| Practical 9 (A) | |
| --- | --- |
| **Aim:** | Compare the Execution of single Process with threads execution. |
| **THREAD**  Threading is a light weight process which shares all the section of the process except for the  stack. A process can have multiple threads. | |
| **Fork Vs Thread**  While threads can execute in parallel with same context. Also, memory and other resources are  shared between the threads causing less overhead. A thread process is considered a sibling while  a forked process is considered a child. Also, threads are known as light-weight processes as they  don't have any overhead as compared to processes (as it doesn't issue any separate command for  creating completely new virtual address space). A single process can have multiple threads. For  all threads of any process, communication between them is direct. While process needs some  interprocess communication mechanism to talk to other processes. Thought, threads seem to be  more useful for any reason, do note that changes in any thread may lead to changes in other  threads of the same process. While, changes in child processes is independent as parent process  has its own execution copy. | |
| **Code:** | #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #define NO\_OF\_THREADS 5  #define ELEMENTS 100  static int arr[ELEMENTS];  static int sum;  int retval[5] = {  1,  2,  3,  4,  5  };  int j = 0;  void \* thread\_sum(void \* arg) {  int i;  int \* current\_thread\_data = (int \* ) arg;  printf("-> Current thread no is : %d\n", current\_thread\_data[j]);  int end = (current\_thread\_data[j]) \* (ELEMENTS / NO\_OF\_THREADS);  int start = end - (ELEMENTS / NO\_OF\_THREADS);  printf("-> Here we will calculate the sum of %d to %d\n", arr[start], arr[end - 1]);  int current\_thread\_sum = 0;  for (i = start; i < end; i++) {  current\_thread\_sum += arr[i];  }  sum += current\_thread\_sum;  printf("-> current\_thread\_sum : %d\n", current\_thread\_sum);  pthread\_exit( & retval[j]);  j++;  return NULL;  }  int main() {  int i, thread\_no = 1;  for (i = 0; i < ELEMENTS; i++)  arr[i] = i + 1;  pthread\_t id[NO\_OF\_THREADS];  int data\_arr[NO\_OF\_THREADS];  printf("-> Creating %d number of threads...\n", NO\_OF\_THREADS);  for (thread\_no = 1; thread\_no <= NO\_OF\_THREADS; thread\_no++) {  data\_arr[thread\_no - 1] = thread\_no;  pthread\_create( & id[thread\_no - 1], NULL, thread\_sum, & data\_arr[thread\_no - 1]);  }  for (i = 1; i <= NO\_OF\_THREADS; i++)  pthread\_join(id[i - 1], NULL);  printf("-> Total sum: %d\n", sum);  return 0;  } |
| Output |  |
| **Questions:** | Differentiate between fork and thread. |
| Answer: | Forking and threading are both mechanisms for achieving concurrent execution in computer programs, but they differ in fundamental ways. Forking involves creating a new independent process that duplicates the entire address space of the parent process, resulting in separate memory and execution contexts. Threads, on the other hand, are lighter-weight units of execution that share the same memory space as the parent process but have their own stack and program counter, allowing them to execute concurrently. Threads are generally more efficient for achieving parallelism within a single process, as they have lower overhead compared to forking, which involves duplicating resources. However, forking is more suitable for achieving true process-level isolation when strong separation between tasks is required. |

| Practical 9 (B) | |
| --- | --- |
| **Aim:** | Perform Thread synchronization using counting semaphores and  mutual exclusion using mutex |
| Code | **B) Perform Thread synchronization using counting semaphores**  #include<pthread.h>  #include<stdio.h>  #include<semaphore.h>  #include<unistd.h>  void \*fun1();  void \*fun2();  int shared=1; //shared variable  sem\_t s; //semaphore variable  int main()  {  sem\_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is  number of processes sharing semaphore, 3rd argument is the initial value of semaphore  variable  pthread\_t thread1, thread2;  pthread\_create(&thread1, NULL, fun1, NULL);  pthread\_create(&thread2, NULL, fun2, NULL);  pthread\_join(thread1, NULL);  pthread\_join(thread2,NULL);  printf("Final value of shared is %d\n",shared); //prints the last updated value of shared  variable  }  void \*fun1()  {  int x;  sem\_wait(&s); //executes wait operation on s  x=shared;//thread1 reads value of shared variable  printf("Thread1 reads the value as %d\n",x);  x++; //thread1 increments its value  printf("Local updation by Thread1: %d\n",x);  sleep(1); //thread1 is preempted by thread 2  shared=x; //thread one updates the value of shared variable  printf("Value of shared variable updated by Thread1 is: %d\n",shared);  sem\_post(&s);  }  void \*fun2()  {  int y;  sem\_wait(&s);  y=shared;//thread2 reads value of shared variable  printf("Thread2 reads the value as %d\n",y);  y--; //thread2 increments its value  printf("Local updation by Thread2: %d\n",y);  sleep(1); //thread2 is preempted by thread 1  shared=y; //thread2 updates the value of shared variable  printf("Value of shared variable updated by Thread2 is: %d\n",shared);  sem\_post(&s);  } |
| Output |  |
|  | **B) Perform Thread synchronization using mutex**  An example code to study synchronization problems  #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  pthread\_t tid[2];  int counter;  void\* trythis(void\* arg)  {  unsigned long i = 0;  counter +=7 1;  printf("\n Job %d has started\n", counter);  for (i = 0; i < (0xFFFFFFFF); i++) ;  printf("\n Job %d has finished\n", counter);  return NULL;  }  int main(void)  {  int i = 0;  int error;  while (i < 2) {  error = pthread\_create(&(tid[i]), NULL, &trythis, NULL);  if (error != 0)  printf("\nThread can't be created : [%s]", strerror(error));  i++;  }  pthread\_join(tid[0], NULL);  pthread\_join(tid[1], NULL);  return 0;  } |
| Output |  |

|  | Practical 10 |
| --- | --- |
| **Aim:** | Implement inter process communication (IPC) using PIPEs and FIFOs |
| **Code:** | **Using Pipes:**  #include<stdio.h>  #include<unistd.h>  int main() {     int pipefds[2];     int returnstatus;     char writemessages[2][20]={"Hi", "Hello"};     char readmessage[20];     returnstatus = pipe(pipefds);       if (returnstatus == -1) {        printf("Unable to create pipe\n");        return 1;     }       printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);     write(pipefds[1], writemessages[0], sizeof(writemessages[0]));     read(pipefds[0], readmessage, sizeof(readmessage));     printf("Reading from pipe – Message 1 is %s\n", readmessage);     printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);     write(pipefds[1], writemessages[1], sizeof(writemessages[0]));     read(pipefds[0], readmessage, sizeof(readmessage));     printf("Reading from pipe – Message 2 is %s\n", readmessage);     return 0;  } |
| **Output:** |  |
|  | **Using FIFO**  #include <iostream>  #include <cstdlib>  #include <cstring>  #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  int main() {  *const* char\* fifo\_path = "myfifo";    *// Create the FIFO if it doesn't exist*      if (mkfifo(fifo\_path, 0666) == -1) {          perror("mkfifo");          exit(EXIT\_FAILURE);      }        pid\_t pid = fork();        if (pid == -1) {          perror("fork");          exit(EXIT\_FAILURE);      }        if (pid == 0) { *// Child process*          int fifo\_read = open(fifo\_path, O\_RDONLY);          if (fifo\_read == -1) {              perror("open");              exit(EXIT\_FAILURE);          }          char buffer[100];          ssize\_t bytes\_read = read(fifo\_read, buffer, sizeof(buffer));          if (bytes\_read == -1) {              perror("read");              exit(EXIT\_FAILURE);          }          std::cout << "Child received: " << std::string(buffer, bytes\_read) << std::endl;          close(fifo\_read);      } else { *// Parent process*          int fifo\_write = open(fifo\_path, O\_WRONLY);          if (fifo\_write == -1) {              perror("open");              exit(EXIT\_FAILURE);          }          std::string message = "Hello from parent";          ssize\_t bytes\_written = write(fifo\_write, message.c\_str(), message.size());          if (bytes\_written == -1) {              perror("write");              exit(EXIT\_FAILURE);          }          close(fifo\_write);      }    *// Remove the FIFO file*      if (unlink(fifo\_path) == -1) {          perror("unlink");          exit(EXIT\_FAILURE);      }        return 0;  } |
| **Output:** |  |