

## 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 *)&i);

    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()
        // is 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

```

```

#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:

```
sahil0612@sahil0612-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
sahil0612@sahil0612-System-Product-Name:~/Desktop$ ./a.out
PARENT: sending SIGHUP
CHILD: I have received a SIGHUP
PARENT: sending SIGINT
CHILD: I have received a SIGINT
PARENT: sending SIGQUIT
My Parent has Killed me!!!
█
```

### 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 successfully
        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:

```

aditya@aditya-Lenovo-G50-80: ~
1031 11096 11142 11084 11032 11202 11247 11145 11097 11033 11203 11248 11098 11149 11056 11099 11204 11249 1
1152 11100 11115 11185 11205 11250 11116 11186 11117 11206 11251 11187 11118 11189 11252 11160 11059 11190 1
1119 11061 11120 11192 11208 11257 11062 11121 11193 11064 11122 11174 11180 11260 11210 11254 11159 11123 1
1076 11181 11261 11124 11287 11262 11077 11196 11184 11125 11264 11078 11163 11126 11265 11301 11267 11081 1
1127 11200 11306 11268 11082 11311 11128 11201 11129 11198 11271 11211 11130 11273 11219 11320 11274 11212 1
1325 11131 11220 11213 11132 11221 11277 11214 11112 11253 11133 11215 11256 11134 11216 11135 11259 11217 1
1136 11263 11345 11218 11137 11266 11222 11138 11269 11223 11139 11107 11297 11272 11224 11146 11275 11298 1
1279 11225 11141 11278 11299 11282 11226 11143 11157 11280 11281 11144 11227 11146 11300 11284 11147 11165 1
1229 11148 11302 11166 11230 11150 11296 11283 11305 11285 11231 11168 11151 11169 11314 11232 11153 11154 1
1233 11315 11170 11156 11236 11321 11171 11172 11175 11176 11322 11238 11162 11177 11324 11239 11179 11178 1
1240 11182 11173 11328 11241 11183 11242 11188 11235 11161 11228 11234 11332 11312 11246 11199 11309 11207 1
1255 11304 11258 11209 11293 11310 11270 11276 11317 11307 11286 11323 11329 11319 11295 11334 11333 11294 1
1327 11335 11313 11291 11336 11326 11318 11341 11330 11308 11347 11316 11303 11350 11331 11397 11292 11398 1
1342 11399 11337 11400 11338 11401 11339 11402 11340 11403 11344 11348 11406 11356 11353 11407 11392 11349 1
1395 11408 11409 11380 11410 11382 11388 11411 11396 11412 11404 11385 11405 11364 11413 11290 11366 11351 1
1415 11367 11346 11419 11391 11369 11420 11393 11421 11372 11343 11422 11352 11375 11423 11416 11376 11289 1
1424 11384 11417 11425 11386 11418 11426 11390 11430 11427 11365 11431 11428 11361 11432 11429 11374 11433 1
1387 11414 11434 11435 11383 11437 11436 11357 11370 11358 11373 11359 11377 11360 11378 11363 11379 11368 1
1381 11362 11389 11355 11394 11288 11354 11371 11438 11439 11440 11442 11441 11444 11446 11443 11445 11454 1
1447 11453 11456 11459 11448 11451 11455 11466 11449 11457 11450 11465 11463 11458 11452 11460 11467 11464 1
1468 11472 11470 11462 11461 11471 11496 11483 11473 11503 11488 11476 11494 11477 11495 11478 11469 11497 1
1509 11497 11510 11474 11501 11475 11489 11491 11517 11504 11490 11479 11498 11506 11480 11510 11499 11481 1
1511 11519 11500 11482 11512 11493 11502 11484 11507 11492 11505 11514 11485 11508 11547 11513 11486 11516 1
1515 11523 11520 11522 11549 11525 11538 11566 11578 11545 11540 11554 11615 11526 11581 11616 11527 11582 1
1573 11617 11583 11528 11618 11529 11562 11585 11575 11546 11565 11563 11553 11584 11587 11524 11588 1
1556 11589 11535 11521 11591 11560 11592 11557 11619 11531 11594 11533 11599 11569 11620 11595 11621 11596 1
1532 11600 11590 11597 11622 11542 11601 11598 11714 11551 11543 11577 11602 11609 11534 11634 11536 11623 1
1635 11603 11626 11624 11537 11637 11625 11604 11642 11544 11627 11629 11605 11606 11643 11568 11570 11644 1
1607 11548 11541 11645 11571 11646 11608 11530 11572 11647 11610 11539 11628 11574 11631 11550 11652 11640 1
1654 11611 11576 11752 11655 11612 11754 11656 11552 11613 11657 11579 11756 11614 11658 11555 11759 11580 1
1558 11761 11660 11593 11661 11766 11663 11743 11767 11648 11638 11768 11665 11745 11649 11722 11746 11666 1
1771 11559 11747 11667 11773 11650 11632 11668 11748 11774 11769 11669 11742 11777 11651 11749 11670 11778 1
1639 11633 11630 11772 11751 11672 11782 11673 11753 11787 11653 11780 11719 11796 11636 11791 11783 11721 1
1793 11724 11641 11686 11856 11755 11659 11706 11765 11664 11744 11770 11725 11857 11858 11726 11860 11729 1
1859 11750 11730 11862 11671 11781 11674 11731 11863 11855 11734 11867 11844 11735 11868 11846 11835 11870 1
1830 11836 11831 11872 11839 11833 11840 11832 11848 11741 11737 11849 11789 11757 11675 11794 11676 11758 1
1838 11799 11677 11760 11850 11790 11678 11762 11847 11884 11763 11662 11854 11802 11837 11764 11679 11883 1
1841 11740 11680 11804 11681 11842 11805 11682 11843 11806 11683 11807 11684 11808 11845 11685 11811 11851 1
1812 11687 11853 11775 11852 11813 11688 11776 11809 11861 11689 11779 11869 11814 11690 11784 11871 11815 1
1691 11816 11692 11817 11693 11818 11694 11819 11695 11820 11696 11821 11697 11874 11822 11873 11698 11823 1
1699 11824 11825 11701 11826 11702 11827 11703 11828 11704 11561 11705 11564 11707 11567 11739 11708 11785 1
1715 11786 11709 11717 11788 11710 11718 11711 11792 11720 11795 11712 11723 11713 11798 11727 11716 11797 1
1728 11800 11736 11700 11810 11864 11732 11738 11865 11829 11733 11866 11834

```

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
```

root:x:0:0:root:/root:/bin/bash	daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin	sys:x:4:6:sys:/dev:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin	news:x:9:9:news:/var/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin	lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
operator:x:11:11:operator:/root:/bin/bash	irc:x:12:12:irc:/var/spool/irc:/usr/sbin/nologin
postfix:x:83:83:Postfix Mail Service:/var/spool/postfix:/usr/sbin/nologin	sshd:x:65534:65534:/usr/sbin/nologin
[a.out] defunct:x:65535:65535:/usr/sbin/nologin	

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.

```
|
// 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);
    }
}
```