Lab Virtual Memory

Due: November 22, 2015

Note: This lab's stencil code comes with two directories. The `mmapcopy` directory will be used for the first portion of the lab and has its own Makefile. The second portion of the lab (after the checkpoint) will be done in the `fractal` directory.

1 mmapcopy

1.1 Introduction

1.1.1 Pawnee Goddesses

Leslie Knope has a new activity for the Pawnee Goddesses summer camp: programming. For the goddesses' first assignment, they will be learning how to program a fractal. Leslie wants her campers to learn how to generate fractals in the most efficient way possible, so she has turned to the students of cs033 for help. We have provided you with a prototype of (an image of) this fractal. Your task will be to quickly and efficiently generate the fractal. We will use virtual memory to accomplish this challenge.

1.1.2 Concepts and Functions

Before virtual memory, programmers would have to manually manage the amount of physical memory their processes had access to. This meant one often had to be very careful with loading too many variables or program text into memory at any one time. Virtual memory solves these problems by abstracting away the management of physical memory for the programmer. It provides a full 32-bit (or 64-bit) virtual address space to every process. This means that each application can assume that it has the exclusive use a large amount of physical memory, while the operating system provides it with whatever memory it actually needs, and handles the translation between virtual and physical addresses during a memory access.

Virtual memory allows programmers to worry less about memory management, and more about writing effective code. But virtual memory can be used by the programmer to improve I/O operations as well, skipping the kernel's buffer cache. Consider the following pseudocode:

```
char buf[BigEnough];
fd = open(file, O_RDWR);
for(i = 0; i < num_records; i++) {
        read(fd, buf, sizeof(buf));
        use(buf);
}
```

In this bit of code, we successively read and then use portions of a file, in chunks of size `BigEnough`. The OS goes to disk with every read call, and uses the I/O buffer to put the file into an OS specific section of memory. Then, the file is read from the OS's section of memory into the user program's memory as appropriate. But using a virtual memory mapping can reduce the number of buffers used in this process: the operating system instead maps data needed from external storage to a program's address space using virtual memory. The `mmap` system call is used to invoke this process:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

`mmap()` creates a new mapping of size `length` bytes and returns a pointer to the location of that new mapping, which is created at a virtual page boundary near `addr`, if `addr` is not `NULL`. If `addr` is `NULL`, then the pointer returned will be chosen by the OS (this is the most portable behavior and is often preferred). The new mapping contains the contents of the open file given by the file descriptor `fd` beginning from byte `offset` of that file. `prot` and `flags` are bit vectors which determine characteristics of the mapping. In particular, `prot` determines memory protection:

- `PROT_READ` indicates that the mapping may be read from.
- `PROT_WRITE` indicates that the mapping may be written to.
- `PROT_EXEC` indicates that the mapping may be executed.
- `PROT_NONE` indicates that the mapping may not be accessed.

`flags` determines behavior of the mapping. For example:

- `MAP_SHARED` indicates that updates to the mapping should be visible to other processes that map the same file. In addition, writes to the mapping will write back to that file.
- `MAP_PRIVATE` indicates that updates to the mapping should not be visible to other processes that map the same file, and that changes made in the mapping should not be written back to the file.
- `MAP_ANONYMOUS` indicates that the new mapping should not be backed by any file, causing the `fd` and `offset` arguments to be ignored. Instead the new mapping points to an anonymous file whose bytes are initialized to zero.

1.2 Assignment

1.2.1 mmap copy

Your first assignment is to code a program to copy files using `mmap()` instead of `read()` and `write()`. The `main()` routine in the provided stencil *mmapcopy.c* explains the steps for doing this.

Note: Don't forget to clean up and delete a mapping once it is no longer needed. You can use the system call `munmap()` to do this.

1.2.2 Tips

- `mmap` creates a new mapping in the virtual address space of the calling process. By mapping the file descriptors of the corresponding source and destination files, you can use `memcpy` to copy from one virtual address to the other. View man pages for more information.

- `lseek` is an useful system call to reposition the offset of an open file descriptor. You can use it to determine the size of a file.
- `munmap` system call deletes the mappings for the specified address. Make sure to call this on any virtual address you called before your program returns.
- `ftruncate` system call forces a given file to be of a specific length. Use this system call to ensure that the length of the destination file is the exact same size as the source file.

You should look at slide XXVIII-42 if you get stuck during this portion of the lab.

Warning: If you find yourself getting a "bus error" try using a for-loop to copy data 1 word (8 bytes) at a time instead of using `memcpy`. This is related to a bug in the Linux kernel.

1.2.3 Testing

To test your code, run the command `./mmapcopy <path to source file> <path to new file>`. You can use the provided image *cit.jpg* to ensure your program is working. The executable `cs033_file_equals` checks if two files (passed as its arguments)

1.2.4 Debugging

Remember to surround system calls with error checking.

1.3 Checkpoint

Once you've completed part 1, call a TA over and have them inspect your work. Once you've been checked off on part 1, you may move onto part 2.

2 Fractals

2.1 Introduction

2.1.1 Fractals

Learning Express thinks that their new toy is too plain and have called upon the students of CS33 to create a new fractal texture for it. Babies love fractals. Your task will be to efficiently generate fractal images using a combination of `fork()` and `mmap()` to speed up the fractal generation process.

2.1.2 A First Glance At Concurrent Programming

Previously in your computer science career, you have written primarily *sequential* programs: programs that execute instructions one-by-one in a specified order. You have also written programs that perform distinct tasks simultaneously using `fork()`. The first of these programming paradigms does not permit parallel execution of distinct tasks, but has no problems with memory access because all program execution comes from a single thread; the second of these paradigms does allow simultaneous execution of tasks, but does not enable memory to be shared between different threads of execution. But with *concurrent programming* techniques, a program can both execute distinct tasks in parallel as well as share memory between those threads of execution.

In this part of the lab, you will use `fork()` and `mmap()` to generate fractals that will be used as the new texture.

2.2 Assignment

2.2.1 The Starting Line

We have provided a program that generates a fractal called the Mandelbrot set using a single thread of execution. Because Leslie Knope wants her campers to learn how to generate image files fast, you will need to edit this program to perform the computation concurrently.

The program you will be working with consists of code from the following files:

- *image.h* - the *image.h* file contains struct definitions and method headers used to generate fractal data.
  - `color_t`

    This struct is used to represent the color of each pixel in the fractal image. Image colors are typically represented in a three-dimensional color space, where the dimensions represent the amounts of red, green, and blue light that produce the color. A `color_t` represents those coordinates as `unsigned char` values, which range from 0 to 255.
  - `int save_image_data(char *file_name, color_t *image_data, int width, int height)`

    This function saves the color data stored in the `image_data` array into the file `file_name`.
  - `void generate_fractal_region(color_t *image_data, float width, float height, float start_y, unsigned int rows)`

    This function generates color data for a particular subset of the pixels of the fractal and stores that data in the `image_data` array. Out of `height` rows of pixels, this function generates data for a number of rows given by the value of `rows`, beginning with row `start_y`.
- *image.c* - the *image.c* file contains definitions for the functions declared in *image.h*. You should not need to be familiar with the contents of this file to complete this lab.
- *colors.h* - the *colors.h* file contains declarations of three functions that generate a color channel value (an integer between 0 and 255). These functions are used in *image.c* to generate color data for each pixel of the fractal.
- *colors.c* - the *colors.c* file contains definitions of the color generation functions declared in *colors.h*. The colors of the fractal generated by the program depend on these functions - change them to change program output.
- *fractal.h* - the *fractal.h* file contains a declaration for the `generate_fractal()` function.
- *fractal.c* - the *fractal.c* file contains the definition for the `generate_fractal()` function, generating the fractal using a single line of execution.

- *main.c* - the *main.c* file contains the program `main()` function. This function parses the `argv` array for program arguments and calls `generate_fractal()`, the function you will be filling in (see below).

You are additionally provided with the file *fractal_forked.c* which provides an empty definition for `generate_fractal()`.

2.2.2 Running the Program

After you've compiled the program by running `make`, you can run the executable binary `fractal` with the following options:

- `-f <file>` and `--file <file>` change the file in which the fractal is saved.
- `-d <width> <height>` or `--dimensions <width> <height>` set the dimensions of the output image.

By default, `fractal` will store a 2000×2000 image in the file *out.png*. View this image using your favorite image viewer or editor; if you don't have one, run `okular out.png` to view the image.

2.2.3 Performance

In its present state, the `generate_fractal()` function of *fractal.c* does not utilize any form of concurrent execution. If you make the program and run `time ./fractal`, you'll notice that it takes a few seconds to execute. Performance of this program can be improved by distributing the task of fractal data generation amongst several *subprocesses*.

Your task for this lab is to re-implement the `generate_fractal()` method of the fractals program in the files *fractal_forked.c* using concurrent programming techniques to improve program performance.

2.2.4 Shared Memory Using `fork` and `mmap`

- `pid_t fork()`
- ```
  void *mmap(void *addr, size_t length, int prot,
       int flags, int fd, off_t offset)
  ```
- `int munmap(void *addr, size_t length)`
- `int waitpid(pid_t pid, int *status, int options)`

As you've seen previously, `fork()` is a system call which duplicates the calling process; the resulting child process receives a copy of its parent process' memory and stack, and begins execution from the point where `fork()` returns. Because memory is not ordinarily shared between the parent and child processes, filling in part of the `color_t` array in each child process will uselessly generate an incomplete image in each child process. To successfully generate the desired image, some mechanism by which the processes can share memory is needed.
  ```

Virtual memory provides such a mechanism with the system call `mmap()`. As you experienced above, this system call creates a new *mapping* in a process' virtual address space - that is, it creates a new region of memory for use by the process. You will find that a combination of the flags described above will set up a convenient region of shared memory between processes; this is exactly what is needed here. It doesn't matter where the mapping is located, so `addr` can be `NULL`.

Note: `mmap()` returns `(void *) -1` if an error occurs.

Once you've set up the shared memory region and `fork()`ed off each child process, the parent must wait until each child has completed its task before it saves the fractal data. To do this, you'll use the `waitpid()` system call to wait for each child process. To do so, you'll need to save the process ID of each child process.

Once the memory mapping is no longer needed, make sure to clean up and delete the mapping. Do this after you've saved the fractal data. You should also do this in each child process before exiting — cleaning up before exiting is always good practice.

Task: use `fork()`, `mmap()`, and `munmap()` to edit *fractal_forked.c* so that work done to generate the fractal is divided amongst several child processes. This functionality should be abstracted to an arbitrary number of child processes; use the `workers` argument to `generate_fractal()`. By default, this argument is 4, but this value can be changed by passing the flag `-n` <`workers`> or `--workers` <`workers`> as an argument to the fractals program. You need not handle the case where the height of the generated image is not divisible by the number of child processes. You can build the fractals program with the command `make forked`, which includes *fractal_forked.c* instead of *fractal.c*. Doing so creates the executable *fractal_forked*.

3 Getting Checked Off

Try to get the runtime of *fractal_forked* to be under 1.4 seconds. Once you've completed the lab, call a TA over and have them inspect your work. If you do not complete the lab during your lab session, you can get credit for it by completing it and bringing it to lab hours before next week's lab. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.