OS TASK 5 - SHARED MEMORY & PTHREADS

Farah Kabesh - 900191706
Karim Sherif - 900191474

ROLES OF TEAM MEMBERS

Farah

- ❑ parallel_sum_mmap function.
- ❑ Excel File.
- ❑ Presentation.

Karim

- ❑ parallel_sum_threads function.
- ❑ Excel File.
- ❑ Presentation.

DESCRIPTION OF PARALLEL_SUM_MMAP

- mmap(): maps between the address space of the process to the memory object represented by the file descriptor.

- *mmap(NULL, sizeof(unsigned long long int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0,0)*

- PROT_READ & PROT_WRITE: data can be read and written.

DESCRIPTION OF PARALLEL_SUM_MMAP

- MAP_SHARED & MAP_ANONYMOUS: shares mapping with all other processes and mapping is not connected to any files.
- File descriptor and offset value set to 0 because we have not mapped any file.

- Children write partial sums to shared memory.
- Parent waits for children to complete before reading the shared memory and computing the total sum.

- To unmap the mapped region (remove a mapping), use munmap().

PSEUDO CODE OF PARALLEL_SUM_MMAP PART 1

```
parallel_sum_mmap(unsigned int n_proc, unsigned int n)

Initialize:

    s=0, partial=0

    start, end=1

    interval = n/n_proc

    *shared = mmap(NULL, sizeof(unsigned long long int),
    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0,0)
```

PSEUDO CODE OF PARALLEL_SUM_MMAP PART 2

```
if shared is equal to MAP_FAILED
    print “mapping failed”
    return 1


for i=1 to i <= n_proc
    start = end
    end = end + interval

    if i is equal to n_proc
        end = n + 1
```

PSEUDO CODE OF PARALLEL_SUM_MMAP PART 3

```
if fork() is equal to 0
    for start to start < end
        partial = partial + start
        shared[i] = partial

    exit
    else
        wait(NULL)
        s = s + shared[i]
```

PSEUDO CODE OF PARALLEL_SUM_MMAP PART 4

```
Initialize:
err = munmap(shared, sizeof(unsigned long long int))

if err not equal to 0
    print Unmapping Failed
    return 1

return s
```

DESCRIPTION OF PARALLEL_SUM_THREADS

- This program is a multithreaded process, where each thread calculates a partial sum and adds it to the total sum.
- pthread_create takes 4 parameters and creates a new thread.
- First argument is pthread_t which is used to store id of the thread
- Second argument is attr is pointer to struct containing the scheduling information.
- Third argument contains the function to be threaded.
- Fourth argument contains the argument for the the threaded function

DESCRIPTION OF PARALLEL_SUM_THREADS

- ❏ pthread_join is a function that lets process to wait for all the threads to finish.
- ❏ It takes the thread_id of the target thread and copies the exit status of that target.
- ❏ Mutexes is used to lock and unlock the final sum so no two threads can change its value at the same time.

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 1

```
Initialize:
    sum_t

    temp

    struct thread_data

        i, s, e, n, num_thread


pthread_mutex_t mutex1
```

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 2

```
void * threaded_sums(void* arg)

Initialize partial=0

if data->i is equal to data->num_thread
    data->e = data->n

for data->s to data->s <= data->e
    partial = partial + data->s
```

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 3

```
pthread_mutex_lock(&mutex1)

sum_t = sum_t + partial

if data->i is equal to data->num_thread
    temp = sum_t

pthread_mutex_unlock(&mutex1)

return (void*) temp
```

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 4

```
parallel_sum_threads(n_thread, n)

Initialize:
start=1, end=interval, ret, *status
interval = n/n_thread

struct thread_data ta[n_thread]
pthread_t threads[n_thread]
```

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 5

```
for j=0 to j < n_thread

    ta[j].i = j + 1
    ta[j].n = N
    ta[j].num_thread = n_thread
    ta[j].s = start
    ta[j].e = end

    ret = pthread_create(&threads[j], NULL, threaded_sums,
    (void *) &ta[j])
```

PSEUDO CODE OF PARALLEL_SUM_THREADS PART 6

```
    if ret not equal to 0
        print "Thread can't be created"

    start = end + 1
    end = end + interval

for j=0 to j < n_thread
    pthread_join(threads[j], &status)

return status
```

HOW TEST 1 WAS DESIGNED & WHY? FIXING N_PROC & N_THREAD

- Machine Name: Intel® Core™ i7-6700 CPU @ 3.40GHz × 8.
- Machine Specifications:
  - Number of cores = 4.
  - Number of threads = 8.

- Therefore n_proc = 3 and n_threads = 2.

- In order to obtain readings:
  - for (n = 1; n < 1000000; n+=5000)

HOW TEST 2 WAS DESIGNED & WHY? FIXING N

- n = 100000.

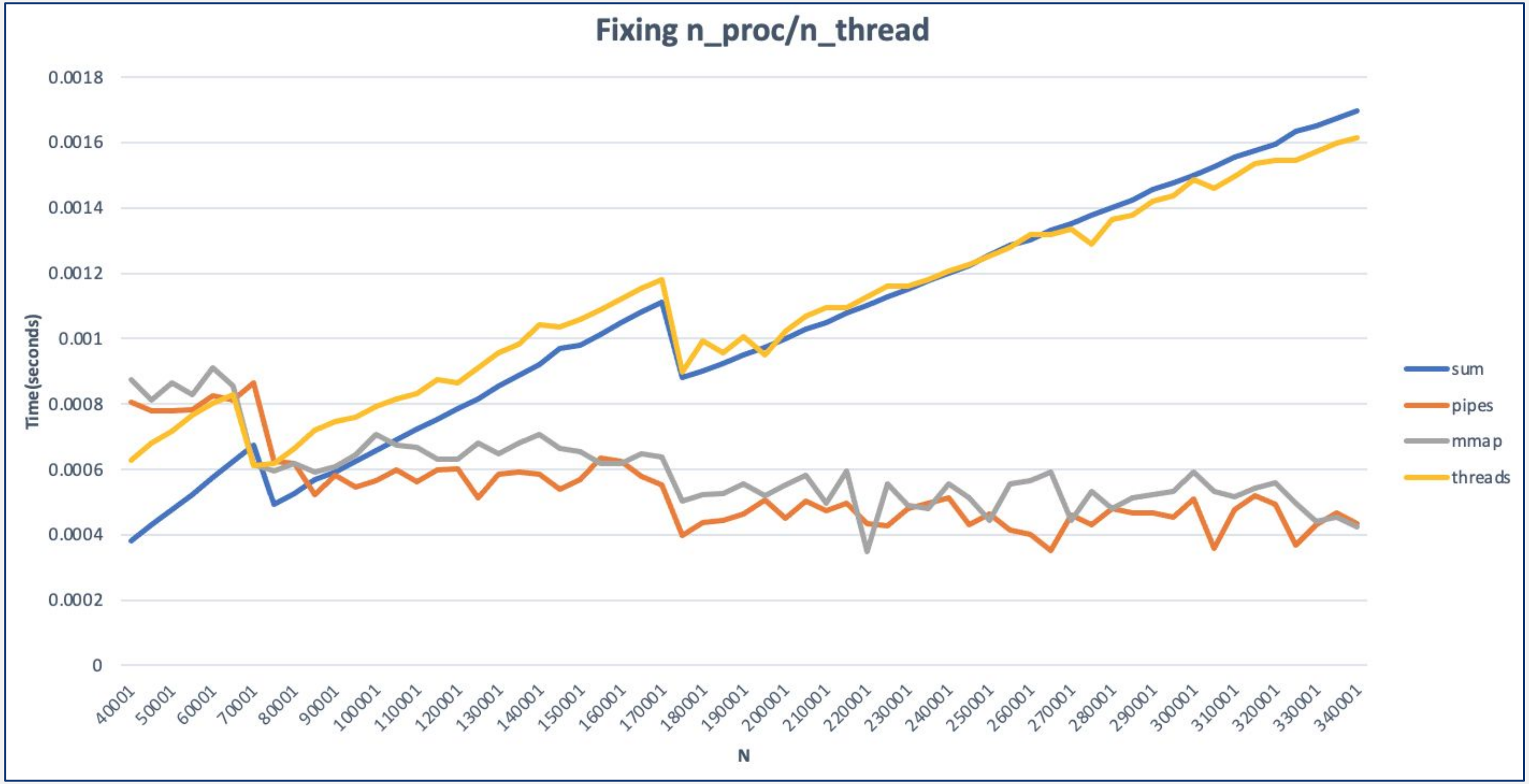
- In order to obtain readings:
  - for (j = 1; j< 150; j++)

Note: j represents number of processes/threads.

QUESTION 1

If we fix *n_proc* and *n_thread* to the same value, then which of the 4 functions take the least time? At which value of N does this behavior persist?

GRAPH 1


Fixing n_proc/n_thread
0.0018
0.0016
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
Time(seconds)
40001
50001
60001
70001
80001
90001
100001
110001
120001
130001
140001
150001
160001
170001
180001
190001
200001
210001
220001
230001
240001
250001
260001
270001
280001
290001
300001
310001
320001
330001
340001
N
sum
pipes
mmap
threads

ANSWER TO QUESTION 1

- Out of the 4 functions, the pipes function takes the least time.

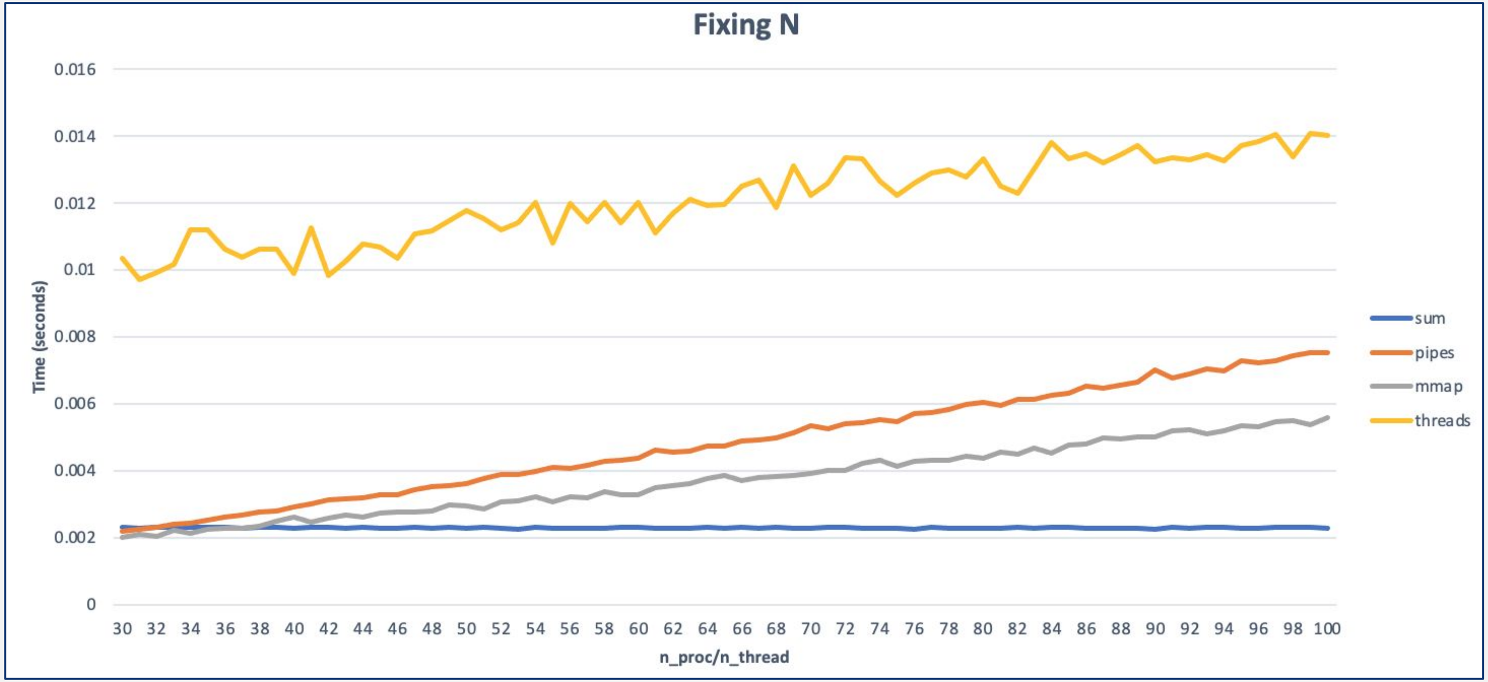
- This behavior persists at the value N = 75001.

- Since n increases and the number of processes/threads is the same, time taken for execution using pipes and shared memory is less than time taken using sum and threads.

QUESTION 2

If we fix *N* to a large number, then which of the 4 functions take the least time? At which value of n_proc/n_thread does this behavior persist?

GRAPH 2



Fixing N
Time (seconds)
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
n_proc/n_thread
sum
pipes
mmap
threads

ANSWER TO QUESTION 2

- ❑ Out of the 4 functions, the sum function takes the least time.

- ❑ This behavior persists at the value n_proc/n_thread = 35

- ❑ As n_proc increases, the overhead of creating new processes increases which leads to more time needed to compute the sums. Therefore, time taken for execution using sum is takes the least time; followed by memory, pipes, and finally threads.