ENCS3390

Task 1: Process and Thread Management

Student name: Kareem Qutob

Student ID: 1211757

Instructor name: Bashar tahayne

Section: 4

Part 1: Process Management:

Description: Implement a program with a parent process that creates child processes using fork(). Use message passing with pipes for IPC between parent and child.

Code:

```
void multiplyMatricesByProcesses(int result[MATRIX_SIZE][MATRIX_SIZE], int numOfProcesses) {
    int FD[numOfProcesses][2];// pipes will be my ipc
    for (int i = 0; i < numOfProcesses; i++) {// creating pipes for each child proccess
        if (pipe( pipedes: FD[i]) == -1) {
            printf( format: "pipe failed\n");
            exit( status: EXIT_FAILURE);
        }
    }

    int rowsPerProc = MATRIX_SIZE / numOfProcesses;// to divide how many rows every proccess will take
    int remaining = MATRIX_SIZE % numOfProcesses;
    int *partialResult = (int *) malloc( size: rowsPerProc * MATRIX_SIZE * sizeof(int));// it will hold partial result of multiblcation

    for (int i = 0; i < numOfProcesses; i++) {// forking for number of proccesses
        pid_t pid = fork();
        if (pid == -1) {
            printf( format: "fork error\n");//error handling
            exit( status: EXIT_FAILURE);
        }
        if (pid == 0) {
            close( fd: FD[i][0]);// closing read pipes when writing
            int startRow = i * rowsPerProc;
            int end_row;

            if (i == numOfProcesses - 1) {// deciding in which rows each proccess will. beging to which row
                end_row = startRow + rowsPerProc + remaining;
            } else {
                end_row = startRow + rowsPerProc;
            }

            multiplyMatrixPartial(result,  start_row: startRow, end_row);// calling partial mul function
```

```
            for (int row = 0; row < rowsPerProc; row++) {
                for (int col = 0; col < MATRIX_SIZE; col++) {
                    partialResult[row * MATRIX_SIZE + col] = result[startRow + row][col];
                }
            }

            write( fd: FD[i][1],  buf: partialResult,  n: rowsPerProc * MATRIX_SIZE * sizeof(int));// writing results in pipes
            close( fd: FD[i][1]);
            exit( status: 0);
        }
    }

    for (int i = 0; i < numOfProcesses; i++) {
        close( fd: FD[i][1]);  // Close write end of the pipe

        read( fd: FD[i][0],  buf: partialResult,  nbytes: rowsPerProc * MATRIX_SIZE * sizeof(int));// father proc reading values from child pipe
        close( fd: FD[i][0]);

        // Copy the partial result back to the result matrix
        int startRow = i * rowsPerProc;
        for (int row = 0; row < rowsPerProc; row++) {
            for (int col = 0; col < MATRIX_SIZE; col++) {
                result[startRow + row][col] = partialResult[row * MATRIX_SIZE + col];
            }
        }
    }

    free( ptr: partialResult);

    for (int i = 0; i < numOfProcesses; i++) {// wait for sons procs
        wait( stat_loc: NULL);
    }
```

In the code above there is an implementation of a program that has a parent process that creates child processes using fork. my function will take a parameter that determines the number of wanted processes and will do a loop that creates child processes using a fork for the number of wanted processes and will call a partial multiplication matrix function to divide work between processes, my choice of IPC between the parents and child was pipes, and I created an array of pipes corresponding to the number of wanted processes, for every process I use the write end of the pipe to store the partial result produced from every process, there is another loop responsible to read data stored in the pipes by the parent process, and a loop with function wait that is responsible to make the father process wait for every child process to end its execution.

Part 2, 4: Multithreaded Processing and Thread Management

*note: these two parts address similar topics so I decided to join them 😊

Using the Pthread.h library in c I implemented both joined and detached threads and assigned them to do the same function to compare them in the later parts of this task

Joined threads code:

```
void
threadsMatrixMulJoinable(
        int numOfThreads) {
    pthread_t th[numOfThreads];
    ThreadARGS args[numOfThreads];// array of a struct used to save more than one argument for the thread routine
    int rows_per_thread = MATRIX_SIZE / numOfThreads;

    for (int i = 0; i < numOfThreads; i++) {
        args[i].startingRow=i*rows_per_thread;
        args[i].endRow=(i+1)*rows_per_thread;
        pthread_create( newthread: &th[i], attr: NULL, start_routine: matrixMulRoutine, arg: (void *)&args[i]);// creating threads and assigning i

    }
    for (int i = 0; i < numOfThreads; i++) {
        if (pthread_join( th: th[i], thread_return: NULL) != 0) {// joining threads
            perror( s: "thread join process failed !\n");
            exit( status: EXIT_FAILURE);
        }
    }
}
```

in the code above is a function that takes a parameter that decides the number of wanted threads, it will create an array of threads corresponding to it and an array of structs that hold the arguments of the thread routine which I will talk about in detail in part 3, in the first loop I divide the assign different parameters to each thread to divide the task between threads and the shared data then I created the threads and assign a routine to it, the second loop is used to synchronize thread execution, with proper errors handling if ever occurred.

Detached threads code:

```
void detachThreadsMul(int numOfThreads){// nearly the same logic for joined

    pthread_t th[numOfThreads];
    ThreadARGS args[numOfThreads];
    int rows_per_thread = MATRIX_SIZE / numOfThreads;
    pthread_attr_t attr;
    pthread_attr_init(&attr);// creating the detached attr
    pthread_attr_setdetachstate(&attr, detachstate: PTHREAD_CREATE_DETACHED);
    for (int i = 0; i < numOfThreads; i++) {
        args[i].startingRow=i*rows_per_thread;
        args[i].endRow=(i+1)*rows_per_thread;
        pthread_create( newthread: &th[i], attr: NULL, start_routine: matrixMulRoutineDetach, arg: (void *)&args[i]);

    }
    usleep( useconds: 100000);// to wait for threads to end excuction

    pthread_attr_destroy(&attr);// destroying the attr after we dont need it


}
```

this function has similar logic to the joined one for most parts as dividing and creating the threads, but in detached threads, we need to create an attr and assign it to be detached then send it as a parameter in the thread creation function, the difference between a detach and joined threads that's a detach threads is a thread that runs in the background independently of the main thread Once a thread is detached, the system is responsible for cleaning up the thread's resources when it completes its execution. So we can't use the join function, this can also cause some problems for example in my code the program terminates before the threads finish their jobs so I had to add a sleep function to my code so it can keep up with it as shown above.

Detached vs joined threads impact on threads:

In the later part of this report, I measured the execution time for both joined and detached threads for different numbers of threads, the relation between the number of threads with throughput is proportional as long as the number of threads is less or equal number of cores, in these case the execution time will be faster which will result to higher throughput, in my case detached threads had less execution time which mean they have higher throughput than joined ones.

Part 3: Performance Measurement and matrix multiplication implementation

To measure performance I implemented matrix multiplication in 3 different ways, the normal approach, processes approach, and thread approach so we can see the execution time for each approach for the same task. I used special two global matrices that are based on my university ID

Matrices creation code:

```
void generateFirstMatrix() {
    srand( seed: time( timer: NULL));

    int myId[] = { [0]: 1, [1]: 2, [2]: 7, [3]: 5, [4]: 6};// my id numbers
    int size = sizeof(myId) / sizeof(myId[0]);// to find size of array
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            FirstMatrix[i][j] = myId[rand() % size];// this code will make sure for e
        }
    }

}

void generateSecondMatrix() {
    srand( seed: time( timer: NULL));

    int myId[] = { [0]: 2, [1]: 4, [2]: 7, [3]: 1, [4]: 6, [5]: 5, [6]: 9};
    int size = sizeof(myId) / sizeof(myId[0]);

    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            SecondMatrix[i][j] = myId[rand() % size];
        }
    }

}
```

these two functions assign random values to my ID number that is stored in an array using the function rand to chose random index of the array every time to insure randomness of the matrices

Normal approach task code:

```
void multiplyMatrices(
        int result[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; ++i) {
        for (int j = 0; j < MATRIX_SIZE; ++j) {
            result[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; ++k) {
                result[i][j] += FirstMatrix[i][k] * SecondMatrix[k][j];
            }
        }
    }
}
```

This is the simplest algorithm for matrix multiplication but it has big o of n^3 run time which is a very big

Process approach task code:

```
void multiplyMatrixPartial(
        int result[MATRIX_SIZE][MATRIX_SIZE], int start_row, int end_row) {
    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            result[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; k++) {
                result[i][j] += FirstMatrix[i][k] * SecondMatrix[k][j];
            }
        }
    }
}
```

This is the function assigned to every process in my code that I showed in the first part, this code has nearly the same algorithm as the normal method but the difference is that it will take a parameter from which row to which row we will do the multiplication,so it will result to the partial result of the task, so we can in the process code divide between processes.

Threads routine task code:

```
void *matrixMulRoutine(void *arg) {// i used this routine for join threadsa only
    ThreadARGS *ARGS = (struct ThreadARGS *) arg;

    for (int i = ARGS->startingRow; i < ARGS->endRow; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            resMat[i][j] = 0;

            for (int k = 0; k < MATRIX_SIZE; k++) {
                resMat[i][j] += FirstMatrix[i][k] * SecondMatrix[k][j];
            }


        }
    }

    pthread_exit( retval: NULL);


}
```

This code has the same algorithm for the process code but in thread routines only hold one void pointer argument so we need to cast the parameters, I used a struct to hold both starting and ending rows parameters because threads can only take one argument.

Measuring method:

To measure execution time I used time.h library, I used two timespec struct from it one to measure the starting time before calling the function and one to measure the ending time after the call then I subtracted the two to get the execution time, the code will look like this:

```
struct timespec start_time, end_time;

clock_gettime( clock_id: CLOCK_MONOTONIC,  tp: &start_time);
multiplyMatricesByProcesses( result: resultMatrix2, numOfProcesses);
clock_gettime( clock_id: CLOCK_MONOTONIC,  tp: &end_time);
double time=(end_time.tv_sec - start_time.tv_sec)+(end_time.tv_nsec - start_time.tv_nsec) / 1e9;
printf( format: "process method time %fs , throuput %f , number of processes %d\n",time,(1/time),numOfProcesses);
```

Can detached thread execution time be measured?

In a multithreaded program, a detached thread is a kind of thread that runs separately from the main program and has its resources automatically released by the system when it finishes, negating the need for explicit joining. When the main program doesn't need to wait for a thread to finish or obtain its exit status, detached threads come in handy. They may have lower management overhead and automatically clean up after themselves, which simplifies resource management.

Measuring the exact execution time of a detached thread within the thread itself can be difficult. This is due to the fact that a detached thread runs independently of the main program and keeps running even after it has terminated. Therefore, trying to measure a detached thread's execution time from within the thread could produce inaccurate/meaningless results.

Because of the independent and asynchronous nature of the detached threads, it is difficult To compute its precise execution time.

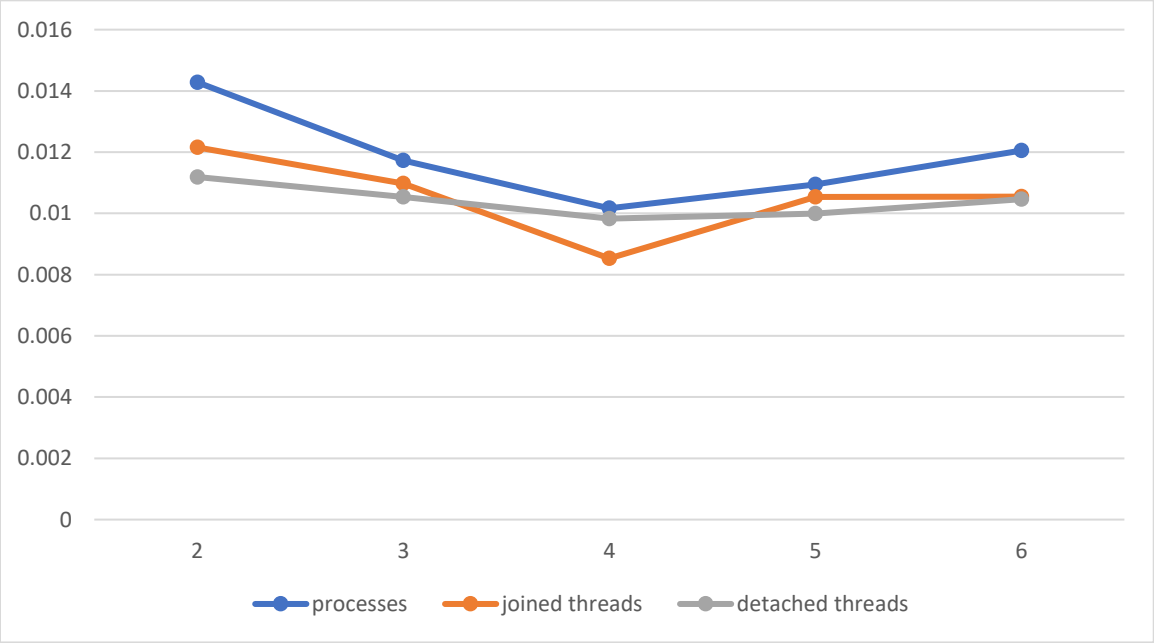
Note that I tried to measure its execution time to compare it with other methods the time will not be 100 percent accurate but it is the closest I can do, the execution time will also have the delay I added to ensure all threads finish before the program finish execution.

Performance comparison:

As I mentioned earlier the normal method has a big o of n^3 run time which is big and through my measuring method the execution time was 0.017388s with a throughput of 57.511655 which is high compared to the process and thraeds methods.

In the case of threads and processes, the execution time will vary corresponding with the number of threads or processes used so I gathered this table that holds different time execution for different numbers of threads/processes:

Method\N0.	2	3	4	5	6
processes	0.014282	0.011733	0.010166	0.010939	0.012055
joined threads	0.012155	0.010973	0.00853	0.010538	0.010548
detached threads	0.011188	0.010538	0.009829	0.009998	0.01046



From the graph above we can conclude that both processes and threads have faster execution times than the normal method with threads being slightly faster than processes and detached threads being the fastest for the majority of the time because detached threads are generally faster in terms of thread creation and termination because the main thread doesn't wait for them, the dividing and parallel execution of the task will lead to a faster execution than threads, threads in our task had the fastest time because my matrices are in the range of its shared memory there is no overhead for larger matrices processes with their IPC methods will have better performance.

Another important thing to notice is that in the graph above after 4 the execution time started to get bigger rather than smaller which means a higher number of threads does not necessarily mean better performance this can happen when the number of cores doesn't correspond to the number of threads/processes so we cant execute all of them at once in parallel, and overhead may occur from context switching and management, also because I implemented this task in a virtual machine I didn't have the access for the full power of my device.