**OS PRACTICAL - 6, 7, 8, 9, 10**

**6. Implementation of Banker’s Algorithm.**

|  |  |
| --- | --- |
|  | **IMPLEMENTATION DETAILS:** |
|  | **INPUT/s:** |
|  | Basic input required to implement the Banker's Algorithm: |
|  | (i) Available  (ii) Max  (iii) Allocation |
|  | **STEPS TO PERFORM:** |
|  | (i) Perform Banker's algorithm when a request for R is made.  (ii) Compute Need[i,j] = Max[i,j] - Allocation[i,j].  (iii) Update accordingly.  Once the resources are *allocated*, check to see if the system state is safe. If unsafe, the process must wait and the old resource-allocated state is restored.\ |
|  | **OUTPUT/s:** |
|  | Detection process specifies if a deadlock is present in a system with listed processes and their needs or not. |

**CODE**

**#include <stdio.h>**

**int main()**

**{**

**// P0, P1, P2, P3, P4 are the Process names here**

**int n, m, i, j, k;**

**n = 5; // Number of processes**

**m = 3; // Number of resources**

**int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix**

**{ 2, 0, 0 }, // P1**

**{ 3, 0, 2 }, // P2**

**{ 2, 1, 1 }, // P3**

**{ 0, 0, 2 } }; // P4**

**int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix**

**{ 3, 2, 2 }, // P1**

**{ 9, 0, 2 }, // P2**

**{ 2, 2, 2 }, // P3**

**{ 4, 3, 3 } }; // P4**

**int avail[3] = { 3, 3, 2 }; // Available Resources**

**int f[n], ans[n], ind = 0;**

**for (k = 0; k < n; k++) {**

**f[k] = 0;**

**}**

**int need[n][m];**

**for (i = 0; i < n; i++) {**

**for (j = 0; j < m; j++)**

**need[i][j] = max[i][j] - alloc[i][j];**

**}**

**int y = 0;**

**for (k = 0; k < 5; k++) {**

**for (i = 0; i < n; i++) {**

**if (f[i] == 0) {**

**int flag = 0;**

**for (j = 0; j < m; j++) {**

**if (need[i][j] > avail[j]){**

**flag = 1;**

**break;**

**}**

**}**

**if (flag == 0) {**

**ans[ind++] = i;**

**for (y = 0; y < m; y++)**

**avail[y] += alloc[i][y];**

**f[i] = 1;**

**}**

**}**

**}**

**}**

**printf("Following is the SAFE Sequence\n");**

**for (i = 0; i < n - 1; i++)**

**printf(" P%d ->", ans[i]);**

**printf(" P%d", ans[n - 1]);**

**return (0);**

**}**

**7.Conversion of resource allocation graph (RAG) to wait-for-graph (WFG) for each type of method used for storing graph**

|  |  |
| --- | --- |
|  | **IMPLEMENTATION DETAILS:** |
|  | One such deadlock detection algorithm makes use of a wait-for graph to track which other processes a process is currently blocking on. In a wait-for graph, processes are represented as nodes, and an edge from process Pi to Pj implies Pj is holding a resource that Pi needs and thus Piis waiting for Pjto release its lock on that resource. There may be processes waiting for more than a single resource to become available. Graph cycles imply the possibility of a deadlock. |
|  | **INPUT/s:**  Output of experiment 7 as Resource allocation graph (RAG) through adjacency matrices/list. |
|  | **STEPS TO PERFORM:** |
|  | (i)Identify the waiting processes in the RAG.  (ii)Accordingly, draw a Wait-for graph for the given RAG.  (iii) To draw it as graphical representation, we introduce graphics.h and work in graphics mode.  (iv)Geometric images are entered by entering graphics, providing parameters and closing graphics.  (v) We now identify circular chain of dependency (i.e., appearance of loops in the graph) |
|  | **OUTPUT/s:** |
|  | (i) The wait-for-graph(graphical representation).  (ii) Also, check presence of loop to detect if loop is present |

**8.Write a program where the parent process takes the average of the odd numbers and the child process will take the average of even numbers present in a given Aadhar number of a person. Use FORK and JOIN construct.**

|  |
| --- |
| **#include <iostream>**  **#include <unistd.h>**  **using namespace std;**    **// Driver code**  **int main()**  **{**  **// array represents aadhar number**  **int a[10] = { 4, 1, 8, 0, 9,, 7, 6, 3, 3, 8, 2, 9 };**  **int sumOdd = 0, sumEven = 0, n, i;**  **n = fork();**    **// Checking if n is not 0**  **if (n > 0) {**  **for (i = 0; i < 10; i++) {**  **if (a[i] % 2 == 0)**  **sumEven = sumEven + a[i];**  **}**  **cout << "Parent process \n";**  **cout << "Sum of even no. is " << sumEven << endl;**  **}**    **// If n is 0 i.e. we are in child process**  **else {**  **for (i = 0; i < 10; i++) {**  **if (a[i] % 2 != 0)**  **sumOdd = sumOdd + a[i];**  **}**  **cout << "Child process \n";**  **cout << "\nSum of odd no. is " << sumOdd << endl;**  **}**  **return 0;**  **}** |

**9. Write a program where the parent process finds additive primes and the child process finds circular prime for a given prime list array. Use FORK and JOIN construct.**

**#include <iostream>**

**#include <unistd.h>**

**using namespace std;**

**void sieve(int maxEle, int prime[])**

**{**

**prime[0] = prime[1] = 1;**

**for (int i = 2; i \* i <= maxEle; i++) {**

**if (!prime[i]) {**

**for (int j = 2 \* i; j <= maxEle; j += i)**

**prime[j] = 1;**

**}**

**}**

**}**

**// Function to return the sum of digits**

**int digitSum(int n)**

**{**

**int sum = 0;**

**while (n) {**

**sum += n % 10;**

**n = n / 10;**

**}**

**return sum;**

**}**

**// Function to print additive primes**

**void printAdditivePrime(int arr[], int n)**

**{**

**int maxEle = \*max\_element(arr, arr + n);**

**int prime[maxEle + 1];**

**memset(prime, 0, sizeof(prime));**

**sieve(maxEle, prime);**

**for (int i = 0; i < n; i++) {**

**// If the number is prime**

**if (prime[arr[i]] == 0) {**

**int sum = digitSum(arr[i]);**

**// Check if it's digit sum is prime**

**if (prime[sum] == 0)**

**cout << arr[i] << " ";**

**}**

**}**

**}**

**bool isPrime(int n)**

**{**

**// Corner cases**

**if (n <= 1)**

**return false;**

**if (n <= 3)**

**return true;**

**// This is checked so that we can skip**

**// middle five numbers in below loop**

**if (n % 2 == 0 || n % 3 == 0)**

**return false;**

**for (int i = 5; i \* i <= n; i = i + 6)**

**if (n % i == 0 || n % (i + 2) == 0)**

**return false;**

**return true;**

**}**

**// Function to check if the number is circular**

**// prime or not.**

**bool checkCircular(int N)**

**{**

**// Count digits.**

**int count = 0, temp = N;**

**while (temp) {**

**count++;**

**temp /= 10;**

**}**

**int num = N;**

**while (isPrime(num)) {**

**// Following three lines generate the next**

**// circular permutation of a number. We move**

**// last digit to first position.**

**int rem = num % 10;**

**int div = num / 10;**

**num = (pow(10, count - 1)) \* rem + div;**

**// If all the permutations are checked and**

**// we obtain original number exit from loop.**

**if (num == N)**

**return true;**

**}**

**return false;**

**}**

**// Driver code**

**int main()**

**{**

**// array represents prime numbers**

**int a[10] = { 3, 5, 7, 9, 11, 13, 17, 19, 23 };**

**n = fork();**

**// Checking if n is not 0**

**if (n > 0) {**

**cout << "Parent process \n";**

**for (i = 0; i < 10; i++) {**

**if (checkCircular(a[i]))**

**cout <<< “Circular Prime”;**

**}**

**}**

**// If n is 0 i.e. we are in child process**

**else {**

**cout << "Child process \n";**

**printAdditivePrime(a, 9);**

**}**

**return 0;**

**}**

**10.Contiguous Allocation Techniques-Implementation of Contiguous allocation techniques:**

**(a) Worst-Fit**

**(b) Best-Fit**

**(c) First-Fit**

|  |
| --- |
| **IMPLEMENTATION DETAILS:** |
| **INPUT/s:** |
| (i) Free space list of blocks from system |
| (ii) List processes and files from the system |
| **STEPS TO PERFORM**: |
| (i) We consider the same free space list and files/processes as created in experiment 3 for our system.  (ii) Implement the above mentioned three contiguous allocation techniques.Also, the free space list is updated from the free blocks left out after performing allocation. |
| **(a) Worst-fit**: In the worst fit technique the largest available block/partition which will hold the page is selected. Blocks are sorted according to their size in descending order.  **(b) Best-fit**: Best-fit is one of the optimal techniques in which page is stored in the block/partition which is large enough to hold it. Blocks are sorted according to their size in ascending order.  **(c) First-fit**: In the first-fit technique page is stored in the block which is encountered first that is big enough to hold it.  (iii) Also, the free space list is updated from the free blocks left out after performing allocation. |
| **OUTPUT/s:** |
| Processes and files allocated to free blocks. List of processes and files which are not allocated memory. The remaining free space list left out after performing allocation. |
|  |

**-------------------------------------------------------------------------------------------------------------------------------------**