Anirvin Iyer
CSE D- 32
240905318

OS LAB 4

1. Write a C program to create a child process. Display different messages in parent process and child process. Display PID and PPID of both parent and child process. Block parent process until child completes using wait system call.

```c
int main() {
    pid_t pid;

    // Create child process
    pid = fork();

    if (pid < 0) {
        // Error occurred
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        // Child process block
        printf("Child Process:\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Message: Hello from the child process!\n");
    }
    else {
        // Parent process block
        // Wait for child to complete
        wait(NULL);

        printf("Parent Process:\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Message: Child has finished. Hello from the parent process!\n");
    }

    return 0;
}
```

```
(base) mca@computinglab25-01:~/Documents/os_240905318/lab4$ ./pgm1
Child Process:
  PID: 8757
  PPID: 8756
  Message: Hello from the child process!
Parent Process:
  PID: 8756
  PPID: 7192
  Message: Child has finished. Hello from the parent process!
```

2. Write a C program to load the binary executable of the previous program in a child process using exec system call.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    // Create child process
    pid = fork();

    if (pid < 0) {
        // Error occurred
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        // Child process block
        printf("Child Process (before exec):\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Message: Now loading the previous program using exec...\n");

        // Replace child process image with the previous program
        // Assume the previous program was compiled as "pgm1"
        execl("./pgm1", "pgm1", (char *)NULL);
        perror("execl failed");
        exit(1);
    }
    else {
        // Parent process block
        // Wait for child to complete
        wait(NULL);

        printf("Parent Process:\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Message: Child has finished executing the binary.\n");
    }

    return 0;
}
```

```
(base) mca@computinglab25-01:~/Documents/os_240905318/lab4$ ./pgm2
Child Process (before exec):
  PID: 8851
  PPID: 8850
  Message: Now loading the previous program using exec...
Child Process:
  PID: 8852
  PPID: 8851
  Message: Hello from the child process!
Parent Process:
  PID: 8851
  PPID: 8850
  Message: Child has finished. Hello from the parent process!
Parent Process:
  PID: 8850
  PPID: 7192
  Message: Child has finished executing the binary.
```

3. Create a zombie (defunct) child process (a child with exit() call, but no corresponding wait() in the sleeping parent) and allow the init process to adopt it (after parent terminates). Run the process as background process and run “ps” command.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        // Child process
        printf("Child process:\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Exiting now to become zombie...\n");
        exit(0);  // Child exits immediately
    }
    else {
        // Parent process
        printf("Parent process:\n");
        printf("  PID: %d\n", getpid());
        printf("  PPID: %d\n", getppid());
        printf("  Sleeping... not calling wait()\n");

        // Sleep long enough to observe zombie state
        sleep(30);

        printf("Parent terminating... child will be adopted by init.\n");
    }

    return 0;
}
//when parent goes to sleep or doesnt call sleep, when child exit()'s then
//init process adopts the child..
//i.e. : whenever parent terminates before child, child is adopted
```

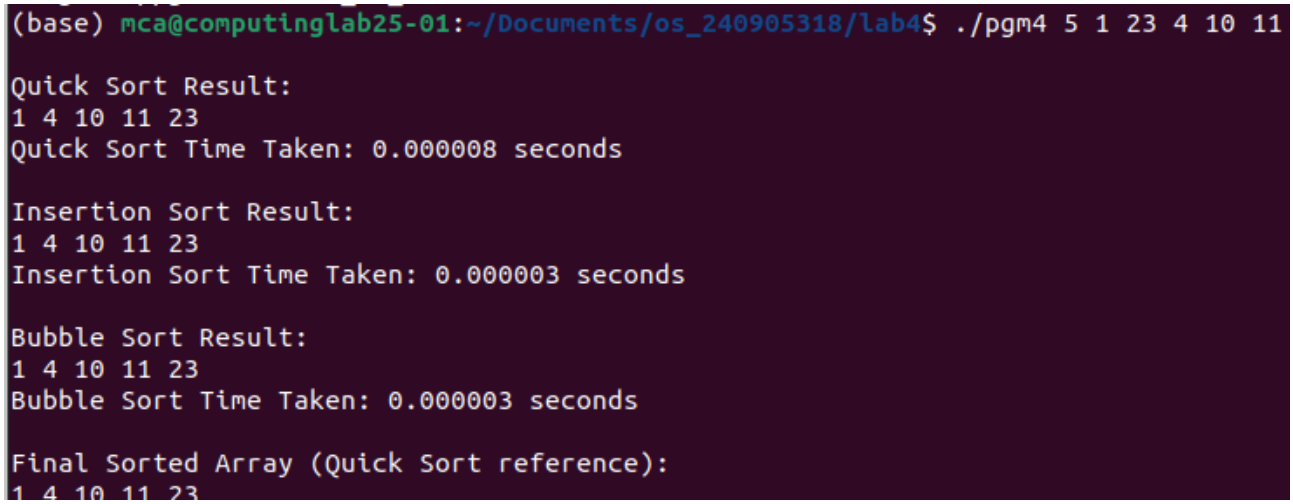
```
(base) mca@computinglab25-01:~/Documents/os_240905318/lab4$ ./pgm3
Parent process:
  PID: 8303
  PPID: 7192
  Sleeping... not calling wait()
Child process:
  PID: 8304
  PPID: 8303
  Exiting now to become zombie...
```

4. Write a multithreaded program that performs different sorting algorithms. The program should work as follows: the user enters on the command line the number of elements to sort and the elements themselves. The program then creates separate threads, each using a different sorting algorithm. Each thread sorts the array using its corresponding algorithm and displays the time taken to produce the result. The main thread waits for all threads to finish and then displays the final sorted array.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>


typedef struct {
    int *arr;
    int n;
    char algo_name[20];
} SortArgs;
void printArray(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
void bubbleSort(int *arr, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
void insertionSort(int *arr, int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
int partition(int *arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
```

```
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;
    return i+1;
}

void quickSort(int *arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi-1);
        quickSort(arr, pi+1, high);
    }
}

void* sortThread(void *args) {
    SortArgs *sargs = (SortArgs*) args;
    int *arr_copy = malloc(sargs->n * sizeof(int));
    memcpy(arr_copy, sargs->arr, sargs->n * sizeof(int));

    clock_t start = clock();

    if (strcmp(sargs->algo_name, "Bubble Sort") == 0) {
        bubbleSort(arr_copy, sargs->n);
    } else if (strcmp(sargs->algo_name, "Insertion Sort") == 0) {
        insertionSort(arr_copy, sargs->n);
    } else if (strcmp(sargs->algo_name, "Quick Sort") == 0) {
        quickSort(arr_copy, 0, sargs->n - 1);
    }

    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\n%s Result:\n", sargs->algo_name);
    printArray(arr_copy, sargs->n);
    printf("%s Time Taken: %f seconds\n", sargs->algo_name, time_taken);

    free(arr_copy);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <number_of_elements> <elements...>\n", argv[0]);
        return 1;
```

```
    }

    int n = atoi(argv[1]);
    if (argc != n + 2) {
        printf("Error: You must provide %d elements.\n", n);
        return 1;
    }

    int *arr = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i+2]);
    }

    pthread_t threads[3];
    SortArgs args[3] = {
        {arr, n, "Bubble Sort"},
        {arr, n, "Insertion Sort"},
        {arr, n, "Quick Sort"}
    };

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, sortThread, (void*)&args[i]);
    }
    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }
    quickSort(arr, 0, n-1);
    printf("\nFinal Sorted Array (Quick Sort reference):\n");
    printArray(arr, n);

    free(arr);
    return 0;
}
```



5. Write multithreaded program that generates the Fibonacci series. The program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program then will create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution the parent will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


typedef struct {
    int *fib;
    int n;
} FibData;


void* generateFibonacci(void *arg) {
    FibData *data = (FibData*) arg;

    if (data->n > 0) data->fib[0] = 0;
    if (data->n > 1) data->fib[1] = 1;

    for (int i = 2; i < data->n; i++) {
        data->fib[i] = data->fib[i-1] + data->fib[i-2];
    }

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n <= 0) {
        printf("Number of terms must be positive.\n");
        return 1;
    }
    int *fib = malloc(n * sizeof(int));
    if (fib == NULL) {
        perror("malloc failed");
        return 1;
    }
    FibData data = {fib, n};
    pthread_t tid;
    pthread_create(&tid, NULL, generateFibonacci, (void*)&data);
    pthread_join(tid, NULL);
    printf("Fibonacci series (%d terms):\n", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", fib[i]);
    }
    printf("\n");
    free(fib);
    return 0;
}
```

```
(base) mca@computinglab25-01:~/Documents/os_240905318/lab4$ ./pgm5 30
Fibonacci series (30 terms):
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368
75025 121393 196418 317811 514229
```