** FORK **

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
   pid_t pid;
   // Create a child process
   pid = fork();
   if (pid < 0) {
      // Fork failed
      fprintf(stderr, "Fork Failed\n");
      return 1;
   } else if (pid == 0) {
      // Child process
      printf("Child Process: PID = %d\n", getpid());
      printf("Child Process: Hello World\n");
   } else {
      // Parent process
      printf("Parent Process: PID = %d\n", getpid());
      printf("Parent Process: Hi\n");
      // Wait for the child process to complete
      wait(NULL);
   }
   return 0;
}
```

*** EXE() ***

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
   pid_t pid;
   pid = fork();
   if (pid < 0) {
      fprintf(stderr, "Fork Failed\n");
      return 1;
   } else if (pid == 0) {
      printf("Child Process: PID = %d\n", getpid());
      execlp("/bin/echo", "echo", "Hello, World!", NULL);
      fprintf(stderr, "Exec Failed\n");
      return 1;
   } else {
      printf("Parent Process: PID = %d\n", getpid());
      wait(NULL);
     printf("Parent Process: Child terminated and control is back to parent.\n");
  }
   return 0;
}
```

Orphan fork() sleep()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
   pid_t pid;
   // Create a child process
   pid = fork();
   if (pid < 0) {
      // Fork failed
      fprintf(stderr, "Fork Failed\n");
      return 1;
   } else if (pid == 0) {
      printf("Child Process: PID = %d, PPID = %d\n", getpid(), getppid());
      sleep(5);
      // Display the child process ID and parent process ID after waking up
      printf("Child Process (Orphan): PID = %d, PPID = %d\n", getpid(), getppid());
   } else {
      // Parent process
      printf("Parent Process: PID = %d\n", getpid());
   }
   return 0;
}
```

Write a program that demonstrates the use of nice () system call. After a child process is started using fork (), assign higher priority to the child using nice () system call.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main() {
   pid_t pid;
   pid = fork();
   if (pid < 0)
      fprintf(stderr, "Fork Failed\n");
      return 1;
   } else if (pid == 0) {
      printf("Child Process: PID = %d\n", getpid());
      int priority = nice(-10);
 if (priority == -1) {
         perror("nice");
         exit(EXIT_FAILURE);
      }
      printf("Child Process: New priority is %d\n", priority);
      for (int i = 0; i < 5; ++i) {
         printf("Child working...\n");
         sleep(1);
      }
   } else {
```

```
        printf("Parent Process: PID = %d\n", getpid());
        wait(NULL);
        printf("Parent Process: Child completed\n");
    }
    return 0;
}
```

Q+==Write a program to find the execution time taken for execution of a given set of instructions (use clock() function)

```
#include <stdio.h>
#include <time.h>
int main() {
    clock_t start, end;
    double cpu_time_used;
    // Start the clock
    start = clock();
    // Instructions to be timed
    for(int i = 0; i < 1000000; i++) {
        // Dummy operations
        int x = i * i;
    }
    end = clock();
    // Calculate the time taken
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", cpu_time_used);
  return 0;
}
```

Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();
    if (pid < 0) {
        // Fork failed
        fprintf(stderr, "Fork Failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child Process: PID = %d\n", getpid());
        // Execute the "ls" command using execl()
        execl("/bin/ls", "ls", NULL);
        // If execl() fails
        perror("execl");
        return 1;
    } else {
        // Parent process
        printf("Parent Process: PID = %d\n", getpid());
```

```
        // Parent process goes to sleep
        sleep(10);
        // After waking up, wait for the child process to complete
        wait(NULL);
        printf("Parent Process: Child completed\n");
    }
    return 0;
}
```

1 Write a C program to accept the number of process and resources and find the need matrix content and display it.

```
#include <stdio.h>
void calculateNeed(int need[][4], int max[][4], int alloc[][4], int np, int nr) {
    for (int i = 0; i < np; i++) {
        for (int j = 0; j < nr; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
}

void displayMatrix(int matrix[][4], int np, int nr) {
    for (int i = 0; i < np; i++) {
        for (int j = 0; j < nr; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
```

```
}

int main() {
    int np, nr;
    printf("Enter the number of processes: ");
    scanf("%d", &np);
    printf("Enter the number of resources: ");
    scanf("%d", &nr);
    int max[4][4], alloc[4][4], need[4][4];
    printf("Enter the Max matrix:\n");
    for (int i = 0; i < np; i++) {
        for (int j = 0; j < nr; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the Allocation matrix:\n");
    for (int i = 0; i < np; i++) {
        for (int j = 0; j < nr; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }

    // Calculate the Need matrix
    calculateNeed(need, max, alloc, np, nr);

    printf("The Need matrix is:\n");
    displayMatrix(need, np, nr);
    return 0;
}
```

1 Write the program to calculate minimum number of resources needed to avoid deadlock

```
#include <stdio.h>
#include <stdbool.h>
int main() {
   int n, m;
   printf("Enter number of processes: ");
   scanf("%d", &n);
   printf("Enter number of resource types: ");
   scanf("%d", &m);
   int allocation[n][m];
   int max[n][m];
   int available[m];
   int need[n][m];
   printf("Enter the Allocation Matrix:\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         scanf("%d", &allocation[i][j]);
      }
   }
   printf("Enter the Max Matrix:\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         scanf("%d", &max[i][j]);
      }
   }
   printf("Enter the Available Resources:\n");
   for (int j = 0; j < m; j++) {
      scanf("%d", &available[j]);
   }
```

```
// Calculate the Need Matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
// Function to check if the system is in a safe state
bool isSafe(int n, int m, int processes[], int available[], int max[][m], int allocation[][m]) {
    int work[m];
    bool finish[n];
    for (int i = 0; i < m; i++) {
        work[i] = available[i];
    }
    for (int i = 0; i < n; i++) {
        finish[i] = false;
    }
    int safeSequence[n];
    int count = 0;
    while (count < n) {
        bool found = false;
        for (int p = 0; p < n; p++) {
            if (finish[p] == false) {
                int j;
                for (j = 0; j < m; j++) {
                    if (need[p][j] > work[j]) {
                        break;
                    }
                }
```

```
            if (j == m) {
                for (int k = 0; k < m; k++) {
                    work[k] += allocation[p][k];
                }
                safeSequence[count++] = p;
                finish[p] = true;
                found = true;
            }
        }
    }
    if (found == false) {
        return false;
    }
  }
  return true;
}
// Check if the system is in a safe state
int processes[n];
for (int i = 0; i < n; i++) {
  processes[i] = i;
}
if (isSafe(n, m, processes, available, max, allocation)) {
  printf("System is in a safe state.\n");
} else {
  printf("System is not in a safe state.\n");
}
return 0;
}
```