Operating Systems – I (CS3510)

Aim – To write a multi-process and multi-threaded program to develop Fibonacci numbers and compare their performances.

Task 1 : Multithreaded program to develop fibonacci numbers.

This program uses the POSIX api's pthread library to generate threads. They are called POSIX Threads or Pthreads, which is an execution model that exists independently from a language, as well as a parallel execution model. They allow the program to control multiple different flows of work that overlap in time. (here, being the calculation of ith fibonacci number).

Header files :

The program contains the normal <*stdio.h*>, <*stdlib.h*>, etc. as header files, and the <*pthread.h*> header file, which defines the necessary identifiers and functions used in order to create and join pthreads, and the <*unistd.h*> header file which is the code's entry point to various constant, type and function declarations that comprise the POSIX operating system API.

```
#include <unistd.h>
#include <pthread.h>
```

Thread creation

The first five fibonacci numbers are calculated by the parent thread in the main() function, and the rest (n – 5), by creating separate (n – 5) threads, thus the ith thread computes the ith fionacci number independently. Each thread has a set of attributes, including stack size and scheduling information. The **pthread_attr_t attr** declaration represents the attributes for the thread. We set the attributes in the function call **pthread_attr_init(&attr)**.

An opaque attribute object used to set thread attributes is created along with an array of thread identifiers, declared to create (n – 5) threads, as shown :

```
pthread_attr_t attr;                // the set of thread attributes
pthread_attr_init(&attr);           // get the default attributes by passing (initialize attr object)
                                    // the address of thread attribute object in pthread_attr_init function
pthread_create(&fib_th[i], &attr, fib_calculate, (void *) i);
```

pthread_create()

We create an array of threads as well, as shown above, and the program then runs a loop from i = 6 to i = n, to generate the ith fibonacci number, and each time the loop is executed, the program creates a new thread, using

pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*sroutine) (void *), void *arg),

which creates a new thread and makes is executable, whose arguments are the opaque thread identifier (***pthread_t *thread***), thread attribute object (***attr***), the routine (defined as function pointer)

which the thread will execute once its created (***sroutine***), and a single argument (here, the loop counter), passed by reference as a pointer cast of type void.

```
for(i = 6;i <= n;i++){
  pthread_create(&fib_th[i], &attr, fib_calculate, (void *) i);
}
```

pthread_join()

Once the threads are created and they are done finishing their respective tasks, the threads are joined in the next for loop, using **pthread_join(thread[i], NULL),** which waits for the specified thread[i] to exit, it blocks the calling thread (which is parent thread here) until the specified **thread_id** thread terminates. "*Joining*" is one way to accomplish synchronization between threads.

```
for(i = 6;i <= n;i++){
  pthread_join(fib_th[i], NULL);
}
```

The following is the definition of the **sroutine** function, named **fib_calculate** , executed by the thread :

```
int fibonacci(int n){
  int prev = 1, curr = prev, next = prev, i;
  for(i = 3;i <= n;i++){
    next = curr + prev;
    prev = curr;
    curr = next;
  }
  return next;
}

void *fib_calculate(void *counter){
  int index = (long) counter;
  fib_array[index] = fibonacci(index);
  pthread_exit(0);
}
```

The **fib_calculate thread** will complete when it calls the function **pthread_exit(0)**. The result is placed in the **global** *shared* **fib_array** data structure (an array).

Once the task is complete, the parent thread outputs the computed sequence, which was placed in a shared global **fib_array** (since threads share global resources as well).

Task 2 : Multiprocess program to develop fibonacci numbers.

This program creates fibonacci numbes by forking a number of child processes and creating a shared memory object, which both the parent as well as the child processes can share, in order to put the result (computed fibonacci sequence) in it.

Shared Memory Object and mmap

The program creates/opens a **POSIX** shared memory object through **shm_open(const char *name, int oflag, mode_t mode),** which creates and opens a new POSIX shared memory object, which is further used by the child and parent (unrelated since they have their own copies of data) processes to **mmap** the same region of shared memory.

mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset) creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping. Thus, we finally type-cast the mmap into (long *), which makes the long array, shareable, by both the child and parent processes.

Header files :

The program contains the normal header files, and in addtion, the header files which describe the definitions for **fork()** and **wait()** system call, in the <*unistd.h*>, <*wait.h*>file, and for shm_open/mmap/shm_open, in the <*fcntl.h*>, <*sys/mman.h*> and <*sys/types.h*> files.

```
#include <sys/mman.h>
#include <sys/wait.h>
#include <sys/types.h>
```

The shared memory object is created though shm_open as shown below :

```
shared = shm_open("/shared_region", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if(shared == -1){
  printf("Shared memory segment open fail.\n");
  exit(1);
}
```

On successful completion, **shm_open** returns a new file descriptor referring to the shared memory object. The arguments are, **O_CREAT** which means create the shared memory object, if it doesn't exist and **O_RDWR**, which means open the object for read-write access.

mmap the shared memory object created

Once the shared memory object is created, **mmap** creates a new mapping in the virtual address space of the calling process, and on success, returns a pointer to the mapped area, which we type-cast to **(int *)**, so that the array becomes shared between the parent and child processes, as shown :

```
int *fib_array;
fib_array = (int *) malloc(sizeof(int)*(n + 1));
fib_array = mmap(NULL, sizeof(fib_array), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, shared, 0);
if(fib_array == MAP_FAILED){
  printf("mmap fail.\n");
  exit(1);
}
```

Then, the program runs a loop from i = 6 to i = n, forking a new child process every time the loop executes, thus creating (n – 5) child processes in all, where each child process computes the ith

fibonacci number, and places the result in the shared array created above, which already contains the first five fibonacci numbers, computed by the parent process.

The code for the above purpose is shown below (if fork() == 0, then it's a child process) :

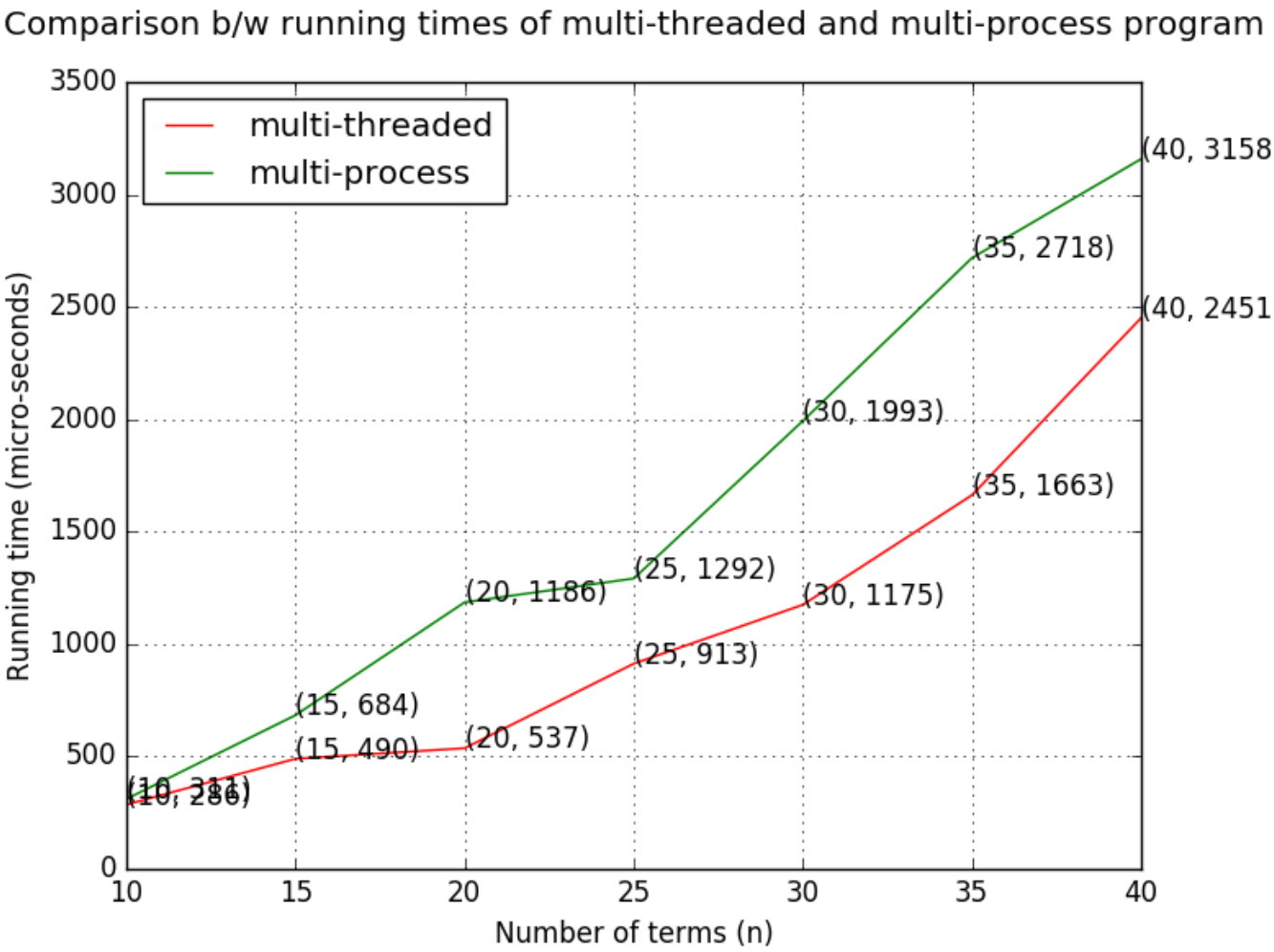
```
for(int i = 6;i <= n;i++){
  pid_t pid;
  if((pid = fork()) == 0){
    fib_array[i] = fibonacci(i);
    exit(0);
  }
  else if(pid < 0){
    cout << "Fork error.\n";
  }
```

Then, once all the child processes are forked, we wait for the child processes to execute and place their result in the shared array as shown (also, a child process will be forked only through a parent process and not through another child process which may then create a "grandchild" process, since the **fork()** is inside the condition of the **if** block, and pid is only zero of a child process) :

```
int status;
for(int i = 6;i < n + 1;++i)
  wait(&status);
```

Once the task is complete, we de-allocate the created shared memory object, using **shm_unlink (char *name)**, and the parent process outputs the computed sequence, which was placed in the shared **fib_array**. The parent waits for the child processes to compute their corresponding fibonacci numbers and their respective exits.

Performance comparison between multi-threaded and multi-process program :



The above plot of running-times of both programs v/s the number of terms clearly indicates that generally, for all values of n, the multi-threaded program takes less CPU time than the multi-process program. This is obvious, cause the overhead of **spawning** a new process is much more than that of a new thread, since in case of threads, very little memory copying is required (just the thread stack), processes are more "**heavier**" to get started since whole process area must be duplicated for the new process copy to start. Once threads are created, each thread independently calculates its fibonacci number and places it in the shared array, whereas in case of processes, only creating n – 5 processes involves more overhead, plus the overhead of creating shared memory, which the child processes share in order to put their computation in it. As the number of terms (n) increases, the multi-process program takes more time than the multi-threaded one, since number of threads/processes is propotional to n, but spawning a new thread in much lighter than creating a new process, as n increases.

CPUs with "hyperthreading" also can run (at least) two threads on a core very rapidly but, not processes (since the "hyperthreaded" threads cannot be using distinct address spaces).

Also, **context-switching** between threads is also much faster than that in multi-processing. Another reason could be that multiple processes *increase memory usage*, since each process has its own

private memory space, but in case of threads, they share resources (global arrays for example), thus no private resource allocation needs to be done for them.

Multi-threading is also better than multi-processing, because it makes complete use of the CPU (can use all the cores (in case of a multi-core processor), by distributing the threads to all the cores, each thread doing a unique task.

Thus, overall, multi-threading is better than multi-processing, and focuses on the point that parallelism is achieved more efficiently in multi-threading.