Lab 4

Alexander Fountain, Jarred Parr

1. This program executes a thread which does some non-blocking operation. Because of the artificial overhead of the `sleep(1)` call added to the thread, there must be a sleep call added to the main function to allow for the program to observe the thread behavior. Otherwise the parent will have exited, cleaned up, and competed before the thread could even run its code.
2. It initially prints a jumbled looking output of "Hello"'s and "World"'s. It appears that these values are fighting for access to stdout. Since there isn't a blocking call on the print, the output gets mangled when placed on the stdout stream, which is why we see the stream of `World` following the first `Hello`, then all of the `Hello`'s.
3. This improves upon question two. The sleep allowed for the threads to sync a bit better and produce a more consistent output across the sub-processes. Inspection of top makes it appear as though the processes are able to execute in much closer parallel. This is what enables the words to align properly.
4. Linux uses Many-to-one. This is because, when we observe the blocking call on a thread, we see that is does not hinder the other threads from competing for access to the standard output stream. When observing the process via the ps and top tools
5. In this run we see that the parent sees 5 as the shared data value. The first child spawned off looks at the shared data value and also sees it as 5 because the parent hasn't increased the value to 6 yet. The second child looks at the shared data value and also sees 5 because neither the parent or the other child has updated the value yet. The first child hasn't updated it's value yet becasue of the sleep(1) before the increment. The first child looks at the shared data value again and now sees 7 because the parent and the first child have increased it by one each. The second child looks at the value and sees 8 because it increased the value by one more. Finally, the parent prints 8 as the final value. This output shows that there is time between when the shared value is updated vs when it is read even though it looks like it should update immediately and give the output, 5, 6, 7, 8, 8.
6. The thread specific data is passed via a pointer (void* arg) that is then used to specify the data type of char val_ptr. You can take the void* argument to cast a different type into the incoming data. This is how we are allowed to share data of any arbitrary (known) type.

Practical

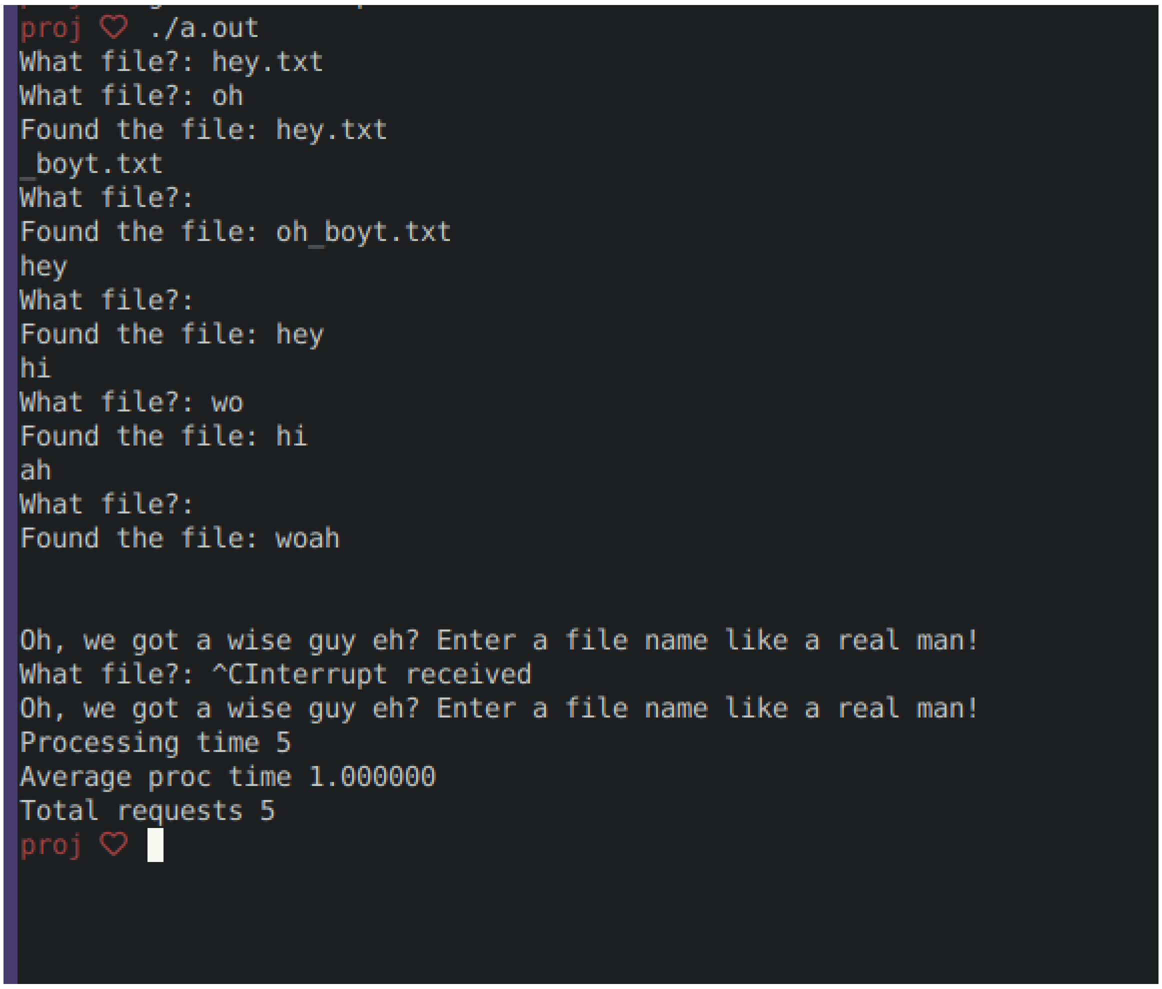
serv.c

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

size_t interrupt = 0;
size_t requests = 0;
size_t runtime = 0;
```

```c
pthread_mutex_t req_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t time_mutex = PTHREAD_MUTEX_INITIALIZER;

void sig_handler(int num) {
  printf("Interrupt received");

  interrupt = 1;

  // Unlock our mutextes
  pthread_mutex_unlock(&req_mutex);
  pthread_mutex_unlock(&time_mutex);
}

void* worker(void* arg) {
    pthread_mutex_lock(&req_mutex);
    pthread_mutex_lock(&time_mutex);

    char* filename = (char*) arg;

    size_t sleep_duration;

    if ((rand() % 10) < 8) {
      sleep_duration = 1;
    } else {
      sleep_duration = (rand() % 3) + 7;
    }

    sleep(sleep_duration);

    printf("\nFound the file: %s\n", filename);

    runtime += sleep_duration;
    requests++;

    pthread_mutex_unlock(&req_mutex);
    pthread_mutex_unlock(&time_mutex);

    pthread_exit(NULL);
}

int server() {
  struct sigaction action;
  action.sa_handler = &sig_handler;
  action.sa_flags = 0;

  pthread_t thread;
  int status = 0;
  char input[1024];

  if (pthread_mutex_init(&req_mutex, NULL) != 0) {
    perror("Failed to make requests mutex");
    return EXIT_FAILURE;
  }
```

```
    if (pthread_mutex_init(&time_mutex, NULL) != 0) {
      perror("Failed to make time mutex");
      return EXIT_FAILURE;
    }

    if (sigaction(SIGINT, &action, NULL) < 0) {
      perror("Failed to catch SIGINT");
      return EXIT_FAILURE;
    }

    while (!interrupt) {
      printf("What file?: ");
      fgets(input, 1024, stdin);

      input[strcspn(input, "\n")] = 0;

      if (strlen(input) == 0) {
        printf("\nOh, we got a wise guy eh? Enter a file name like a real man!\n");
        continue;
      }

      if ((status = pthread_create(&thread, NULL, worker, input)) != 0) {
        perror("Thread create error");
        return EXIT_FAILURE;
      }
    }


    float proc = (float)runtime / requests;
    printf("Processing time %ld\n", runtime);
    printf("Average proc time %f\n", proc);
    printf("Total requests %ld\n", requests);

    pthread_mutex_destroy(&req_mutex);
    pthread_mutex_destroy(&time_mutex);
    return EXIT_SUCCESS;
  }

  int main() {
    server();
  }
```

Sample Output



```
proj ♡ ./a.out
What file?: hey.txt
What file?: oh
Found the file: hey.txt
_boyt.txt
What file?:
Found the file: oh_boyt.txt
hey
What file?:
Found the file: hey
hi
What file?: wo
Found the file: hi
ah
What file?:
Found the file: woah


Oh, we got a wise guy eh? Enter a file name like a real man!
What file?: ^CInterrupt received
Oh, we got a wise guy eh? Enter a file name like a real man!
Processing time 5
Average proc time 1.000000
Total requests 5
proj ♡
```