  - **unique_ptr** – .get(), .release(), .reset()
  - **shared_ptr** – .get(), .use_count(), .unique()
  - **weak_ptr** – .lock(), .use_count(), .expired()

Exercise 1

Exercise 1 Solution

Convert the `Node` struct to be "smart" by using `shared_ptr`s.

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}

  ~Node() { delete value; }

  T* value;
  Node<T>* next;
};
```

Exercise 1 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}

  ~Node() { delete value; }

  shared_ptr<T> value;
  shared_ptr<Node<T>> next;
};
```

Exercise 1 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(shared_ptr<T>(val)),
                               next(shared_ptr<Node<T>>(node)) {}

  ~Node() { delete value; }

  shared_ptr<T> value;
  shared_ptr<Node<T>> next;
};
```

Exercise 1 Solution

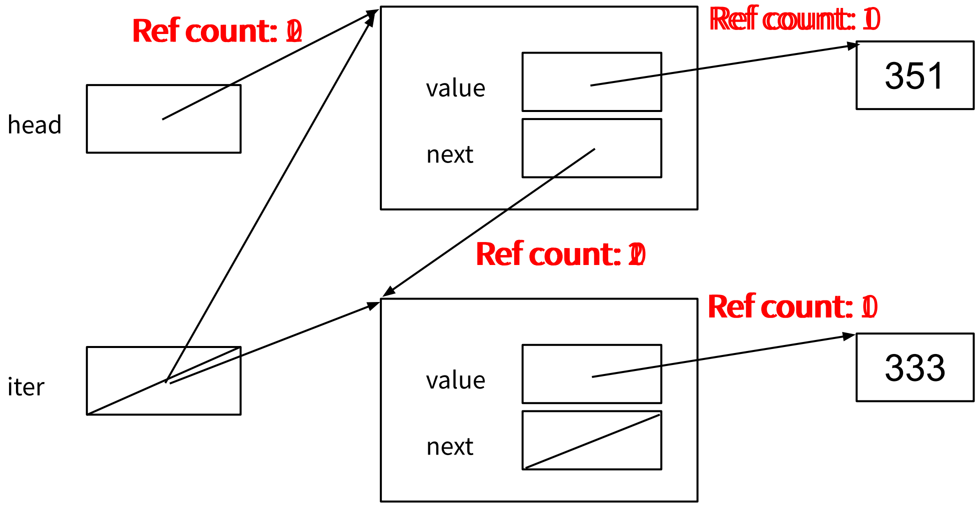
```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(shared_ptr<T>(val)),
                               next(shared_ptr<Node<T>>(node)) {}

  ~Node() { delete value; }

  shared_ptr<T> value;
  shared_ptr<Node<T>> next;
};
```

Exercise 1 Solution demo




```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  shared_ptr<Node<int>> head =
                    shared_ptr<Node<int>>(new Node<int>(new int(351), nullptr));
  head->next = shared_ptr<Node<int>>(new Node<int>(new int(333), nullptr));
  shared_ptr<Node<int>> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Processes review

Processes

- Created using `fork()` - the only function that returns twice!
  - Child gets 0
  - Parent gets new pid (process id) of child
- Essentially duplicates the parent process
- Get status of children with `waitpid(...)`
- Replace currently running process with a new one using `exec*()`
- Communicate between processes with pipe(int fds[2])

Processes and files/pipes

- If we create a pipe or access a file, there is one instance of it system wide
- When a process forks, it copies the file descriptors of the parent
- Multiple process can have access to the same file/pipe, but through their own file descriptors.
- When one process closes its file descriptors, other processes file descriptors remain open

Exercise 2

Exercise: fill in the blanks

```
int main (int argc, char** argv) {
  // create a pipe to send input to program
  int in_pipe[2];
  pipe(in_pipe);

  pid_t pid = fork();

  if (pid == 0) {
    // child
    close( in_pipe[1] ); // close writeend
    dup2( in_pipe[0] , STDIN_FILENO); // replace stdin with read end of pipe
    close( in_pipe[0]); // close read end since it has been duplicated

    // exec the program "./numbers" with no command line args
    string command( "./numbers");
    char* args[] = { "./numbers", nullptr };
    execvp(command.c_str(), args );

    // should NEVER get here
    return EXIT_FAILURE;
  } else {
```

Exercise: fill in the blanks

```
} else {
  close( in_pipe[0]               ); // close read end

  // write inputs to the pipe
  string inputs = "30\n40\n50\n6";
  wrapped_write(to_echo,  in_pipe[1]        );

  // close pipe so that exec'd
  // program knows there is no more piped contents to read
  close(   in_pipe[1]          );

  // wait for child to finish
  waitpid(pid, nullptr, 0 );
}
```