# REPORT

## Course - Operating System 1

## Instructor - Sathya Peri

## What is Happening in the Multi Process code

Using POSIX shared memory and forking concepts to write a program for finding primes less than the given number  by creating child processes entered by the user and finding primes by dividing number in all child processes. Then writing primes in shared memory from the child processes and accessing that shared memory created for every child process from parent process and writing primes in the file.

## HOW CODE WORKS

1. First a number is taken as input from user upto which prime numbers are to be calculated.
2. Then shared memories are created with read write permissions with **shm_open** function.
3. Then **ftruncate** is used to truncate the created memory to specified size in bytes.
4. Then **mmap** is used to establish a mapping between a process' address space and a file, shared memory objects.
5. Then a child process is forked using **fork** and **parent process** is sent to **wait** state.
6. In child process a pointer is created for shared memory.
7. Then Prime numbers  are calculated by dividing them in different child processes and written to shared memory one by one using **sprintf**.

8. After this when all children processes stops then parent process continues and prints the data in the Shared memory to the file.
9. After completion Shared Memory is unmapped and closed.

# Libraries Used

1. sys/stat - For mode constraints
2. Fcntl - For O_ constraints
3. sys/wait - Wait for process to change state
4. Unistd - Provides access to the POSIX operating system API.
5. sys/mman - Map or Unmap files or devices into memory

# Shm_open

shm_open() creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to mmap(2) the same region of shared memory.

# Ftruncate

The ftruncate() function truncates the file indicated by the open file descriptor file_descriptor to the indicated length. file_descriptor must be a "regular file" that is open for writing. If the file size exceeds length, any extra data is discarded. If the file size is smaller than length, the file is extended and filled with binary zeros to the indicated length.The ftruncate() function does not modify the current file offset for any open file descriptors associated with the file.

# Mmap

mmap() creates a new mapping in the virtual address space of calling process. The starting address for the new mapping is in addr. The length argument specifies the length of mapping. If *addr* is NULL, then the kernel chooses the address at which create the mapping; this is the most portable method of creating a mapping. If *addr* is not NULL, then the kernel takes it as a hint where to place the mapping; on Linux, the mapping will be at a nearby page boundary. The address of the new mapping is as the result of the call.

# Fork

Creates a new process. The new process (the child process) is an exact duplicate of the process that calls fork() (the parent process).

# Wait

Suspends the calling process until any one of its child processes ends. More precisely, wait() suspends the calling process until the system obtains status information on the ended child. If the system already has status information on a completed child process when wait() is called, wait() returns immediately. wait() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

The argument status_ptr points to a location where wait() can store a status value. This status value is zero if the child process explicitly returns zero status. If it is not zero, it can be analyzed with the status analysis macros.

# Sprintf

The sprintf() function formats and stores a series of characters and values in the array buffer. Any argument-list is converted and put out according to the corresponding format specification

in the format-string.The format-string consists of ordinary characters and has the same form and function as the format-string argument for the printf() function.

## What is Happening in the Multi Threaded code

Using global 2-d array and threading concepts to write a program for finding primes less than the given number by creating threads entered by the user and finding primes by dividing numbers equally in all threads. Then writing primes in global array from the child processes and accessing that array created by thread from parent process and writing primes in the file.

## HOW CODE WORKS

1. First a number is taken as input from user upto which prime numbers are to be calculated.
2. Then a 2-d array is created with no of rows as no of threads.
3. Then **pthread_t** is used to get thread id of all the threads.
4. Then **pthread_attr_t** and **pthread_attr_init** is used to initialize all thread attribute to the default value.
5. Then a threads are created using **pthread_create** to our function PrimeCalculator and **pthread_join** is called to suspend the parent thread .
6. Then prime numbers are calculated in the threads and written to its corresponding row in the 2-d array.
7. After this when all threads are complete then parent thread continues and prints the data in the array to the file.

# Libraries Used

## 1. Pthread - **POSIX threads**

**Pthread_attr_init -**

int pthread_attr_init(pthread_attr_t *attr);

Initializes attr with the default thread attributes, whose defaults are:

- Stacksize - Inherited from the STACK runtime option
- Detachstate - Undetached
- Synch - Synchronous
- Weight - Heavy

Using a thread attribute object, you can manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object. All threads are of equal priority.

**Pthread_create -**

int pthread_create(pthread_t *_thread_, pthread_attr_t *_attr_,void *(*_start_routine_)

(void *_arg_), void *_arg_);

Creates a new thread within a process, with attributes defined by the thread attribute object, *attr*, that is created by pthread_attr_init().

If *attr* is NULL, the default attributes are used. See pthread_attr_init() — Initialize a thread attribute object for a description of the thread attributes and their defaults. If the attributes specified by *attr* are changed later, the thread's attributes are not affected.

*pthread_t* is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

The thread is created running *start_routine*, with *arg* as the only argument. If pthread_create() completes successfully, *thread* will contain the ID of the created thread. If it fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

System default for the thread limit in a process is set by MAXTHREADS in the BPXPRMxx parmlib member.

The maximum number of threads is dependent upon the size of the private area below 16M. pthread_create() inspects this address space before creating a new thread. A realistic limit is 200 to 400 threads.

**Pthread_join -**

int pthread_join(pthread_t *thread*, void *******status*);

Allows the calling thread to wait for the ending of the target *thread*.

*pthread_t* is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

*status* contains a pointer to the *status* argument passed by the ending thread as part of pthread_exit(). If the ending thread terminated with a return, *status* contains a pointer to the return value. If the thread was canceled, *status* can be set to -1.

# COMPARISON

Theoretical Comparison of Multi-threading and Multiprocessing
Thread is a sort of mini-process wherein a single process can spawn multiple threads and scheduler schedules them independently.

Creation of thread involves copying the registers and stack. Heap, code and data are not copied and are shared among the processes. Thus creation of thread requires less overhead than creating a new process and hence task can be completed in lesser time.

Moreover thread pools are there which contain created threads and threads upon cancellation fall back into the pool. This prevents latency in creating a new task and hence helps in finishing tasks in lesser time.
On a uniprocessor system , if processor encounters an I-O wait , then CPU will remain idle as it can only schedule now processes(no threads) but in case of multithreading other parallel piece of code will start executing and reduce latency.

Graphical Comparison of Multi-threading and Multiprocessing

From the graph we can see how on increasing the value of n , the two multi threaded and multi processor execution time starts to diverge .

Creating a process consumes time and even exhaust the system resources. However creating threads is economical as threads belonging to the same process share the belongings of that process.

Threads have much lighter weight , much lesser footprint as well as lesser stack size(protects buffer overflows)

# References

Man Pages and OPERATING SYSTEM CONCEPTS by ABRAHAM SILBERSCHATZ