Thread

How to Create and Run a Thread ?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>


void *SampleThread(void *vargp)
{
    int i = 0;

    printf("SampleThread is running ... \n");
    for(i = 0; i < 15; i++) {
        sleep(1);
        printf("timer running inside thread = %d\n", i);
    }
    printf("SampleThread is exiting ... \n");
    return NULL;
}

int main()
{
    int i = 0;

    pthread_t tid;
    pthread_create(&tid, NULL, SampleThread, NULL);

    for(i = 0; i < 5; i++) {
        sleep(2);
        printf("timer running outside thread = %d\n", i);
    }

    // this is to make it sure that the program (Application) waiting until
the tid thread
    // completes. without this routine, the application move to the next step
right away
    // exit(0) in this example.
    pthread_join(tid, NULL);

    exit(0);
}

Result :-----------------------------------------------
```

// What I am trying to show you is that printf() outside of thread is running even while printf() within the thread is running. The part in red is from inside of Thread and the part in black is from outside of Thread.

```
SampleThread is running ...
timer running inside thread = 0
timer running outside thread = 0
timer running inside thread = 1
timer running inside thread = 2
timer running outside thread = 1
timer running inside thread = 3
timer running inside thread = 4
timer running outside thread = 2
timer running inside thread = 5
timer running inside thread = 6
timer running outside thread = 3
timer running inside thread = 7
timer running inside thread = 8
timer running outside thread = 4
timer running inside thread = 9
timer running inside thread = 10
timer running inside thread = 11
timer running inside thread = 12
timer running inside thread = 13
timer running inside thread = 14
SampleThread is exiting ...
```

Running Two Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void *SampleThread1(void *vargp)
{
    int i = 0;

    printf("SampleThread(1) is running ... \n");
    for(i = 0; i < 10; i++) {
        sleep(1);
        printf("timer running inside SampleThread(1) = %d\n", i);
    }
    printf("SampleThread(1) is exiting ... \n");
    return NULL;
};


void *SampleThread2(void *vargp)
{
```

```
    int i = 0;

    printf("SampleThread(2) is running ... \n");
    for(i = 0; i < 15; i++) {
        sleep(1);
        printf("timer running inside SampleThread(2) = %d\n", i);
    }
    printf("SampleThread(2) is exiting ... \n");
    return NULL;
};


int main()
{
    int i = 0;

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, SampleThread1, NULL);
    pthread_create(&tid2, NULL, SampleThread2, NULL);

    for(i = 0; i < 7; i++) {
        sleep(2);
        printf("timer running outside thread = %d\n", i);
    }
    printf("timer outside Thread is ended ..\n");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    exit(0);
}


Result :-------------------------------------------------------

SampleThread(1) is running ...
SampleThread(2) is running ...
timer running inside SampleThread(1) = 0
timer running inside SampleThread(2) = 0
timer running outside thread = 0
timer running inside SampleThread(2) = 1
timer running inside SampleThread(1) = 1
timer running inside SampleThread(2) = 2
timer running inside SampleThread(1) = 2
timer running outside thread = 1
timer running inside SampleThread(2) = 3
timer running inside SampleThread(1) = 3
timer running inside SampleThread(2) = 4
timer running inside SampleThread(1) = 4
timer running outside thread = 2
timer running inside SampleThread(2) = 5
timer running inside SampleThread(1) = 5
timer running inside SampleThread(2) = 6
```

```
timer running inside SampleThread(1) = 6
timer running outside thread = 3
timer running inside SampleThread(2) = 7
timer running inside SampleThread(1) = 7
timer running inside SampleThread(2) = 8
timer running inside SampleThread(1) = 8
timer running outside thread = 4
timer running inside SampleThread(2) = 9
timer running inside SampleThread(1) = 9
SampleThread(1) is exiting ...
timer running inside SampleThread(2) = 10
timer running outside thread = 5
timer running inside SampleThread(2) = 11
timer running inside SampleThread(2) = 12
timer running outside thread = 6
timer outside Thread is ended ..
timer running inside SampleThread(2) = 13
timer running inside SampleThread(2) = 14
SampleThread(2) is exiting ...
```

1. **/ C program to find maximum number of thread within**
// a process

```
#include<stdio.h>
#include<pthread.h>

// This function demonstrates the work of thread
// which is of no use here, So left blank
void *thread ( void *vargp){      }

int main()
{
    int err = 0, count = 0;
    pthread_t tid;

    // on success, pthread_create returns 0 and
    // on Error, it returns error number
    // So, while loop is iterated until return value is 0
    while (err == 0)
    {
        err = pthread_create (&tid, NULL, thread, NULL);
        count++;
    }
    printf("Maximum number of thread within a Process"
                                   " is : %d\n", count);
}
```

Output:

```
Maximum number of thread within a Process is : 32754
```

2. **A simple C program to demonstrate use of pthread basic functions**
   Please note that the below program may compile only with C compilers with pthread library.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep().
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing Quiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.
pthread_create() takes 4 arguments.
The first argument is a pointer to thread_id which is set by this function.
The second argument specifies attributes. If the value is NULL, then default attributes shall be used.
The third argument is name of function to be executed for the thread to be created.
The fourth argument is used to pass arguments to the function, myThreadFun.
The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

How to compile above program?
To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Before Thread
Printing Quiz from Thread
After Thread
gfg@ubuntu:~/$
```

2.b. Retrieving Process Identifiers: getpid() and getppid()

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
  pid_t pid, ppid;

  //get the process'es pid
  pid = getpid();

  //get the parrent of this process'es pid
  ppid = getppid();

  printf("My pid is: %d\n",pid);
  printf("My parent's pid is %d\n", ppid);

  return 0;
}
```

If we run this program a bunch of times, we will see output like this:

```
#> ./get_pid_ppid
My pid is: 14307
My parent's pid is 13790

#> ./get_pid_ppid
My pid is: 14308
My parent's pid is 13790

#> ./get_pid_ppid
My pid is: 14309
My parent's pid is 13790
```

3. **A C program to show multiple threads with global and static variables**

As mentioned above, all threads share data segment. Global and static variables are stored in data segment. Therefore, they are shared by all threads. The following example program demonstrates the same.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);

    pthread_exit(NULL);
    return 0;
}
```

OUTPUT:

```
gfg@ubuntu:~/$ gcc multithread.c -lpthread
gfg@ubuntu:~/$ ./a.out
Thread ID: 3, Static: 2, Global: 2
Thread ID: 3, Static: 4, Global: 4
Thread ID: 3, Static: 6, Global: 6
gfg@ubuntu:~/$
```

Please note that above is simple example to show how threads work. Accessing a global variable in a thread is generally a bad idea. What if thread 2 has priority over thread 1 and thread 1 needs to change the variable. In practice, if it is required to access global variable by multiple threads, then they should be accessed using a mutex

//Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include<string.h>
void *thread_function(void *arg);
int i,n,j;
int main() {
char *m="5";
pthread_t a_thread;  //thread declaration
void *result;
pthread_create(&a_thread, NULL, thread_function, m); //thread is created
pthread_join(a_thread, &result); //process waits for thread to finish .
Comment this line to see the difference
printf("Thread joined\n");
for(j=20;j<25;j++)
{
printf("%d\n",j);
sleep(1);
}
printf("thread returned %s\n",result);
}
void *thread_function(void *arg) {    // the work to be done by the thread is
defined in this function
int sum=0;
n=atoi(arg);

for(i=0;i<n;i++)
{
printf("%d\n",i);
```

```
sleep(1);
}
pthread_exit("Done"); // Thread returns "Done"
}
```

Program 2:

//Program to create a thread. The thread is passed more than one input from the main process.
//For passing multiple inputs we need to create structure and include all the variables that are to be passed in this structure.

```
#include <stdio.h>
#include <pthread.h>

struct arg_struct {   //structure which contains multiple variables that are to passed as input to the thread
    int arg1;
    int arg2;
};

void *arguments(void *arguments)
{
    struct arg_struct *args=arguments;
    printf("%d\n", args -> arg1);
    printf("%d\n", args -> arg2);
    pthread_exit(NULL);

}

int main()
{
    pthread_t t;
    struct arg_struct args;
    args.arg1 = 5;
    args.arg2 = 7;
    pthread_create(&t, NULL, arguments, &args); //structure passed as 4th
```

argument
pthread_join(t, NULL); /* Wait until thread is finished */
}

4. a. **USING Fork()**

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
 pid_t c_pid;
 c_pid = fork(); //duplicate
 if( c_pid == 0 ){
  //child: The return of fork() is zero
  printf("Child: I'm the child: %d\n", c_pid);

 }else if (c_pid > 0){
  //parent: The return of fork() is the process of id of the child
  printf("Parent: I'm the parent: %d\n", c_pid);
 }else{
  //error: The return of fork() is negative

  perror("fork failed");
  _exit(2); //exit failure, hard
 }
 return 0; //success
}
```

4.b. Using fork() to produce 1 parent and its 3 child processes

Program to create four processes (1 parent and 3 children) where they terminates in a sequence as follows :
(a) Parent process terminates at last
(b) First child terminates before parent and after second child.
(c) Second child terminates after last and before first child.
(d) Third child terminates first.
Prerequisite : fork(),

```
// CPP code to create three child
// process of a parent
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Driver code
int main()
{
```

```
    int pid, pid1, pid2;

    // variable pid will store the
    // value returned from fork() system call
    pid = fork();

    // If fork() returns zero then it
    // means it is child process.
    if (pid == 0) {

        // First child needs to be printed
        // later hence this process is made
        // to sleep for 3 seconds.
        sleep(3);

        // This is first child process
        // getpid() gives the process
        // id and getppid() gives the
        // parent id of that process.
        printf("child[1] --> pid = %d and ppid = %d\n",
               getpid(), getppid());
    }

    else {
        pid1 = fork();
        if (pid1 == 0) {
            sleep(2);
            printf("child[2] --> pid = %d and ppid = %d\n",
                   getpid(), getppid());
        }
        else {
            pid2 = fork();
            if (pid2 == 0) {
                // This is third child which is
                // needed to be printed first.
                printf("child[3] --> pid = %d and ppid = %d\n",
                       getpid(), getppid());
            }

            // If value returned from fork()
            // in not zero and >0 that means
            // this is parent process.
            else {
                // This is asked to be printed at last
                // hence made to sleep for 3 seconds.
                sleep(3);
                printf("parent --> pid = %d\n", getpid());
            }
        }
    }

    return 0;
}
```

Output:

```
child[3]-->pid=50 and ppid=47
```

```
child[2]-->pid=49 and ppid=47
child[1]-->pid=48 and ppid=47
parent-->pid=47
```

4.c. Create n-child process from same parent process using fork() in C

```
#include<stdio.h>


int main()
{
    for(int i=0;i<5;i++) // loop will run n times (n=5)
    {
        if(fork() == 0)
        {
            printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());
            exit(0);
        }
    }
    for(int i=0;i<5;i++) // loop will run n times (n=5)
    wait(NULL);

}
```

Output:

```
[son] pid 28519 from [parent] pid 28518
[son] pid 28523 from [parent] pid 28518
[son] pid 28520 from [parent] pid 28518
[son] pid 28521 from [parent] pid 28518
[son] pid 28522 from [parent] pid 28518
```

4.d. Using Fork(), exec(), wait()

```
/*fork_exec_wait.c*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char * argv[]){
  //arguments for ls, will run: ls  -l /bin
  char * ls_args[3] = { "ls", "-l", NULL} ;
  pid_t c_pid, pid;
  int status;

  c_pid = fork();
```

```
  if (c_pid == 0){
    /* CHILD */

    printf("Child: executing ls\n");

    //execute ls
    execvp( ls_args[0], ls_args);
    //only get here if exec failed
    perror("execve failed");
  }else if (c_pid > 0){
    /* PARENT */

    if( (pid = wait(&status)) < 0){
      perror("wait");
      _exit(1);
    }

    printf("Parent: finished\n");

  }else{
    perror("fork failed");
    _exit(1);
  }
  return 0; //return success
}
```

And the execution:

```
aviv@saddleback: demo $ ./fork_exec_wait
Child: executing ls
total 5120
-rwxr-xr-x  2 root  wheel    18480 Sep  9 18:44 [
-r-xr-xr-x  1 root  wheel   628736 Sep 26 22:03 bash
```

5. **Fork(), Kill(), signal()**

C signal handling- Communication between child and parent processes
In this post, the communication between child and parent processes is done using kill() and signal(), fork() system call.

- **fork()** creates the child process from the parent. The pid can be checked to decide whether it is the child (if pid == 0) or the parent (pid = child process id).
- The parent can then send messages to child using the pid and kill().
- The child picks up these signals with signal() and calls appropriate functions.

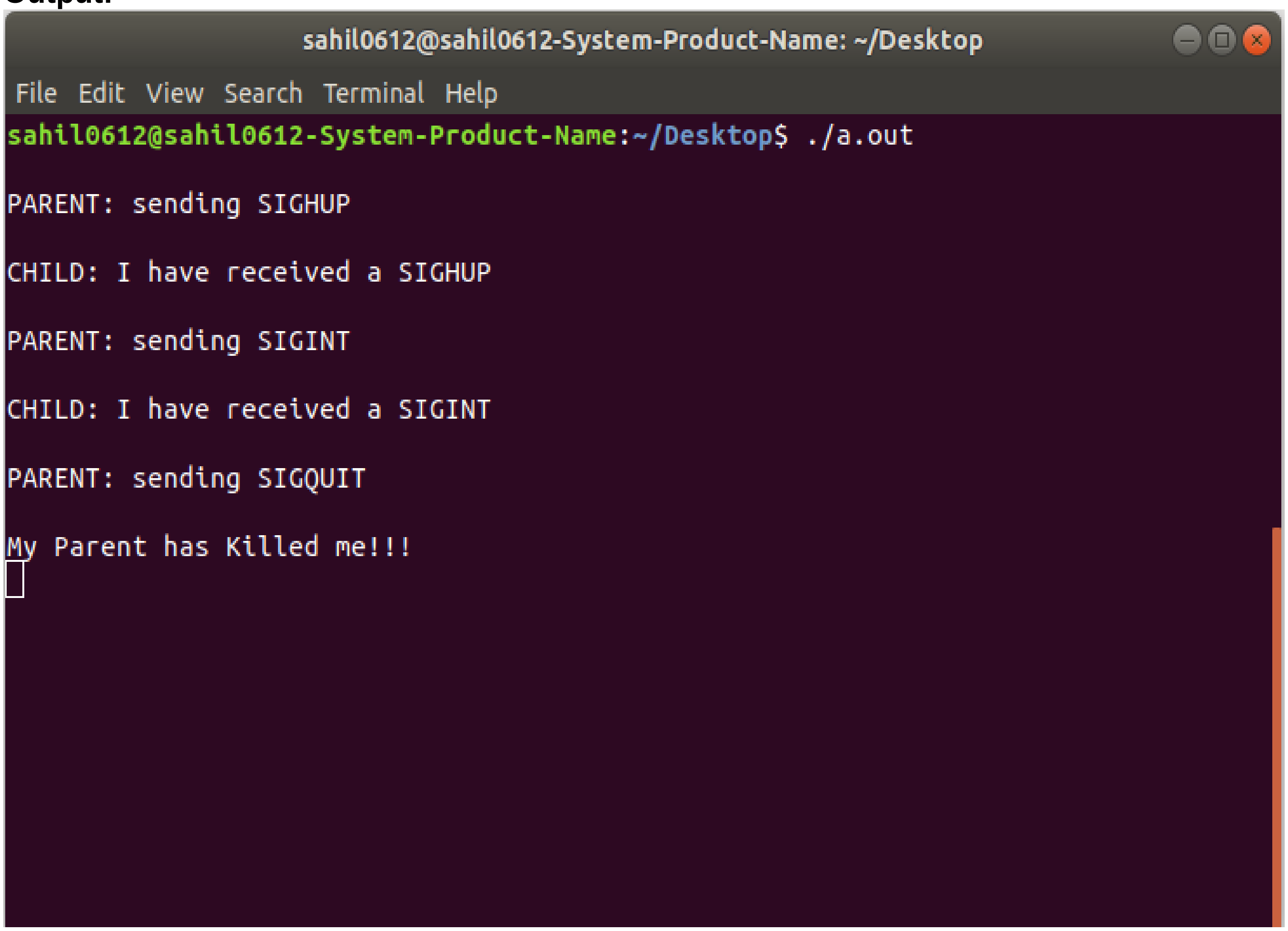
Example of how 2 processes can talk to each other using kill() and signal():

```
// C program to implement sighup(), sigint()
// and sigquit() signal functions
```

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

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
    int pid;

    /* get child process */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { /* child */
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for (;;)
            ; /* loop for ever */
    }

    else /* parent */
    { /* pid hold id of child */
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid, SIGHUP);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid, SIGINT);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid, SIGQUIT);
        sleep(3);
    }
}

// sighup() function definition
void sighup()

{
    signal(SIGHUP, sighup); /* reset signal */
    printf("CHILD: I have received a SIGHUP\n");
}
```

```
// sigint() function definition
void sigint()

{
    signal(SIGINT, sigint); /* reset signal */
    printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

Output:



5. **Using Fork() and Pipe()**

Pass the value from child process to parent process

Prerequisite: Pipe() and Fork() Basic

Write a C program in which the child process takes an input array and send it to the parent process using pipe() and fork() and then print it in the parent process.

Examples: Suppose we have an array a[ ]={1, 2, 3, 4, 5} in child process, then output should be 1 2 3 4 5.

```
Input:  1 2 3 4 5
Output: 1 2 3 4 5
```

```
// C program for passing value from
// child process to parent process
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define MAX 10

int main()
{

  int fd[2], i = 0;
  pipe(fd);
  pid_t pid = fork();

   if(pid > 0) {
      wait(NULL);

      // closing the standard input
      close(0);

      // no need to use the write end of pipe here so close it
      close(fd[1]);

       // duplicating fd[0] with standard input 0
      dup(fd[0]);
      int arr[MAX];

      // n stores the total bytes read succesfully
      int n = read(fd[0], arr, sizeof(arr));
      for ( i = 0;i < n/4; i++)

         // printing the array received from child process
          printf("%d ", arr[i]);
  }
  else if( pid == 0 ) {
      int arr[] = {1, 2, 3, 4, 5};

      // no need to use the read end of pipe here so close it
      close(fd[0]);

       // closing the standard output
      close(1);

      // duplicating fd[0] with standard output 1
```

```
        dup(fd[1]);
        write(1, arr, sizeof(arr));
    }

    else {
        perror("error\n"); //fork()
    }
}
```

Steps for executing above code:

- To compile, write **gcc program_name.c**
- To run, write **./a.out**

7.

Zombie Processes and their Prevention

Zombie Process or **Defunct Process** are those Process which has completed their execution by exit() system call but still has an entry in **Process Table**. It is a process in terminated state.
When child process is created in **UNIX** using **fork()** system call, then if somehow parent process were not available to reap child process from Process Table, then this situation arise. Basically, **Zombie Process** is neither completely **dead** nor completely **alive** but it has having some state in between.

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.
In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.

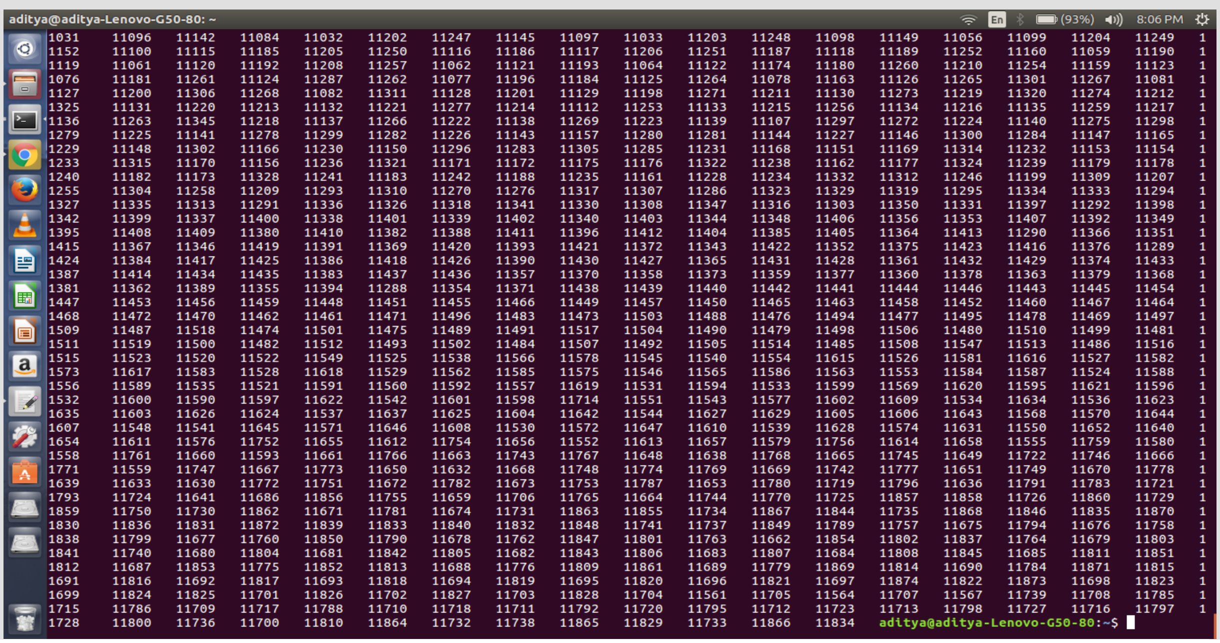
```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);
```

```
    // Child process
    else
        exit(0);

    return 0;
}
```

Note that the above code may not work with online compiler as fork() is disabled.

```
// C program to find number of Zombie processes a  system can handle.
#include<stdio.h>
#include<unistd.h>

int main()
{
    int count = 0;
    while (fork() > 0)
    {
        count++;
        printf("%d\t", count);
    }
}
```

Output:



In the image, we can see after 11834**, increment of count get stopped. However, this is not a fixed number but it will come around it.**

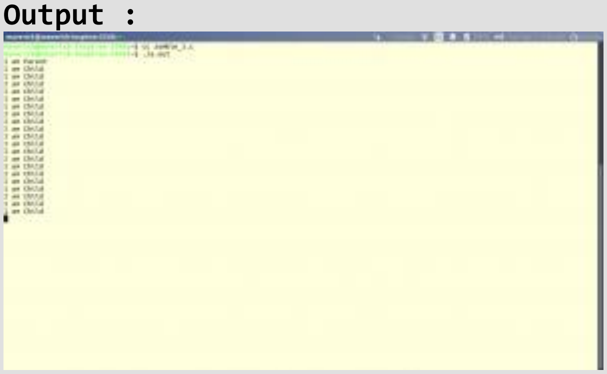
Also, it will depend upon system configuration and strength.

Prerequisites: [fork() in C], [Zombie Process]

Zombie state : When a process is created in UNIX using fork() system call, the address space of the Parent process is replicated. If the parent process calls wait() system call, then the execution of parent is suspended until the child is terminated. At the termination of the child, a ‘SIGCHLD’ signal is generated which is delivered to the parent by the kernel. Parent, on receipt of ‘SIGCHLD’ reaps the status of the child from the process table. Even though, the child is terminated, there is an entry in the process table corresponding to the child where the status is stored. When parent collects the status, this entry is deleted. Thus, all the traces of the child process are removed from the system. If the parent decides not to wait for the child’s termination and it executes its subsequent task, then at the termination of the child, the exit status is not read. Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.

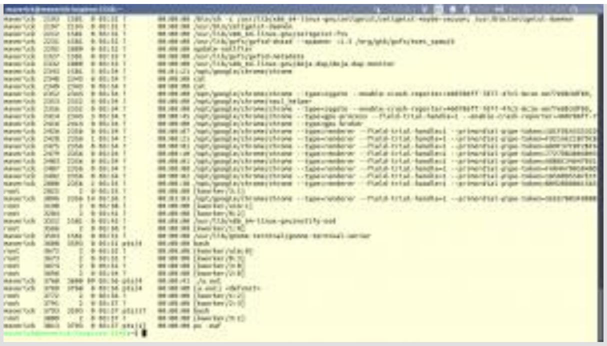
```
// A C program to demonstrate working of fork() and process table entries.
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();

    if (pid == 0)
    {
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        printf("I am Parent\n");
        while(1);
    }
}
```

Output :



Now check the process table using the following command in the terminal

$ ps -eaf



Here the entry **[a.out] defunct** shows the zombie process.

Why do we need to prevent the creation of Zombie process?
There is one process table per system. The size of the process table is finite. If too many zombie processes are generated, then the process table will be full. That is, the system will not be able to generate any new process, then the system will come to a standstill. Hence, we need to prevent the creation of zombie processes.

Three Different ways in which creation of Zombie can be prevented

1. Using wait() system call : When the parent process calls wait(), after the creation of child, it indicates that, it will wait for the child to complete and it will reap the exit status of the child. The parent process is suspended(waits in a waiting queue) until the child is terminated. It must be understood that during this period, the parent process does nothing just waits.

```
// A C program to demonstrate working of
// fork()/wait() and Zombie processes
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();
    if (pid==0)
    {
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        wait(NULL);
        printf("I am Parent\n");
        while(1);
    }
}
```

2. By ignoring the SIGCHLD signal : When a child is terminated, a corresponding SIGCHLD signal is delivered to the parent, if we call the ‘signal(SIGCHLD,SIG_IGN)’, then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. Thus, no zombie is created. However, in this case, the parent cannot know about the exit status of the child.

```c
// A C program to demonstrate ignoring
// SIGCHLD signal to prevent Zombie processes
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();
    if (pid == 0)
        for (i=0; i<20; i++)
            printf("I am Child\n");
    else
    {
        signal(SIGCHLD,SIG_IGN);
        printf("I am Parent\n");
        while(1);
    }
}
```