Laboratory Assignments 4
Subject: Design Principles of Operating Systems
Subject code: CSE 3249

Assignment 4: Familiarization with Process Management in Linux environment.

Objective of this Assignment:

- To trace the different states of a process during its execution.
- To learn the use of different system calls such as (fork(),vfork(),wait(),execl()) for process handling in Unix/Linux environment.

1. Write a C program to create a child process using fork() system call. The child process will print the message "Child" with its process identifier and then continue in an indefinite loop. The parent process will print the message "Parent" with its process identifier and then continue in an indefinite loop.
   a) Run the program and trace the state of both processes.
   b) Terminate the child process. Then trace the state of processes.
   c) Run the program and trace the state of both processes. Terminate the parent process. Then trace the state of processes.
   d) Modify the program so that the parent process after displaying the message will wait for child process to complete its task. Again run the program and trace the state of both processes.
   e) Terminate the child process. Then trace the state of processes.

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

int main() {
   pid_t pid;

   pid = fork();  // Create a child process

   if (pid == 0) {  // Child process
      while (1) {
```

```
            printf("Child process: PID = %d\n", getpid());
            sleep(1);  // Sleep for 1 second to prevent overloading the terminal
        }
    } else if (pid > 0) {  // Parent process
        while (1) {
            printf("Parent process: PID = %d\n", getpid());
            sleep(1);  // Sleep for 1 second to prevent overloading the terminal
        }
    } else {
        // Fork failed
        perror("fork failed");
        return 1;
    }

    return 0;
}
```

2. Trace the output of the following codes:

a)	b)

int main( )
{
if(fork()==0)
printf("1");
else
printf("2");
printf("3");
return 0;
}
	int main( )
	{
	if(vfork()==0)
	{
	printf("1");
	_exit(0);
	}
	else
	printf("2");
	printf("3");
	}

1 3 2 3	1 3

c)

```
int main( )
{
 pid_t pid;
 int i=5;
 pid=fork();
 i=i+1;
 if(pid= =0)
 {
      printf("Child: %d",i);
  }
  else

  {
      wait(NULL);
      printf("Parent: %d",i);
   }
   return 0;
}
```

Child: 6
Parent: 6

d)

```
int main( )
{
  pid_t pid;
  int i=5;
  pid=vfork();
  i=i+1;
  if(pid==0)
  {
      printf("Child: %d",i);
      _exit(0);
  }

  else
 {
      printf("Parent: %d",i);
  }
  return 0;
 }
```

Child: 6

e)

```
 int main( )
{
 pid_t pid;
 int i=5;
 pid=fork();
 if(pid= =0)
 {
    i=i+1;
    printf("Child: %d",i);
  }
   else
  {
    wait(NULL);
    printf("Parent: %d",i);
   }
   return 0;
  }
```

Child: 6
Parent: 5

f)

```
int main( )
{
  pid_t pid;
  int i=5;
   pid=vfork();
   if(pid==0)
  {
     i=i+1;
     printf("Child: %d",i);
     _exit(0);
   }
   else
  {
     printf("Parent: %d",i);
   }
   return 0;
  }
```

Child: 6

g)
```
int main( )
{
 int i=5;
 if(fork( )==0)
 {
      printf("Child: %d",i);
 }
 else
 {
     printf("Parent: %d",i);
 }
 return 0;
}
```

Child: 5
Parent: 5

h)
```
int main( )
{
 int i=5;
 if(vfork( )==0)
 {
     printf("Child: %d",i);
     _exit(0);
 }
 else
 {
     printf("Parent: %d",i);
 }
  return 0;
}
```

Child: 5

i)
```
int main( )
{
 if(fork( )==0)
 {
      printf("1");
 }
 else
 {
       wait(NULL);
       printf("2");
       printf("3");
  }
  return 0;
 }
```

1 2 3

j)
```
int main( )
{
 if(vfork( )==0)
 {
      printf("1");
      _exit(0);
 }
  else
 {
      printf("2");
     printf("3");
 }
 return 0;
}
```

1 3

k)
```
int main( )
{
pid_t c1;
int n=10;
c1=fork( );
if(c1==0)
{
      printf(" Child\n");
      n=20;
      printf("n=%d \n",n);
 }
 else
{
       wait(NULL);
       printf("Parent\n");
       printf("n=%d \n",n);
  }
   return 0;
}
```

```
Child
n=20
Parent
n=10
```

l)
```
int main( )
{
 pid_t c1;
 int n=10;
 c1=vfork( );
 if(c1==0)
 {
       printf(" Child\n");
       n=20;
       printf("n=%d \n",n);
       _exit(0);
  }
  else
  {
        printf("Parent\n");
        printf("n=%d \n",n);
  }
   return 0;
}
```

```
Child
n=20
Parent
n=20
```

m)
```
int main( )
{
 int i=5;
 fork();
 i=i+1;
 fork();
 printf ( "% d",i);
 return 0;
}
```

n)
```
int main( )
{
 pid_t pid;
 int i=5;
 pid=vfork();
 if(pid==0)
 {
       printf("Child: %d",i);
       _exit(0);
  }
   else
   {
         i=i+1;
         printf("Parent: %d",i);
    }
    return 0;
}
```

6 6 6 6

Child: 5Parent: 6

o)
```
int main( )
{
 int i=5;
 if(fork()==0)
 i=i+1;
 else
 i=i-1;
 printf("%d",i);
 return 0;
}
```

6 4

p)
```
int main( )
{
 int i=5;
 if(vfork()==0)
 {
      i=i+1;
      _exit(0);
  }
  else
       i=i-1;
   fprintf(stderr,"%d",i);
   return 0;
   }
```

4

q)
```
 int main( )
{
 int j,i=5;
 for(j=1;j<3;j++)
{
      if(fork()==0)
      {
           i=i+1;
           break;
       }
else
   wait(NULL);
 }
 printf("%d",i);
 return 0;
 }
```

6 6 6

r)
```
int main( )
{
 int j,i=5;
 for(j=1;j<3;j++)
{
      if(fork()!=0)
     {
          i=i-1;
          break;
       }

  }
   fprintf(stderr,"%d",i);
   return 0;
  }
```

4

s)

```
int main( )
{
 if(fork() == 0)
 if(fork())
 printf("1\n");

 return 0;
}
```

1

t)

```
void fun1(){
 fork();
 fork();
 printf("1\n");
 }


 int main() {
 fun1();
 printf("1\n");
 return 0;
 }
```

1 1 1 1

3. Write a C program that will create three child process to perform the following operations respectively:
   - First child will copy the content of file1 to file2
   - Second child will display the content of file2
   - Third child will display the sorted content of file2 in reverse order.
   - Each child process being created will display its id and its parent process id with appropriate message.
   - The parent process will be delayed for 1 second after creation of each child process. It will display appropriate message with its id after completion of all the child processes.

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

void copy_file(const char *source, const char *destination) {
    FILE *src = fopen(source, "r");
    FILE *dest = fopen(destination, "w");
    if (!src || !dest) {
        perror("File open error");
        exit(EXIT_FAILURE);
    }

    char ch;
    while ((ch = fgetc(src)) != EOF) {
        fputc(ch, dest);
    }

    fclose(src);
    fclose(dest);
}

void display_file(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("File open error");
        exit(EXIT_FAILURE);
    }
```

```
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }
    fclose(file);
}

void display_sorted_reverse(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("File open error");
        exit(EXIT_FAILURE);
    }

    char lines[100][256];
    int count = 0;

    while (fgets(lines[count], sizeof(lines[count]), file)) {
        count++;
    }
    fclose(file);

    // Sort lines
    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            if (strcmp(lines[i], lines[j]) > 0) {
                char temp[256];
                strcpy(temp, lines[i]);
                strcpy(lines[i], lines[j]);
                strcpy(lines[j], temp);
            }
        }
    }

    // Display sorted lines in reverse order
    for (int i = count - 1; i >= 0; i--) {
        printf("%s", lines[i]);
    }
}
```

```
int main() {
    pid_t pid1, pid2, pid3;

    // First child: Copy content of file1 to file2
    if ((pid1 = fork()) == 0) {
        printf("First child (PID: %d, Parent PID: %d): Copying file1 to file2...\n", getpid(),
      getppid());
        copy_file("file1.txt", "file2.txt");
        printf("First child: File copy completed.\n");
        exit(0);
    }

    sleep(1); // Delay for 1 second after creating the first child

    // Second child: Display content of file2
    if ((pid2 = fork()) == 0) {
        printf("Second child (PID: %d, Parent PID: %d): Displaying content of file2...\n",
      getpid(), getppid());
        display_file("file2.txt");
        printf("\nSecond child: Content displayed.\n");
        exit(0);
    }

    sleep(1); // Delay for 1 second after creating the second child

    // Third child: Display sorted content of file2 in reverse order
    if ((pid3 = fork()) == 0) {
        printf("Third child (PID: %d, Parent PID: %d): Displaying sorted content of file2 in
      reverse order...\n", getpid(), getppid());
        display_sorted_reverse("file2.txt");
        printf("\nThird child: Sorted content displayed.\n");
        exit(0);
    }

    sleep(1); // Delay for 1 second after creating the third child

    // Parent process: Wait for all child processes to complete
    printf("Parent process (PID: %d): Waiting for children to complete...\n", getpid());
    waitpid(pid1, NULL, 0);
```

```
    waitpid(pid2, NULL, 0);
    waitpid(pid3, NULL, 0);
    printf("Parent process: All child processes completed. Exiting...\n");

    return 0;
  }
```

4. Write a C program that will create a child process to generate a Fibonacci series of specified length and store it in an array. The parent process will wait for the child to complete its task and then display the Fibonacci series and then display the prime Fibonacci number in the series along with its position with appropriate message.

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

// Function to check if a number is prime
bool is_prime(int num) {
   if (num <= 1) return false;
   for (int i = 2; i * i <= num; i++) {
      if (num % i == 0) return false;
   }
   return true;
}

// Function to generate Fibonacci series
void generate_fibonacci(int *fib, int length) {
   if (length >= 1) fib[0] = 0;
   if (length >= 2) fib[1] = 1;
   for (int i = 2; i < length; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
   }
}

int main() {
   int n;
```

```
printf("Enter the length of the Fibonacci series: ");
scanf("%d", &n);

if (n <= 0) {
    printf("Invalid length. Please enter a positive integer.\n");
    return EXIT_FAILURE;
}

int fib[n];
pid_t pid = fork();

if (pid < 0) {
    perror("Fork failed");
    return EXIT_FAILURE;
}

if (pid == 0) {
    // Child process: Generate Fibonacci series
    printf("Child process (PID: %d): Generating Fibonacci series...\n", getpid());
    generate_fibonacci(fib, n);

    // Write the Fibonacci series to a pipe
    int pipefd[2];
    pipe(pipefd);

    close(pipefd[0]); // Close read end of the pipe
    write(pipefd[1], fib, sizeof(fib));
    close(pipefd[1]); // Close write end of the pipe

    printf("Child process: Fibonacci series generated and sent to parent.\n");
    exit(0);
} else {
    // Parent process: Wait for child to complete
    wait(NULL);

    printf("Parent process (PID: %d): Receiving Fibonacci series from child...\n", getpid());
```

```
    // Read the Fibonacci series from the pipe
    int pipefd[2];
    pipe(pipefd);

    close(pipefd[1]); // Close write end of the pipe
    read(pipefd[0], fib, sizeof(fib));
    close(pipefd[0]); // Close read end of the pipe

    printf("Fibonacci series: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fib[i]);
    }
    printf("\n");

    // Identify and display prime Fibonacci numbers with their positions
    printf("Prime Fibonacci numbers:\n");
    for (int i = 0; i < n; i++) {
        if (is_prime(fib[i])) {
            printf("Position %d: %d\n", i + 1, fib[i]);
        }
    }

    printf("Parent process: Task completed.\n");
}

return 0;
  }
```