CSE344 – Systems Programming

Homework 3 Report

Oğuzhan Agkuş – 161044003

The main idea of this homework is to calculate multiplication of two matrices in distributed fashion with using pipes and to calculate the singular values of result matrix.

The program gets 3 command line arguments: 2 input path and a positive integer value n. It creates $2^n x 2^n$ matrices for each input file. It reads the input file, convert each character to ASCII value and store it into corresponding matrix. Then the program creates 4 children processes. The parent sends some parts of the main matrix to children by using pipes. After then each child calculates a quarter of the result matrix and sends back to parent. After that the parent combines the partial results and calculates the singular value of result matrix.

My submission is consisting of 6 files:

- svd.h
- svd.c
- helper.h
- helper.c
- program.c
- makefile

SVD files are outsourced. They are not my code. I only edit them to use in my program. Its owner is written on the top of the file.

Helper files contains “includes”, a struct and some useful functions that do routine stuff like dynamically allocation of matrices, freeing, printing and multiplying these matrices. Also, it has a function that fills the matrices with a file. The struct name bidirect_pipe_t represents a bidirectional pipe. It consists of the two pipes, in other words, two file descriptor arrays, each one is for one direction: parent to child, child to parent. “going” is used for transferring from parent to child and “coming” is used for transferring from child to parent.

```
typedef struct bidirect_pipe {
  int going[2];
  int coming[2];
} bidirect_pipe_t;
```

```
unsigned int **create_matrix(size_t m, size_t n);
unsigned int **multiplication(unsigned int ** matrix_1, unsigned int ** matrix_2, size_t m, size_t n, size_t
int fill_square_matrix(char *path, unsigned int ***matrix, size_t n);
void free_matrix(unsigned int **matrix, size_t n);
void print_matrix(unsigned int **matrix, size_t m, size_t n);
void print_array(double *array, size_t n);
```

Firstly, it checks the arguments. In case of any problem, prints the error and exits with -1.

After checking arguments, it tries to create pipes. It has two types of pipe. First one is for synchronization barrier and the other is for data transferring between parent and child. It creates a pipe for each child. In case of error it prints errno explanation and exits with -1.

```
if(pipe(barrier) == -1) {
    fprintf(stderr, "Failed to create barrier pipe: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  for(i = 0; i < 4; i++) {
    if((pipe(pipes[i].going) == -1) || (pipe(pipes[i].coming) == -1)) {
      fprintf(stderr, "Failed to create data pipes: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
    }
  }
```

After creating pipes, it sets a new signal handler for SIGCHLD signal. Also creates a signal mask to block SIGCHLD to prevent its delivery if a child terminates before the parent commences the sigsuspend() loop at the end. In case of error it prints errno explanation and exist with -1.

```
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sa.sa_handler = sigchld_handler;

  if(sigaction(SIGCHLD, &sa, NULL) == -1) {
    fprintf(stderr, "Cannot set new signal handler: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  sigemptyset(&block_mask);
  sigaddset(&block_mask, SIGCHLD);

  if(sigprocmask(SIG_SETMASK, &block_mask, NULL) == -1) {
    fprintf(stderr, "Cannot set signal mask: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
```

The program has a global variable for live child count. In case of successful fork, it increases, and in case of SIGCHLD catch it decreases. The signal handler for SIGCHLD is updates this variable. The signal handles is a static function. It performs a waitpid() call with WNOHANG flag for any children.

```c
static void sigchld_handler(int signal) {
  int status, saved_errno;
  pid_t child_pid;

  saved_errno = errno;

  while((child_pid = waitpid(-1, &status, WNOHANG)) > 0) {
    // printf("Handler reaped child %d\n", child_pid);
    live_children_count--;
  }

  if(child_pid == -1 && errno != ECHILD) {
    fprintf(stdout, "No child processes!\n");
    exit(EXIT_FAILURE);
  }

  errno = saved_errno;
}
```

It performs a sigsuspend() at end of the program. Thanks to this loop all pending SIGCHILD signals are handled and live child count is updated. When all children are exits, the loop ends.

```c
sigemptyset(&empty_mask);

while(live_children_count > 0) {
  if(sigsuspend(&empty_mask) == -1 && errno != EINTR) {
    fprintf(stderr, "Sigsuspend error: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
}
```

If all stuff up to here is done with no errors, it starts to fork children. It forks 4 child process. In case of error it exits with -1. Because of error checking at each possible situation, all processes exit with -1 synchronously. Also, if the processes have allocated memory, they free them just before the exiting.

```c
for(i = 0; i < 4; i++) {
  live_children_count++;

  switch(fork()) {
    case -1:
      live_children_count--;
      fprintf(stderr, "Fork error!\n");
      exit(EXIT_FAILURE);

    case 0: // Child
      ..
      ..
```

First thing done after the creating processes is closing unused file descriptors of pipes. After that child process start to wait for data from parent. At the same moment the parent calls fill square matrix function and try to produce data. If there is no enough data to fill matrices, it will exit with -1 and free allocated resources, children are also exit because read call will fail. If there is no error, the parent firstly sends the size of the arrays. Just after the children allocate some memory for the data will come from parent and start to wait. The parent sends quarters to children row by row. Then it sends all data, it free resources as soon as possible and start to wait with synchronization barrier. At that time the children are calculating the multiplication of quarters. After all calculations done parent continue to run and reads results from the children. While parent reading it is also combining the result matrix. Children are exit after sending all data. At that time the parent ready to calculate singular values of result matrix. If any error occurs during this calculation it exits with -1. Finally, it prints the singular values to terminal and exit with 0.

I create automated test cases with a python script. If the n value is bigger than 10, the hardware might be not enough. It uses approximately 95% of CPU. Also, in case of n >= 15, the kernel kills the process directly. I test at most with n=11, it took a long time. It is work well at most n = 10. It can calculate with 11 < n < 15, but it will take very long time. If the hardware is not powerful, the system can be unavailable. So, I set an upper bound for n, because the evil user can give big value and the system will crash.

I added 2 examples output of my program with smaller n values, because it is impossible to get screenshot of long out output.

```
oguzhan@ubuntu:~/Desktop/CSE344/Homework/HW03$ ./program -i ./input -j ./input -n 2

-> Singular values:

136224.315
4954.243
1736.355
50.008

oguzhan@ubuntu:~/Desktop/CSE344/Homework/HW03$ ./program -i ./input -j ./input -n 3

-> Singular values:

568962.339
11854.328
7334.744
5348.956
1853.203
336.197
134.005
816.999
```