CMPSC 472 Project one by Daksh Upadhyay

Description of the project: This project is an single c file which does multiple operations and uses the API(s) provided by OS to create processes, threads and test memory sharing/message passing along with loading file parallely by the user input from the Command line interface

GitHub link: https://github.com/Daksh14/cmpsc-472-hw

Code structure:

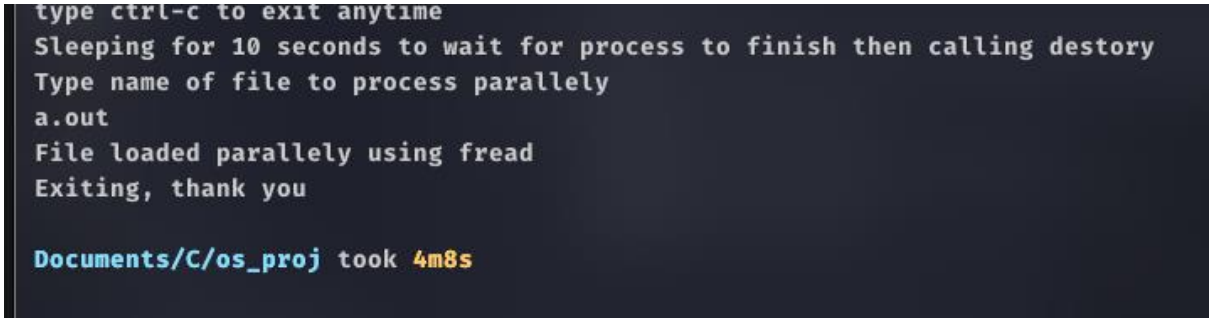
```
std::vector<std::string> process_names;

pthread_mutex_t lock;

// Create an individual process
struct Process {
  int num_of_threads;
  // 3 threads per process
  pthread_t threads[4];
};

// Create the manager struct
struct ProcessManager {
  // max 2 processes
  Process process[2];
  // pid(s) for the processes
  pid_t pid[2];
  // the callback for the thread
  static void *thread_callback(void *i) {
    std::cout << "Simulating shared memory with mutex" << std::endl;
    pthread_mutex_lock(&lock);
    char buff[100];

    snprintf(buff, sizeof(buff), "From thread: %p\n", pthread_self());

    std::string buffAsStdStr = buff;

    process_names.push_back(buffAsStdStr);

    pthread_mutex_unlock(&lock);
  };
```

I have two structs here, one is the ProcessManager which holds the pids of the processes we create and it holds threads, the number of threads that are created per process can be specified when calling the constructor of the ProcessManager
Below is the constructor of the process manager

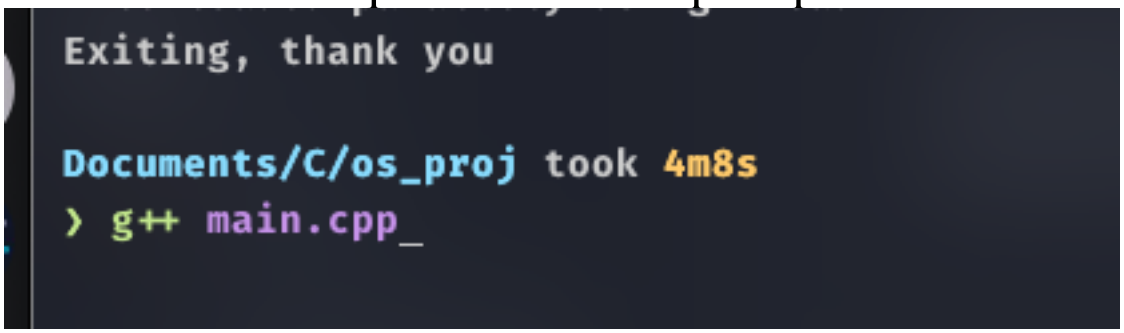
We support n number of threads per process!

```cpp
int ProcessManager_create(int num_of_threads) {
  // create TWO child processes to keep IPC as simple as possible
  if (fork() == 0) {
    pid_t current_pid = getpid();
    pid[0] = current_pid;
    // child process
    // record time
    std::cout << "Init Child process, PID from fork: " << current_pid
              << std::endl;

    for (int i = 0; i < num_of_threads; i++) {
      auto t0 = std::chrono::steady_clock::now();
      pthread_create(&process[0].threads[i], NULL,
                     ProcessManager::thread_callback, NULL);

      pthread_join(process[0].threads[i], NULL);
      auto t1 = std::chrono::steady_clock::now();
      std::chrono::duration<uint64_t, std::nano> elapsed = t1 - t0;
      uint64_t duration = elapsed.count();

      std::cout << "Thread ended, took duration: with shared memory "
                << duration << std::endl;
    }
    std::cout << "Allocated memory for child process one " << std::endl;

    for (auto const &c : process_names)
      std::cout << c << ' ';

    exit(0);
  } else {
    if (fork() == 0) {

      pid_t current_pid = getpid();
      pid[1] = current_pid;

      std::cout << "Init Second Child process, PID from fork: " << current_pid
                << std::endl;
      for (int i = 0; i < num_of_threads; i++) {
        auto t0 = std::chrono::steady_clock::now();
        int fd[2];
        pipe(fd);

        int *args = (int *)malloc(sizeof(*args));
        *args = fd[0];

        pthread_create(&process[1].threads[i], NULL,
                       ProcessManager::thread_callback_with_message_passed,
                       args);

        char *input_str = "This is a message sent";

        write(fd[1], input_str, strlen(input_str) + 1);
        close(fd[1]);

        pthread_join(process[1].threads[i], NULL);

        auto t1 = std::chrono::steady_clock::now();
        std::chrono::duration<uint64_t, std::nano> elapsed = t1 - t0;
        uint64_t duration = elapsed.count();

        std::cout << "Thread ended, took duration: with message passing: "
                  << duration << std::endl;
      }
      exit(0);
    }
  }

  // wait for the processes to finish
  while (wait(NULL) > 0)
    ;

  return 0;
}
```

The main function calls the struct constructor and constructs the processes and the threads, it also waits for them to finish and then asks for the file name from the user in the command line. When you run the file you can type the name of the file which needs to be read and processed, multiple files can be passed by separating them with the \n new line keyword

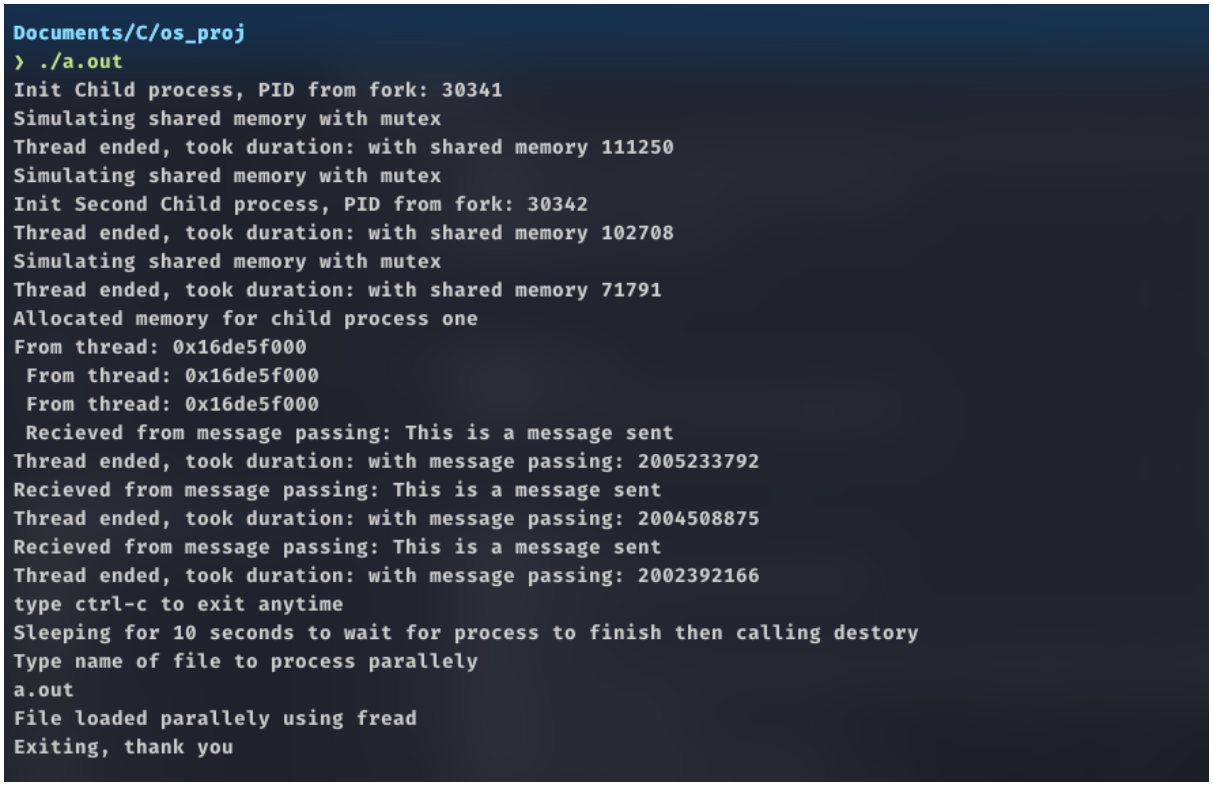


How to use: compile file with c plus plus



then run the binary output, should yield the following

result.



Verification:
1. In the above screenshot you can see the From Thread: happens exactly 3 times that's accurate to how many threads we have passed in to create in main.cpp file

2. There are exactly two processes created and we can verify that by looking at the above screenshot, we see "PID from fork" printed two times and **the PID(s) are consecutive meaning they were created together**.

3. The time it takes for data to transfer with message passing is significantly higher than shared memory because pip channels have overhead which makes sense.

Findings:

I found out how to implement multiple processes and manage

threads in between those created processes using C plus plus and also learned how to process an file parallely using fread which is helpful as it will speed up opening of large binary files.

I faced the challenge implementing an dynamic array to support n number of threads per process for that I had to switch to c plus plus in order to use std::vector and store the pthread id(s) there, I also used it with an mutex to implement shared memory

Limitations: Currently only supports 2 processes, in the future support n processes

Area for improvement: Allow n number of processes, allow n number of threads per processes, add an custom thread function passes to the struct to be executing instead of an static function (hard)