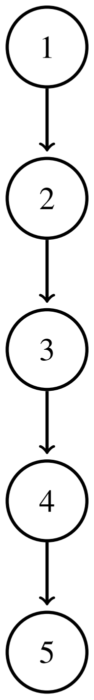
LABORATORY ASSIGNMENTS-04

UNIX Systems Programming (CSE 3041)

Experiment with programs, processes, memory allocation and manipulation, Learn to create processes and to explore the implication of process inheritance

Practice Assignment/ Minor Assignment::

1. (a) Develop a C code to create a chain of $n = 5$ processes by calling **`fork()`** in a loop as per the below structures. Display the process ID, parent ID and return value of `fork()` in the order 1, 2, 3, 4, and 5. Modify the code to print the process ID and parent ID such that each child will be completed before it's parent terminate. The value of n will be given from command-line.



   (b) Do experiment with different values of n and predict the output

2. How many times **`iter`** will be printed and how many processes are create by following code including program process.

```
int main()
{
   int pid,pid2;
   pid=fork();
   if(pid)
   {
     pid2=fork();
     printf("iter");
   }
   else
   {
     printf("iter");
     pid2=fork();
   }
return 0;
}
```

3. Modify the above code by adding a newline chracter `"\n"` in the **`else`** block printf statement. Then find how many times **`iter`** will be displayed.

4. Check how many **`ITER`** will be printed for the following code snippet?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
   fork();
   fork();
   fork();
   fork();
   printf("ITER\n");
   printf("ITER\n");
   printf("ITER\n");
   sleep(4);
   return 0;
}
```

   Find out a simple formula to compute the output for this like questions.

5. Calculate the number of **`hello`** will be printed for the below code; Find a manual solution and formula to compare with the output generated with this code.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(void)
{
  printf("hello\n");
  fork();
  printf("hello\n");
  fork();
  printf("hello\n");
  fork();
  printf("hello\n");
  sleep(2);
  return 0;
}
```

6. How many times the **`printf`** statement will be executed for the below given program? Also trace the manual output to match with the system generated output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
   pid_t childpid;
   int i, n=3;
   for (i = 1; i < n; i++){
      childpid=fork();
      if (childpid==-1)
         break;
   }
   printf("i:%d process ID:%ld\n",i,(long)getpid());
   return 0;
}
```

7. Check the output and state the reason for this output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
   printf("Fork create many confusion");
   fork();
   printf("We suppose to read and Practice\n");
   return 0;
}
```

8. Check the output and state the reason for this output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
   printf("Fork create many confusion\n");
   fork();
   printf("We suppose to read and Practice\n");
   return 0;
}
```

9. How many display will be there for the given below code sample;

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
   int i;
   for(i=0;i<4;i++)
      fork();
   printf("Learing by doing practice or !!!\n");
   return 0;
}
```

10. Predict the Output of the following program

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
void forksample() {
   if (fork()==0)
      printf("Hello from Child!\n");
   else
      printf("Hello from Parent!\n");
}
int main(void)
{
   forksample();
   return 0;
}
```

11. Predict the Output of the following program

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
void forksample() {
   if (fork()==0)
      printf("Hello from Child!\n");
   else{
      wait(NULL);
      printf("Hello from Parent!\n");
   }
}
int main(void)
{
   forksample();
   return 0;
}
```

12. Predict the output of the below code;

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork() && fork();
   printf("Really Interesting!!!\n");
   sleep(2);
   return 0;
}
```

13. Predict the output of the below code;

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork() || fork();
   printf("Really Interesting!!!\n");
   sleep(2);
   return 0;
}
```

14. Predict the output of the below code;

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork();
   fork() && fork();
   fork();
   printf("Really Interesting!!!\n");
   sleep(2);
   return 0;
}
```

15. Predict the output of the below code;

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork();
   fork() || fork();
   fork();
   printf("Really Interesting!!!\n");
   sleep(2);
   return 0;
}
```

16. Some unconditional fork is used in the below written code. Try to predict output of the program.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   fork();
   fork() && fork() || fork();
   fork();
   printf("Really Interesting!!!\n");
   sleep(2);
```

```
    return 0;
}
```

☞ **Many more variation can be created. Try yourself.**

17. What happens when the following program executes?

```
int main(void) {
   pid_t childpid, mypid;
   mypid = getpid();
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)/* child code */
      printf("I am child %ld, ID = %ld\n", (long)getpid(),
          (long)mypid);
   else                /* parent code */
      printf("I am parent %ld, ID = %ld\n", (long)getpid()
         , (long)mypid);
   return 0;
}
```

18. Change the statement **`if((childpid=fork())`**$\leq$ **`0)`** inside the for loop to **`if((childpid=fork())`**$\leq$ **`1)`** in the fan of n processes code. Show how many processes are created for each value of n. Now modify the program so that you can use **`ps`** command to see the processes that are created.

19. The following modification of the process fan of causes the original process to print out its information after all children have exited. Test the code and veify the output. The *errno* `EINTR` is the cause of *function was interrupted by a signal*. Add appropiate header as per the usages.

```
pid_t waitForAllChld(int *loc);
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;
   int status;
   if (argc != 2){
         /* check for valid number of command-line
            arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++){
      childpid=fork();
      if (childpid <=0)
         break;
   }
   while(waitForAllChld(NULL)>0);
   fprintf(stderr, "i:%d process ID:%ld parent ID:%ld
      child ID:%ld\n",i, (long)getpid(), (long)getppid(),
      (long)childpid);
   return 0;
}
pid_t waitForAllChld(int *loc){
   int retval;
   while(((retval=wait(loc))==-1)&& (errno==EINTR));
   return retval;
}
```

20. Now add the statement **`sleep(120)`** just before the **`return 0;`** statement in the above code. Now run the code to get the output and state the reason for getting that output.

21. Replace the system call **`wait()`** mentioned in the user defined function **`waitForAllChld(...)`** to its equivalent **`waitpid()`** system call and conclude the the output.

22. Display a count of how many processes are created by the following code;

```
#include < stdio.h >
#include < unistd.h >
int main()
{
     int i;
     for (i = 0; i < 4; i++)
          fork();
     return 0;
}
```

23. Using the program given below, identify the values of pid at lines A, B,C, and D . (*A test case for manual execution*: Assume that the actual pids of the parent and child are 2600 and 2603 respectively.)

```
int main()
{
   pid t pid, pid1;
   int status;
   pid = fork();
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid);          /* A */
      printf("child: pid1 = %d",pid1);        /* B */
   }
   else {                /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid);         /* C */
      printf("parent: pid1 = %d",pid1);       /* D */
      wait(&status);
   }
return 0;
}
```

24. What output will be at Line X and Line Y?

```
#define SIZE 5
int nums[SIZE] = { 0,1,2,3,4 } ;
int main()
{
   int i;
   pid t pid;
   pid = fork();
   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]);       /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]);      /* LINE Y */
      }
   return 0;
}
```

25. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8,

    Write a C program using the fork() system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child.

26. What output will be at Line A?

```
int value = 5;
int main()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) {
      value += 15;
      return 0;
   }
   else if (pid > 0) {
      wait(NULL);
      printf("PARENT: value= %d",value);     /* LINE A  */
      return 0;
```

```
    }
}
```

27. Write a C **`MulThree.c`** program to multiply three numbers and display the output. Now write another C program **`PracticeExecl.c`**, which will fork a child process and the child process will execute the file **`MulThree.c`** and generate the output. The parent process will wait till the termination of the child and the parent process will print the process ID and exit status of the child.

28. You know the usages of the command **`grep`**. Implement the working of **`grep -n pattern filename`** in a child process forked from the parent process using **`execl`** system call. The parent process will wait till the termination of the child and the parent process will print the process ID and exit status of the child.

29. Implement the above question using **`execv`**, **`execlp`**, **`execvp`**, **`execle`**, **`execve`** system calls.

30. Select few external(i.e. **`ls`**, **`wc -l -c`**, **`pr -t -5`**, $\cdots$) and shell builtin commands. Implement the execution of the command from a C file using **`execv`**, **`execlp`**, **`execvp`**, **`execle`**, **`execve`** system calls.