AIR UNIVERSITY

OS LAB TASK

MUHAMMAD ABDULLAH | 221546 | BSCYS-IIIA

CODE 1:

```
GNU nano 7.2
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid, mypid, myppid;
    pid = getpid();
    printf("Before fork: Process id is %d\n", pid);

    pid = fork();
    if (pid < 0) {
        perror("fork() failure\n");
        return 1;
    }

    if (pid == 0) {
        printf("This is child process\n");
        mypid = getpid();
        myppid = getppid();
        printf("Process id is %d and PPID is %d\n", mypid, myppid);
    } else {
        sleep(2);
        printf("This is parent process\n");
        mypid = getpid();
        myppid = getppid();
        printf("Process id is %d and PPID is %d\n", mypid, myppid);
        printf("Newly created process id or child pid is %d\n", pid);
    }

    return 0;
}
```

OUTPUT:

```
┌──(kali㉿kali)-[~/Documents]
└─$ ./lab
Before fork: Process id is 4144
This is child process
Process id is 4148 and PPID is 4144
This is parent process
Process id is 4144 and PPID is 1792
Newly created process id or child pid is 4148
```

CODE EXPLANATION:

This C program uses the **fork ()** system call to create a child process from the parent process. The parent and child processes then print their respective process IDs **(PID)** and the parent's parent process **ID (PPID)**. The child process prints its own **PID** and the **PPID** of its parent (which is the original process). The parent process additionally sleeps for 2 seconds before printing the process information to illustrate a delay between parent and child execution.

CODE 2:

```
GNU nano 7.2
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t cpid;

    if (fork() == 0) {
        exit(0);
    } else {
        cpid = wait(NULL);
        printf("Parent pid = %d\n", getpid());
        printf("Child pid = %d\n", cpid);
    }

    return 0;
}
```

OUTPUT:

```
┌──(kali㉿kali)-[~/Documents]
└─$ ./lab
Parent pid = 8890
Child pid = 8891
```

CODE EXPLANATION:

This C program uses the **fork ()** system call to create a child process. In the child process, it immediately exits, terminating the child. The parent process waits for the child to complete using **wait ()**, capturing the child's process ID. Finally, both the parent and child processes print their respective process **IDs**, demonstrating the creation and termination of the child process.

TASK 01:

```
GNU nano 7.2
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t child_pid;
    child_pid = fork();

    if (child_pid < 0)
    {
        perror("fork() failure");
        exit(EXIT_FAILURE);
    }
    else if (child_pid == 0)
    {
        printf("Child process (PID=%d) is created and becomes a zombie.\n", getpid());
        exit(0);
    }
    else
    {
        printf("Parent process (PID=%d) created a child with PID=%d.\n", getpid(), child_pid);
        sleep(10);
        printf("Parent process exits.\n");
    }

    return 0;
}
```

OUTPUT:

```
┌──(kali㉿kali)-[~/Documents]
└─$ ./lab
Parent process (PID=19706) created a child with PID=19710.
Child process (PID=19710) is created and becomes a zombie.
Parent process exits.
```

Now when we use the command **ps -l** it will enlist all the current running processes and we will check our respective process ID from the list as shown and highlighted below.

```
┌──(kali㉿kali)-[~/Documents]
└─$ Parent process exits.

[1]  + done       ./lab
┌──(kali㉿kali)-[~/Documents]
└─$ ./lab &
[1] 22718

Child process (PID=22720) is created and becomes a zombie.
Parent process (PID=22718) created a child with PID=22720.
┌──(kali㉿kali)-[~/Documents]
└─$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1792    1781  0  80   0 -  3406 sigsus pts/0    00:00:05 zsh
0 S  1000   22718    1792  0  85   5 -   616 hrtime pts/0    00:00:00 lab
1 Z  1000   22720   22718  0  85   5 -     0 -      pts/0    00:00:00 lab
0 R  1000   22751    1792  0  80   0 -  2824 -      pts/0    00:00:00 ps
```

When you append an **ampersand (&)** at the end of a command in the shell, it runs the command in the background.

CODE EXPLANATION:

- The C program initiates a child process using the **fork ()** system call, creating a new process in which the child immediately exits, becoming a zombie.
- The parent process, upon successfully creating the child, prints information about both the parent and child processes, including their respective process IDs **(PIDs).**
- The parent process then sleeps for ***10 seconds*** using the **sleep ()** function, allowing the child process to persist as a zombie during this period.
- After the sleep duration, the parent process exits, leaving the child process in a zombie state as the parent did not wait for the child's termination using functions like **wait().**
- When running the program in the background and checking the process status using **ps -l**, the child process with a state of **'Z' (zombie)** should be visible, confirming its existence in the system.

TASK 02:

```
  GNU nano 7.2                                              lab.c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int pno;  // Process number
  int pri;  // Priority (lower value means higher priority)
  int btime; // Burst time
  int wtime; // Waiting time
} Process;

int main() {
  int i, j, n;
  int total_burst_time = 0, total_waiting_time = 0;
  Process *p;

  printf("\n PRIORITY SCHEDULING.\n");
  printf("\n Enter the number of processes: ");
  scanf("%d", &n);

  // Allocate memory for n processes
  p = (Process*)malloc(n * sizeof(Process));
  if (p == NULL) {
    printf("Memory allocation failed!\n");
    return 1; // Indicate error
  }

  printf("Enter burst time and priority for each process:\n");
  for (i = 0; i < n; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d %d", &p[i].btime, &p[i].pri);
    p[i].pno = i + 1;
    p[i].wtime = 0;
  }

  // Priority-based sorting (bubble sort)
  for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
      if (p[i].pri > p[j].pri) {
        Process temp = p[i];
        p[i] = p[j];
        p[j] = temp;
      }
    }
  }

  printf("\n  Process\t  Burst Time\t  Waiting Time\t Turnaround Time\n");
  for (i = 0; i < n; i++) {
    total_burst_time += p[i].btime;
```

```
    p[i].wtime = (i > 0) ? p[i-1].wtime + p[i-1].btime : 0;  // Waiting time calculation
    int turnaround_time = p[i].wtime + p[i].btime;
    printf("   %d\t\t   %d\t\t   %d\t\t     %d\n", p[i].pno, p[i].btime, p[i].wtime, turnaround_time)
    total_waiting_time += p[i].wtime;
  }

  float avg_waiting_time = (float)total_waiting_time / n;
  float avg_turnaround_time = (float)(total_burst_time + total_waiting_time) / n;

  printf("\nTotal Waiting Time: %d\n", total_waiting_time);
  printf("Average Waiting Time: %.2f\n", avg_waiting_time);
  printf("Total Turnaround Time: %d\n", total_burst_time + total_waiting_time);
  printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

  // Free allocated memory
  free(p);

  return 0;
}
```

OUTPUT:

```
 PRIORITY SCHEDULING.

 enter the no of processes: 2
enter the burst time and priority:
process1:1

2
process2:2
3

 process       bursttime       waiting time     turnaround time
1              1               0                1
2              2               1                3

 total waiting time: 1
 average waiting time: 0.500000
 total turnaround time: 3
 average turnaround time: 1.500000
```

****