**OPERATING SYSTEMS**

**AIM:** Process Creation and

**ROLL NO: CE056**

**LAB NO: 03**

Termination (Use of fork,

wait, getpid, and getppid

system calls).

**FORK() SYSTEM CALL:**

* Fork system call is used to create a new child process same as parent process.
* The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content.
* The child has its own unique process ID, and this PID does not match the ID of any existing process group.
* The child's parent process ID is the same as the parent's process ID.
* **LIBRARIES TO BE INCLUDED:**

sys/types.h, unistd.h

* **SYNTAX**

pid\_t fork(void);

The fork() function doesn’t require any argument and the fork() function returns the pid of child process created by fork function and returns 0 to child process.

If because of limited resources or limited memory, child process is not generated then the fork() function returns -1 to parent process and child process is not created.

* **ZOMBIE PROCESS:**
* A process who has completed execution but whose entry still remain in process table, such a process is called zombie process.
* A child process always first becomes a zombie before being removed from the process table.
* **ORPHAN PROCESS:**
* The process whose parent process is already terminated is called as orphan process.
* An orphan process has a parent process with pid=1 which is called init process.

**GETPID AND GETPPID SYSTEM CALL:**

* getpid() system call returns the pid of the current process.
* getppid() system call returns the pid of parent process.
* **LIBRARIES TO BE INCLUDED:**

sys/types.h, unistd.h

* **SYNTAX:**

pid\_t getpid(void);

pid\_t getppid(void);

* getpid() and getppid() function doesn’t require any argument.
* Both functions return pid the process.
* **WAIT SYSTEM CALL:**
* Wait system call waits for process to change state.
* All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
* If a child has already changed state, then this call returns immediately. Otherwise, it blocks until either a child changes state or a signal handler interrupts the call.
* **LIBRARIES TO BE INCLUDED:**

sys/types.h

sys/wait.h

* **SYNTAX:**

pid\_t wait(int \*status)

* **PROGRAMS:**

**1. Call fork once, twice, thrice and print “Hello”. Observe and interpret the outcomes.**

* Once:

#include <unistd.h>

#include <stdio.h>

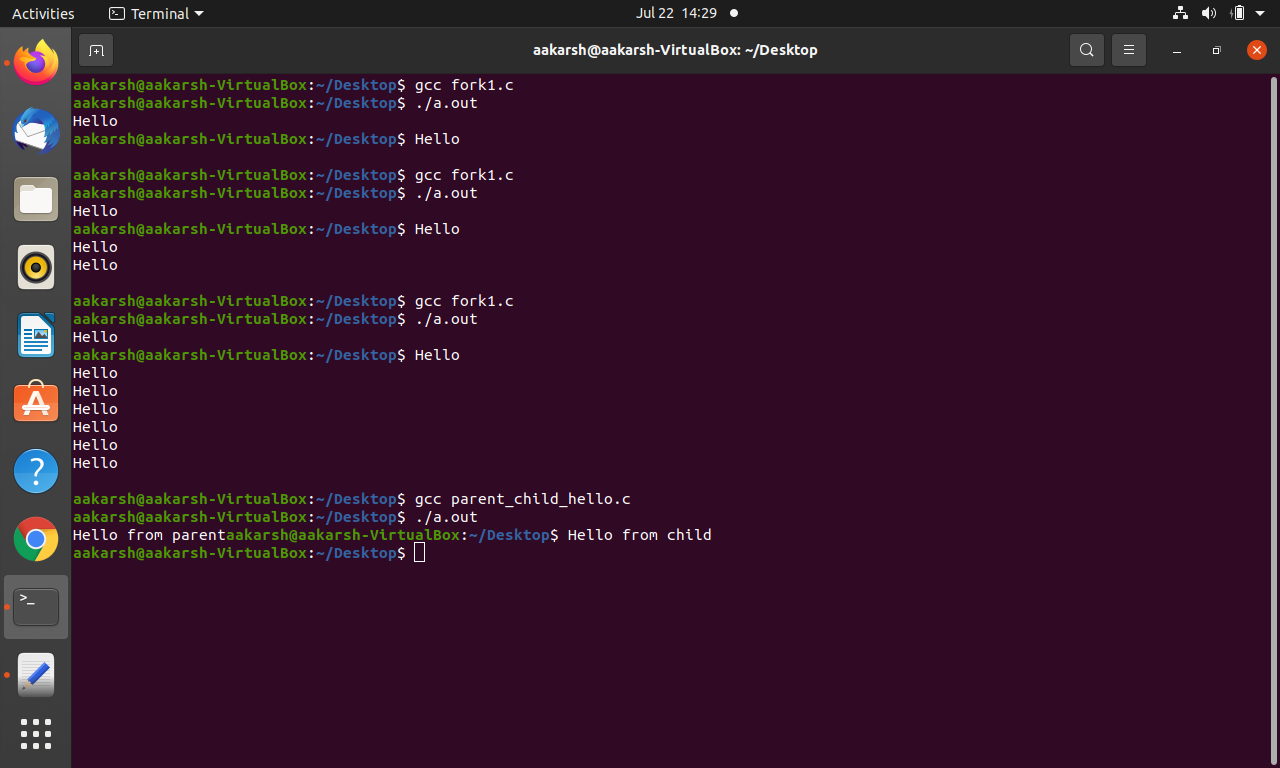
int main()

{

fork();

printf("Hello\n");

}

* **OUTPUT:**

* Hello is printed by parent as well as child process.
* Twice:

#include <unistd.h>

#include <stdio.h>

int main()

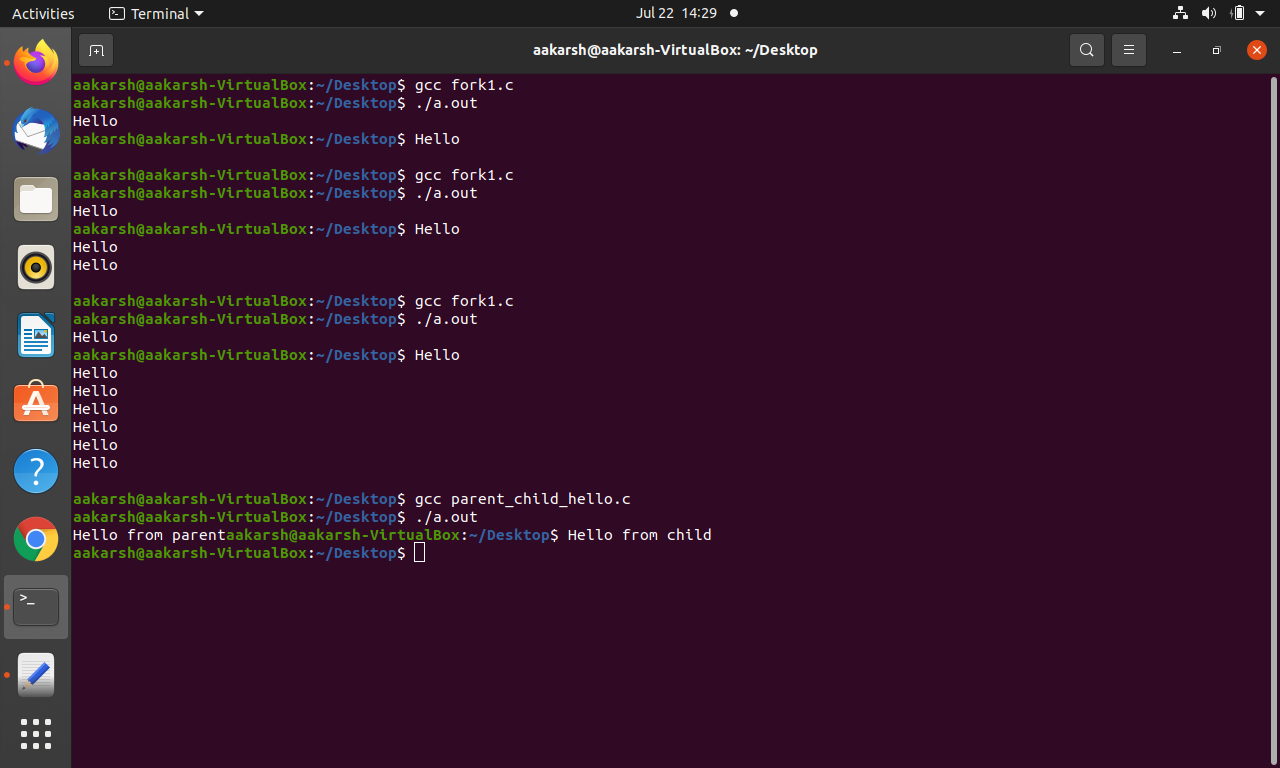
{

fork();

fork();

printf("Hello\n");

}

* By first fork 1 child is created, by second process 2 more child are created. So, total 1 parent and 2 child processes are there so total 4 times “Hello” will be printed.
* Thrice:

#include <unistd.h>

#include <stdio.h>

int main()

{

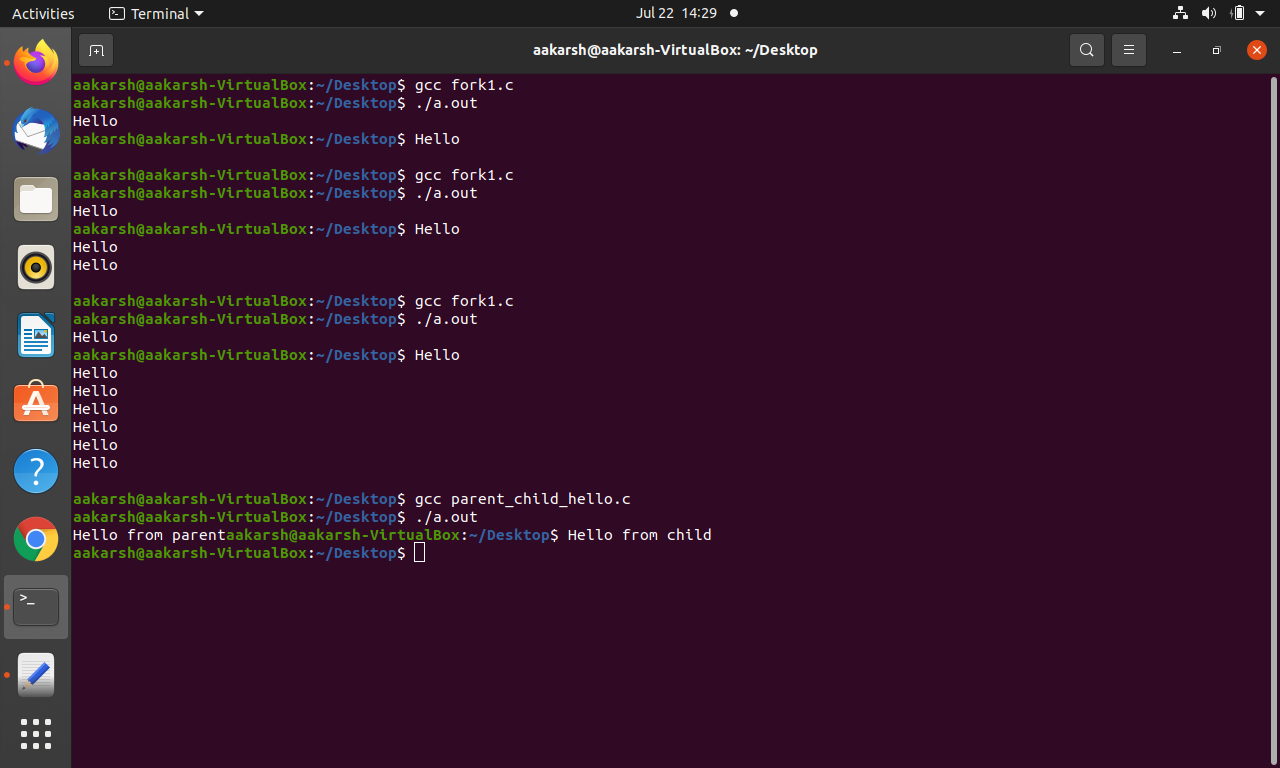
fork();

fork();

fork();

printf("Hello\n");

}



* By first process, 1 child is generated. By second fork 2more child is generated, by third fork 4 more child is generated. So, in total 8 processes consisting of 1 parent process and 7 child process is generated.

**2. Print “Hello from Parent” using the parent process and “Hello from Child” using child process.**

#include <unistd.h>

#include <stdio.h>

int main()

{

int pid = fork();

if(pid == -1)

{

printf("Error");

}

else if(pid == 0)

{

printf("Hello from child");

}

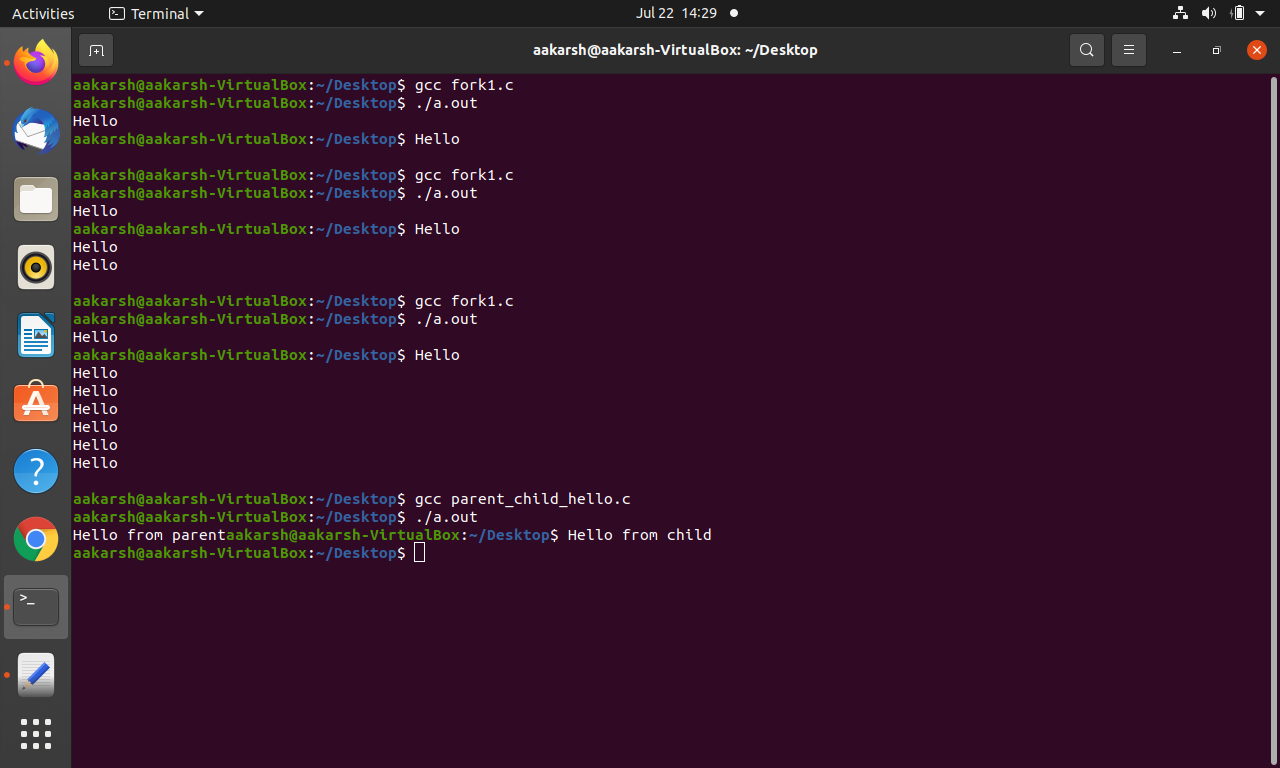
else

{

printf("Hello from parent");

}

}



**3. Print PID and PPID for parent and child processes. Observe and interpret the outcomes.**

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

int main()

{

int pid = fork();

if(pid == -1)

{

printf("Error");

}

else if(pid == 0)

{

printf("Hello from child(%d) of parent(%d)",getpid(),getppid());

}

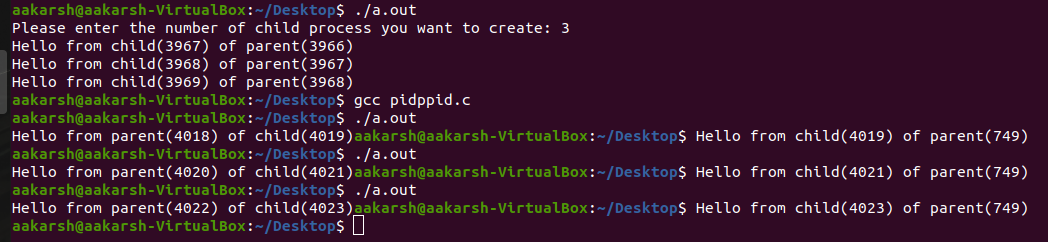
else

{

printf("Hello from parent(%d) of child(%d)",getpid(),pid);

}

}



* Here we could note that child’s id of parent process matches the child’s process’s pid.
* Similarly, the pid of parent process should match to pid to parent process of child process.
* But here due to process switching, the id’s are not matching.

**4. Add wait to the code of task 2. Observe and interpret the outcomes.**

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

int pid = fork();

if(pid == -1)

{

printf("Error");

}

else if(pid == 0)

{

printf("Hello from child(%d) of parent(%d)",getpid(),getppid());

}

else

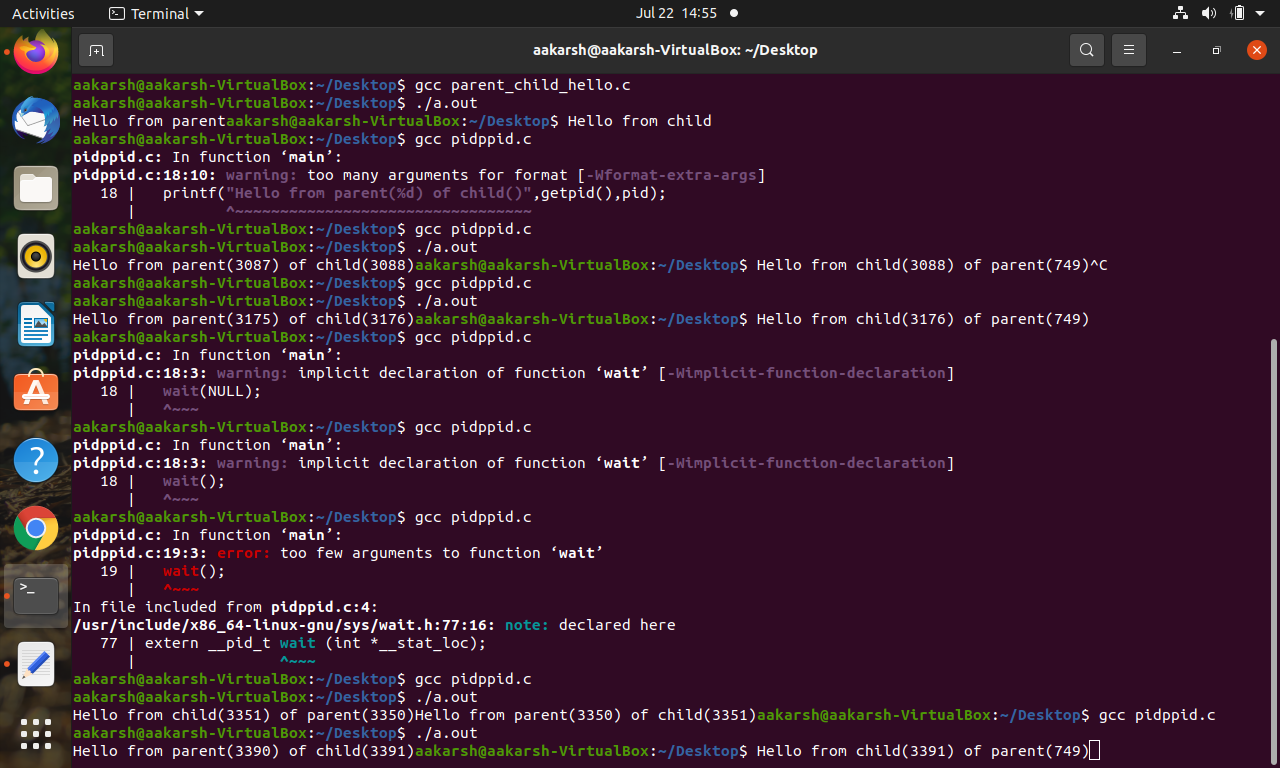
{

wait(NULL);

printf("Hello from parent(%d) of child(%d)",getpid(),pid);

}

}



**5. Write a program to implement fan of n processes.**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

int number\_of\_child;

printf("Please enter the number of child process you want to create: ");

scanf("%d",&number\_of\_child);

int pid = fork();

for(int i=0;i<number\_of\_child;i++)

{

if(pid == -1)

{

printf("Error");

}

else if(pid == 0)

{

printf("Hello from child(%d) of parent(%d)\n",getpid(),getppid());

exit(0);

}

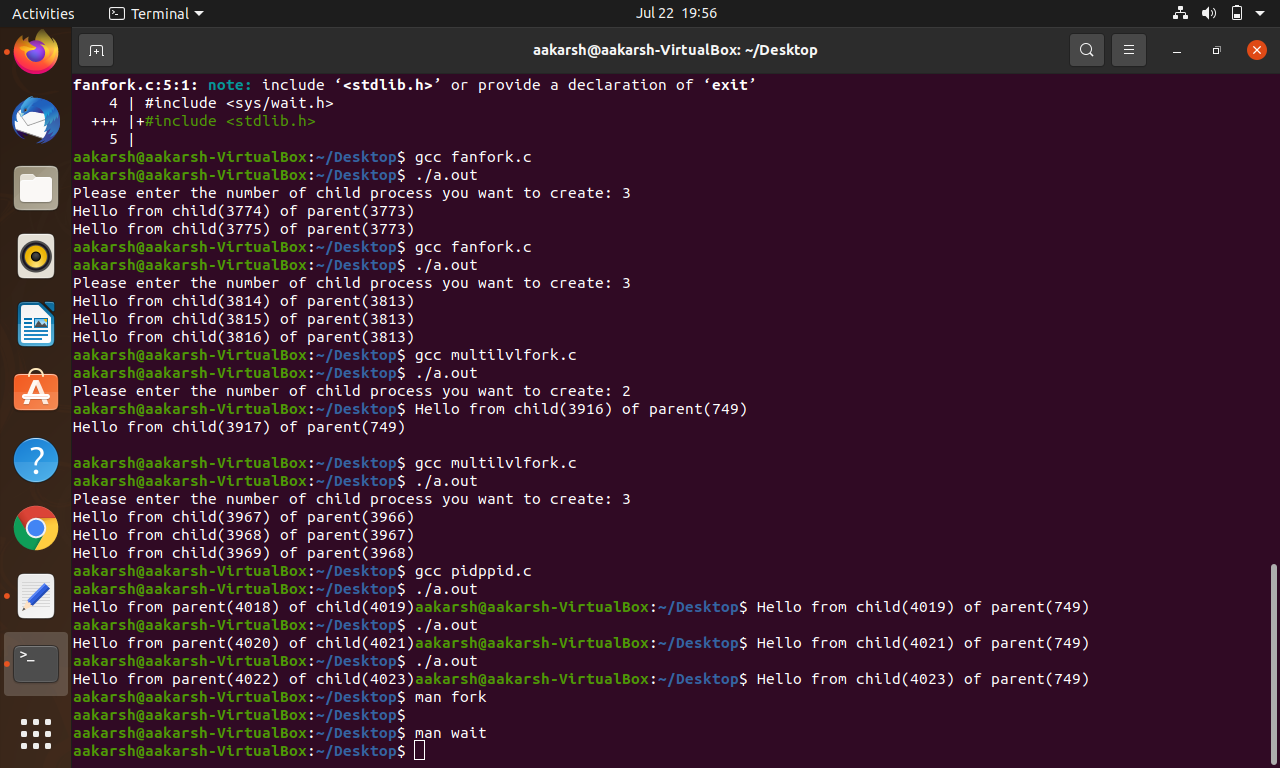
else

{

wait(NULL);

}

pid=fork();}



**6. Write a program to implement chain of n processes.**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

int number\_of\_child;

printf("Please enter the number of child process you want to create: ");

scanf("%d",&number\_of\_child);

int pid = fork();

for(int i=0;i<number\_of\_child;i++)

{

if(pid == -1)

{

printf("Error");

}

else if(pid == 0)

{

printf("Hello from child(%d) of parent(%d)\n",getpid(),getppid());

}

else

{

wait(NULL);

exit(0);

}

pid=fork();

}

}

