CIT 5950
Recitation 9

HW3 and Processes

Logistics

- HW3
  - Due Thursday April 12th @ 11:59 PM
- Survey #2: GitHub and Partner Info
  - Due Tomorrow April 5th @ 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 .hpp files for Page and PageTable

HIGHLY suggest that you follow the recommended approach in the writeup

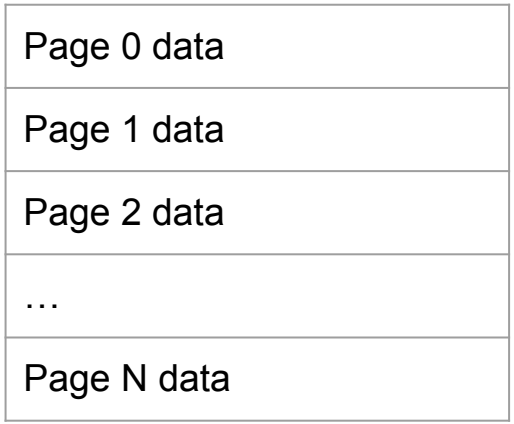
Swap File

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.

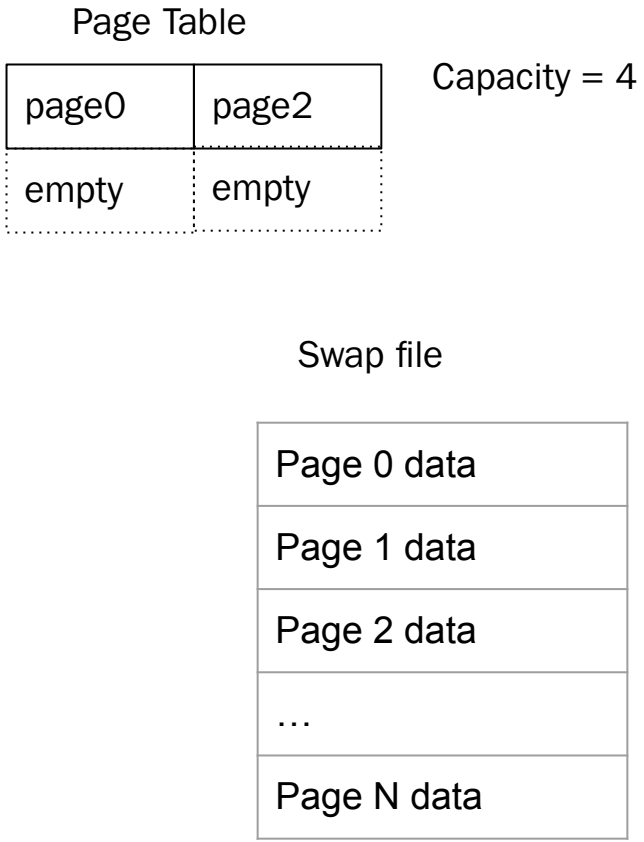
Page Table

Contains an LRU cache of Page's
Least recently used - LRU

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?

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
  - Child starts where fork() returns
- Get status of children with `waitpid(...)`
- Replace currently running process with a new one using `exec*()`
- Communicate between processes with pipe(int fds[2])
  - (more in lecture next week)

Exercise 1

Processes Exercise 1

How many times is :) printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

Processes Exercise 1

How many times is :) printed?

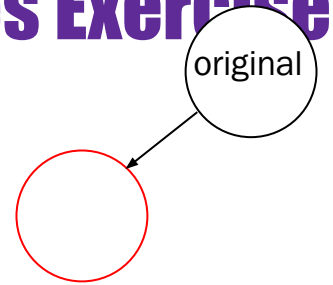
```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

Child processes copy the ‘i’ of the parent when fork is called and start executing in the loop

Processes Exercise 1


```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
     fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

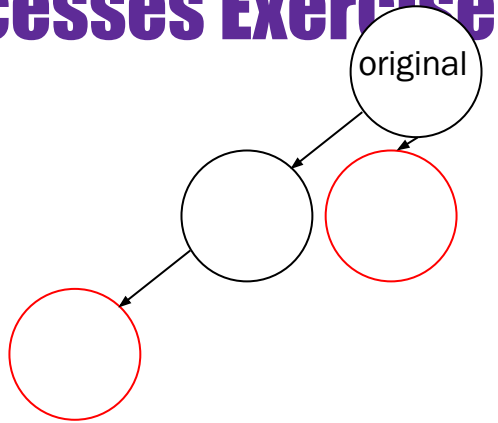
Processes Exercise 1



i = 0
Original forks, makes a child

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
     fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

Processes Exercise 1



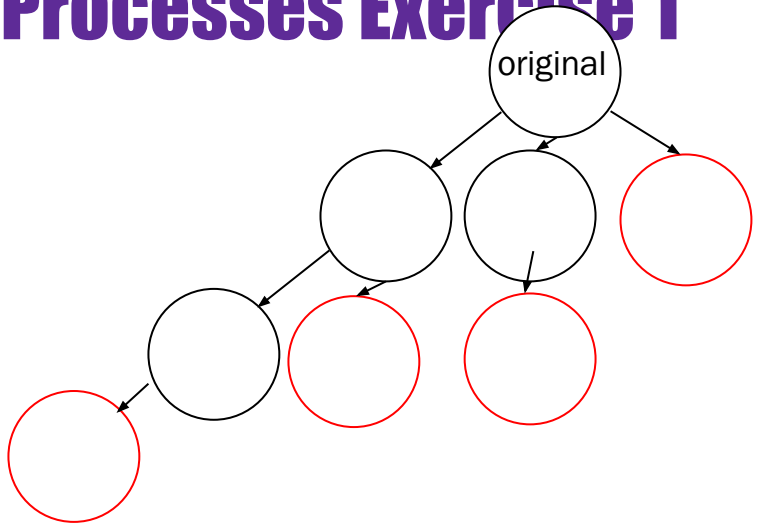
i = 1

Original and its child forks, each makes 1 child.

2 new processes created

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
     fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

Processes Exercise 1

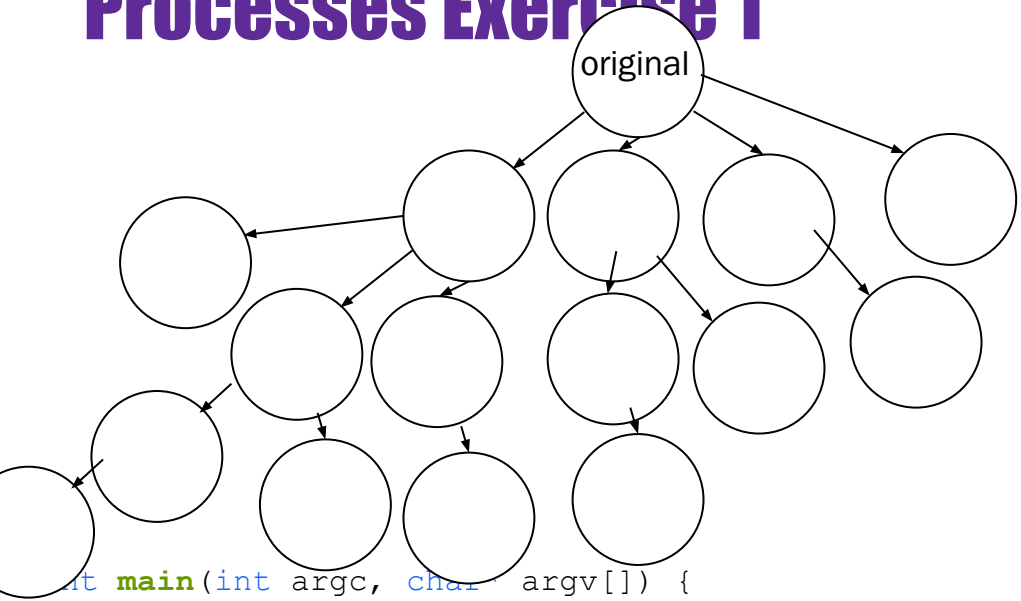



```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
     fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

i = 2

More children are made

Processes Exercise 1




```
int main(int argc, char* argv[]) {
   for (int i = 0; i < 4; i++) {
      fork();
   }
   cout << ":)\n"; // "\n" is similar to endl
   return EXIT_SUCCESS;
}
```

i = 3

16 processes, therefore 16 :) printed

4 iterations and number of processes doubles on each iteration.

Processes & Wait

fork() returns twice

- zero for the newly created child
- non-zero value to the parent. This value is the Process ID number of the child

The process ID number gotten by fork() can be used by the parent to wait or “join” a child process.

Processes & Wait

The process ID number returned by fork() to the parent can be used by the parent to wait or “join” the child process.

```
int main(int argc, char* argv[]) {
  pid_t pid =  fork();
  if (pid == 0) {
     cout << "child\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "parent\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

This code always prints”

```
child
parent
```

Parent process waits for child to finish, so `child` must be printed before `parent`

Exercise 2

Processes Exercise 2

```
int main(int argc, char* argv[]) {
  pid_t pid =  fork();
  if (pid == 0) {
     pid =  fork();
     if (pid == 0) {
       cout << "my\n";
     } else {
       cout << "mind\n";
     }
     exit(EXIT_SUCCESS);
  }
  pid =  fork();
  if (pid == 0) {
     cout << "skies\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "of\n"; // "\n" is similar to endl
}
```

What does are the possible outputs of this program?

Processes Exercise 2 Solution

```
int main(int argc, char* argv[]) {
  pid_t pid =  fork();
  if (pid == 0) {
     pid =  fork();
     if (pid == 0) {
       cout << "my\n";
     } else {
       cout << "mind\n";
     }
     exit(EXIT_SUCCESS);
  }
  pid =  fork();
  if (pid == 0) {
     cout << "skies\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "of\n"; // "\n" is similar to endl
}
```

Fork() is called three times, so we have four processes in total
(1 overall parent, 2 children, 1 grandchild)

Processes Exercise 2 Solution

```
int main(int argc, char* argv[]) {
  pid_t pid =  fork();
  if (pid == 0) {
     pid =  fork();
     if (pid == 0) {
       cout << "my\n";
     } else {
       cout << "mind\n";
     }
     exit(EXIT_SUCCESS);
  }
  pid =  fork();
  if (pid == 0) {
     cout << "skies\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "of\n"; // "\n" is similar to endl
}
```

Each print statement only comes after all processes have been created.

The only “synchronization” is the parent waits for the second child to exit, and then the parent prints.

So… “of” must come after “skies”
As long as that is maintained all orderings of the four print statements is possible

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

dup2() and redirection

We can use dup2() to redirect a file descriptor to something else.

Each process also has its own file descriptors tables.

Fork copies the file descriptor of the parent into the child.

Always writes
child
parent
To the file “hello.txt”

```
int main(int argc, char* argv[]) {
  int fd = open("hello.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     wrapped_write(fd, "child"); // helper function to write a string to a fd
     close(fd);
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  dup2(fd, STDOUT_FILENO);  // redirects STDOUT to the file specified by fd
  cout << "parent\n";  // writes to STDOUT_FILENO
}
```

Exercise 3

dup2 Exercise 3

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd); // fd is redirected
     wrapped_write(fd, "dust"); // helper function to write a string to a fd
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";
}
```

What is printed to the terminal and what is written to begin.txt?

dup2 Exercise 3 Solution

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd);
     wrapped_write(fd, "dust");
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";
}
```


fd points to cout

STDOUT and fd point to the same thing (begin.txt)

though we closed fd, STDOUT still points to file

dupe2 Exercise 3 Solution

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd);
     wrapped_write(fd, "dust");
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";
}
```

begin.txt contains:
star
platinum

what was printed:
dust
crusader

Bonus question: what do we know about the order of the words being printed/written?

Pipe()

More pipe information

Exercise 4

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 );
}
```