**Module 2 Assignment**

**Q1 & Q2.**

#include <stdio.h>

#include <pthread.h>

#include <time.h>

#include <unistd.h>

int isprime(int x){

int flag=0;

for(int i=2;i<x;i++){

if(x%i==0){

flag=1;

}

}

if(flag){

return 0;

}

else{

return 1;

}

}

void\* t1(void\* args){

static int s=2;

int x=3;

int count=0;

while(count<=100){

if(isprime(x)==1){

s+=x;

count++;

}

x++;

}

return (void\*)&s;

}

void\* t2(void\* args){

for(int i=1;i<=50;i++){

printf("Thread 1 is running\n");

sleep(2);

}

}

void\* t3(void\* args){

for(int i=1;i<=33;i++){

printf("Thread 2 is running\n");

sleep(3);

}

}

int main() {

// Write C code here

struct timespec s,e;

clock\_t start = clock();

pthread\_t tid1,tid2,tid3;

int N=12;

void\* primesum;

pthread\_create(&tid1,NULL,t1,&N);

pthread\_create(&tid2,NULL,t2,NULL);

pthread\_create(&tid3,NULL,t3,NULL);

pthread\_join(tid1,&primesum);

pthread\_join(tid2,NULL);

pthread\_join(tid3,NULL);

printf("%d is the sum of first %d prime numbers",\*((int\*)(primesum)),N);

clock\_t end = clock();

double ans = ((double)(end)-(double)(start))/(CLOCKS\_PER\_SEC);

printf("\n%f sec",ans);

return 0;

}

**OUTPUT:**

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running

25237 is the sum of first 12 prime numbers

0.003325 sec

**Q3.**

PROCESS :

It is the running instance in OS. Eg: Executing Programs, Opening up applications and running.

OS creates memory space for each process with Text segment, Read only part, initialized and uninitialized memory, heap and stack , environment variables and command line arguments space with CPU time and other resources for execution.

Each process will be waiting for its turn to get executed (OS should schedule it for current execution).

Each process will be given its PID and PPID (parent PID) by OS.

Threads are the smallest unit of execution inside process which shares same memory space with the same process except stack and registers.

LIFE CYCLE OF THREAD:

1. New or created: thread is just now created (memory spaces are allocated but not scheduled yet by OS for execution.)

2. Runnable: it is ready to run state for thread waiting for OS scheduling in ready queue.

3. Running: It is the stage where thread is currently in execution and may get terminated or paused if it comes out of the thread function or external process or thread signals it to terminate or it might be waiting to get any events triggered or input to be received from other process.

4. Blocked: It might be waiting to receive input or in shared resource access or for signal

5. Terminated: Finishes its execution or asked to terminate from external process.

CONCURRENCY : It is the ability of system to make progress on multiple simultaneously (logically) even with single core using multitasking or rapid switching between contexts to complete all tasks assigned.

FORK:

It is the system call to create new process from parent process.

It works in C using fork() API under unistd.h header.

Without defining, both processes will do the same action with different PID

Memory will also be shared by both process until child process needs writing. In that case, Copy-On-Write mechanism is used to provide separate copy of memory with separate memory pages and task\_struct for child process to work on.

If fork returns -1. it indicates, child process is not created. If returns 0, then current execution is under child process, if not (returning PID of child process itself), then it is under parent process execution.

It is advisable to wait for child process to get completed under parent process to avoid zombie processes.

SAMPLE PROGRAM:

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

return 1;

}

else if (pid == 0) {

printf("I am the child! My PID is %d\n", getpid());

}

else {

printf("I am the parent! My PID is %d, my child's PID is %d\n", getpid(), pid);

}

return 0;

}

ANOTHER PROGRAM TO SHOW THE EFFECT OF COPY ON WRITE:

int x = 10;

pid\_t pid = fork();

if (pid == 0) {

x = 20;

printf("Child: x = %d\n", x);

} else {

printf("Parent: x = %d\n", x);

}

In above program, variable "x" was being shared in common by both parent and child processes. However, since x is being modified inside child's execution, there will be two copies of "x" for parent and child processes to use and that is reflected in both print statements present there.

SAMPLE PROGRAM - 3:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

pid = fork();

if (pid < 0) {

perror("Fork failed");

return 1;

}

else if (pid == 0) {

printf("Child Process: PID = %d\n", getpid());

printf("Child is counting from 1 to 5:\n");

for (int i = 1; i <= 5; i++) {

printf("Child: %d\n", i);

sleep(1);

}

printf("Child Process: Finished counting.\n");

exit(0);

}

else {

printf("Parent Process: PID = %d\n", getpid());

printf("Parent: Waiting for child to finish...\n");

int status;

wait(&status);

if (WIFEXITED(status)) {

printf("Parent: Child exited with status %d\n", WEXITSTATUS(status));

} else {

printf("Parent: Child exited abnormally.\n");

}

printf("Parent Process: Done.\n");

}

return 0;

}

In above program, parent and child processes are doing different activity and parent process waits until child process gets terminated. Status of termination will be identified by WIFEXITED() function which will be non zero iff proper termination and WEXITSTATUS() API will work iff normal termination happens.

SIGNAL:

1. It is the software interrupt that is delivered to a process to signify that some event has happened and expects some response from it.

2. Response for signal is called signal handling and it is essential to gracefully clean up resources and to get terminated, to handle exceptional situations

SAMPLE PROGRAM TO HANDLE CTRL+C TERMINATION INTERRUPT:

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

#include <unistd.h>

void handle\_sigint(int sig) {

printf("\nCaught signal %d (SIGINT). Exiting safely!\n", sig);

exit(0);

}

int main() {

signal(SIGINT, handle\_sigint);

while (1) {

printf("Running... Press Ctrl+C to stop\n");

sleep(1);

}

return 0;

}

Here, signal functions takes up two arguments -> Signal to handle, handler to handle

SAMPLE PROGRAM TO HANDLE ONE OF THE USER DEFINED SIGNALS :

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

// Handler for SIGUSR1

void handle\_sigusr1(int sig) {

printf("Received SIGUSR1 (signal number %d)\n", sig);

}

int main() {

// Print process ID so we can send signal manually

printf("Process PID: %d\n", getpid());

// Register the signal handler

signal(SIGUSR1, handle\_sigusr1);

// Wait indefinitely for signals

while (1) {

printf("Waiting for SIGUSR1...\n");

sleep(5);

}

return 0;

}

Here, SIGUSR1 is one of the user defined signals that is signalled by "kill -USR1 <pid>" (To know PID, getpid() is invoked in this program, so after executing this program and knowing PID, kill command can be issued either from terminal or from another C program using kill(pid,SIGUSR1))

LIST OF STANDARD SIGNALS:

1. SIGINT- 2 (interrupt from keyboard) (Ctrl+c to invoke)

2. SIGTERM - 15 (termination request and it will be processed) (kill <pid> to invoke)

3. SIGKILL - 9 (Kill signal thus process should terminate immediately without concentrating on cleaning up the resources) (kill -9 <pid>)

4. SIGSTOP - 19 (To pause the process) (kill -19 <pid> or CTRL + Z)

5. SIGCONT - 18 (To resume the paused process) (kill -18 <pid>)

6. SIGSEGV - 11 (Invalid memory access - terminates process) (dereferencing NULL or freed memory)

7. SIGCHLD - 17 (automatically generated whenever child process is terminated and ignored by default by parent process)

8. SIGUSR1,2 - 10,12 (User defined signals with custom handling logic) (kill -USR1 <pid>)

KERNEL CRASH :

1. Kernel is the main part or core of OS that handles processes, files, memory, devices or resources etc.

2. Kernel crash happens whenever something goes unexpected like module programming bugs, invalid memory access, Hardware problem etc, deadlocks etc.

3. In the event of kernel crash, system may reboot multiple times, or go off or kill that particular process that offends etc.

4. Null pointer dereference -> Here, In kernel code, If a pointer is declared NULL and if it is accessed, then it leads to Oops or panic.

5. Kernel Oops is like minor accident which is not fatal but logs the error (writing to invalid memory or division by zero or writing in protected memory region)

6. Kernel panic -> it is full system crash that is unrecoverable issue (Eg: No memory left, stack overflow, buffer overflow on critical systems, hardware failure etc. ) -> this leads to system going off.

7. Segmentation fault -> Trying to access the memory which it doesn't own. Eg. returning local variable to main. Doing wrong pointer arithmetics.

8. Deadlock -> It is the condition when two threads hold one resource already and waits for other resource to release already holded resource. Thus, no process will proceed further leading to CPU consumption (Heavier risk in case of spinlocks since sleep is not allowed)

9. Hardware failure -> Overheating CPU or failing RAM or power supply issues lead to this error and it will lead to raising machine check exception.

10. Invalid opcode -> when kernel receives a command which it cant understand -> due to buggy code or memory corruption

11. Watchdog timer expiry -> Periodically watchdog packets will be sent and responded by kernel to advertise its presence. However, if response is not obtained for significant time interval, system will be rebooted.

Kernel crashes can be debugged using Kernel module logs or memory snapshots using kdump like crash dump tools which activates light weight kernel in case of crash in memory to save current state of memory for future debugging.

TIME COMPLEXITY:

1. It is the mathematical abstraction to evaluate how fast or slow the algorithm is considering only the input size irrespective of hardware.

2. Essential in time critical embedded programming or big data analytics etc.

3. It uses asymptotic notations that expresses growth trends by ignoring constants and lower order terms

Big O (f(n)) -> takes worst case

Omega (f(n)) -> takes best or easy case

theta (f(n)) -> takes average or realistic case

small o (f(n)) -> strictly smaller than f(n)

small omega (f(n)) -> strictly greater than f(n)

Common examples:

O(1)-> constant time access Eg: Array with indexing

O(log n) -> for binary search since it divides its search area into half every iteration

O(n) -> linear time Eg: Linear search (since it takes worst case)

O(n log n) -> Merge sort or quick sort

O(n squared) -> for matrix traversal or bubble sort or selection sort

O(2 ^ n) -> Recursive Fibonacci

It considers each operation like arithmetics, swapping, comparison into play for calculating time complexity but focussing on dominant terms.

MUTEX / SPINLOCK:

1. A mutex is a synchronization primitive used to protect shared resources (critical sections) from concurrent access by multiple threads.

2. Only one thread can hold the mutex at a time. Others must wait.

3. Without Mutex : When multiple threads access and modify shared data simultaneously, unpredictable behavior occurs.

4. Thus mutex guarantees that at any given time, only one thread is in critical section and all others will be in blocked or sleep mode until lock is released by holding thread.

5. Recursive mutex is possible -> That is single thread can lock the same resource multiple times but should unlock the same number of times for other threads to utilize.

6. Used in case of protecting shared variables, database, files , buffers etc.

7. Leads to deadlock if one thread holds resource-A and waits for B to release A and opposite in case of another thread -> no further steps will be proceeded consuming CPU unnecessarily.

8. OS should consider priority in this case when low priority thread holds any critical resource that high priority task later requests for. In this case, OS should do context switching.

9. However, if short duration critical sessions are guaranteed, then mutex might not be the good choice since it introduces thread blocking or sleeping and context switching overhead.

10. Mutex is simply a structure in implementation with lock state, owner thread ID, queue of waiters, optional attributes.

11. Spinlock is similar to mutex but it introduces spinning state for threads if resources are locked instead of blocking or sleeping.

12. It is more advisable in case of locks that are held for very short durations since overhead of sleeping and waking is worse than just looping a bit.

13. It is preferred to disable preemption or handling interrupts except critical during spinlock sessions since other threads continuously spins for access instead of waiting in the queue for their turn.

In any synchronization primitives, Atomicity is essential since it guarantees that the requested action happens or doesn't start at all instead of putting it in race condition with other thread risking the access of resources and crashes.