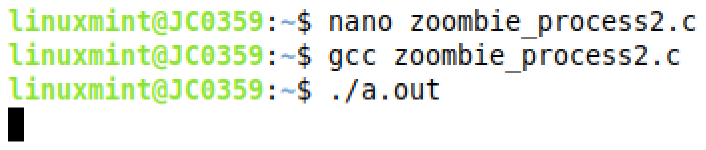
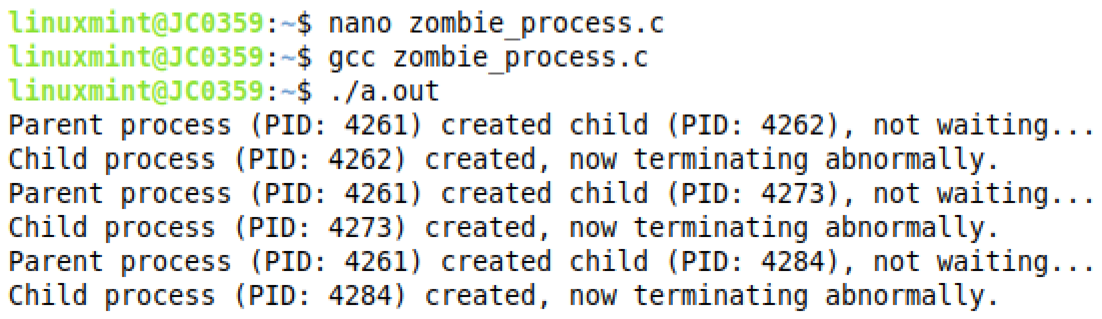
1. If a child terminates before the parent and the parent does not read child's exit status, the kernel does
the following:
(a) Frees memory and closes opened file descriptors used by child.
(b) Saves process ID, terminating status and some other information of the terminated child, so that the parent can read the exit status when the parent is ready.
During this period (the child has terminated but the parent has not read the child's exit status and the parent is still existing), the child process becomes a zombie process. When the parent terminates, the kernel knows that now, no process is going to read the exit status of that child. So the zombie process is removed by the kernel. But if the parent runs for a long time (such as a server process, which might spawn children to do server tasks and terminates only when the system is re-booted), the zombie processes, if generated, can tie up system resources.

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>
int main(void)
{
pid_t x;
while(1) // endless loop to simulate a server
{
// parent creates a new process
x = fork();
if( x == -1 ) { perror("fork"); exit(1); }
if( x == 0 ) // child process
{
// child exits normally
exit(0);
} // end of child process block
// rest is parent process
// parent does not read the exit status of a
// child as the following line is commented
// wait( NULL );
sleep(2);
}
return 0;
}
```

OUTPUT:

```
linuxmint@JC0359:~$ nano zoombie_process2.c
linuxmint@JC0359:~$ gcc zoombie_process2.c
linuxmint@JC0359:~$ ./a.out
█
```

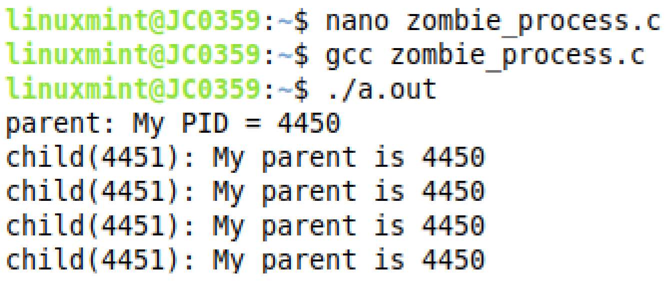
2. The program shows that a child process becomes a zombie process, also on abnormal termination, if the conditions exist.

CODE:
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(void)
{
    pid_t x;

    while (1)
    {
        x = fork();

        if (x == -1)
        {
            perror("fork");
            exit(1);
        }

        if (x == 0) // Child process
        {
            printf("Child process (PID: %d) created, now terminating
abnormally.\n", getpid());
            raise(SIGFPE); // Send SIGFPE to self – terminates abnormally
            exit(1); // Just in case raise fails
        }

        // Parent does NOT call wait(), so child becomes a zombie
        printf("Parent process (PID: %d) created child (PID: %d), not
waiting...\n", getpid(), x);

        sleep(2); // Allow time to observe zombie process
    }

    return 0;
}
```

OUTPUT:

```
linuxmint@JC0359:~$ nano zombie_process.c
linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
Parent process (PID: 4261) created child (PID: 4262), not waiting...
Child process (PID: 4262) created, now terminating abnormally.
Parent process (PID: 4261) created child (PID: 4273), not waiting...
Child process (PID: 4273) created, now terminating abnormally.
Parent process (PID: 4261) created child (PID: 4284), not waiting...
Child process (PID: 4284) created, now terminating abnormally.
```

3. The [init] adopts an orphaned process:
When a process terminates, the kernel checks all the existing processes to find out if the terminated process is the parent of any existing process. In such case, the PPID of the existing process is changed to 1 (PID of init). In this way it is ensured that every process has a parent. If such a child process terminates, it does not become zombie, because [init] arranges to read exit status of all its children (spawned or adopted). The following program illustrates that orphaned processes are adopted by [init]. Though [init] is the parent of these processes, the ownership and the controlling terminal (ttyx) of these processes do not change

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
pid_t y, i;
// main (parent) creates a new process
y = fork();
if( y == -1 ) { perror("fork"); exit(1); }
if( y == 0 ) // child process
{
// child prints it's parent's ID every second
// in an endless loop
while( 1 )
{
i = getppid();
printf("child(%u): My parent is %u \n", getpid(), i );
sleep(1);
}
} // end of child process block
// rest is parent process
printf("parent: My PID = %u \n", getpid() );
// parent exits after 10 seconds
sleep(10);
exit(0); // now the child becomes orphan
}
```

OUTPUT:

```
linuxmint@JC0359:~$ nano zombie_process.c
linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
parent: My PID = 4450
child(4451): My parent is 4450
child(4451): My parent is 4450
child(4451): My parent is 4450
child(4451): My parent is 4450
```

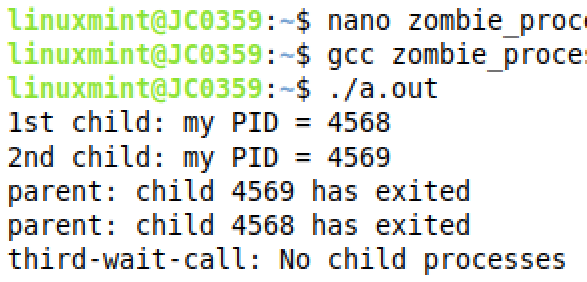
4. The [wait] system call suspends execution of the current process until a child has exited (normally or abnormally). This statement assumes that the parent is not terminated by a

signal. [wait] is not affected if the child process is stopped. [SIGCHLD] signal is delivered to the parent, when a child is stopped or terminated. By default, parent ignores this signal. [wait] system call may be used in signal handler of [SIGCHLD].

The returned value of [wait] is -1, if an error occurs. On success, the process ID of the child which exited, is returned. [wait] call returns immediately (does not block), with -1, if no child process exists.
The following program illustrates that [wait] system call blocks, till the child terminates normally or abnormally. It also shows that [wait] returns immediately with -1, when no child process exists.

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>  // use sys/wait.h, not wait.h

int main(void)
{
    pid_t i;

    // main (parent) creates a process
    i = fork();
    if( i == -1 ) { perror("fork-1"); exit(1); }
    if( i == 0 ) // 1st child process
    {
        printf("1st child: my PID = %u \n", getpid() );
        sleep(30);
        exit(0); // 1st child exits after 30 seconds
    }

    // parent creates another process
    i = fork();
    if( i == -1 ) { perror("fork-2"); exit(2); }
    if( i == 0 ) // 2nd child process
    {
        printf("2nd child: my PID = %u \n", getpid() );
        sleep(20);
        exit(0); // 2nd child exits after 20 seconds
    }

    // rest is parent process

    // parent waits for termination of any child
    i = wait( NULL );
    if( i == -1 ) perror("first-wait-call");
    else printf("parent: child %u has exited \n", i );

    // parent waits for termination of any child
    i = wait( NULL );
    if( i == -1 ) perror("second-wait-call");
    else printf("parent: child %u has exited \n", i );
    // parent waits for termination of any child
    i = wait( NULL );
    if( i == -1 ) perror("third-wait-call");

    return 0;
}
```

OUTPUT:



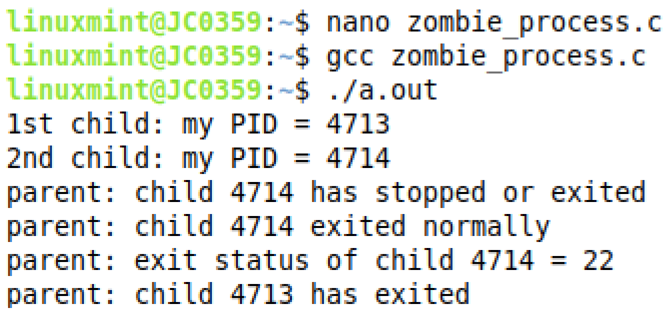
```
linuxmint@JC0359:~$ nano zombie_process.c
linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
1st child: my PID = 4568
2nd child: my PID = 4569
parent: child 4569 has exited
parent: child 4568 has exited
third-wait-call: No child processes
```

5. If a parent has more than one child, [wait] system call returns if any one of the children terminates. If it is required to know when a specific child terminates, [waitpid] system call can be used. This call may be used to return, when the child stops, with [WUNTRACED] option. Read manual [waitpid(2)].
This is a very simple example for [waitpid] system call.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>
int main(void)
{
pid_t i,j,k,m;
// parent creates a process
i = fork();
if( i == -1 ) { perror("fork-1"); exit(1); }
if( i == 0 ) // 1st child process
{
printf("1st child: my PID = %u \n", getpid() );
sleep(30);
exit(0); // 1st child exits after 30 seconds
} // end of 1st child process block
// parent creates another process
j = fork();
if( j == -1 ) { perror("fork-2"); exit(2); }
if( j == 0 ) // 2nd child process
{
printf("2nd child: my PID = %u \n", getpid() );
sleep(20);
exit(22); // 2nd child exits after 20 seconds
} // end of 2nd child process block
// parent process
// parent is waiting for 2nd child. j is the PID of 2nd child
k = waitpid( j, &m, WUNTRACED );
if( k == -1 ) { perror("wait-1"); exit(3); }
printf("parent: child %u has stopped or exited \n", k );

if( WIFSTOPPED(m) != 0 )
printf("parent: child %u has stopped \n", k );
if( WIFEXITED(m) != 0 )
{
printf("parent: child %u exited normally \n", k );
printf("parent: exit status of child %u = %u \n",
k, WEXITSTATUS(m) );
}
if( WIFSIGNALED(m) != 0 )
{
printf("parent: abnormal termination of child %u \n", k );
printf("parent: child %u terminated by signal %u \n",
k, WTERMSIG(m) );
}
k = wait( NULL );
if( k == -1 ) { perror("wait-2"); exit(4); }
printf("parent: child %u has exited \n", k );
return 0;
}
```

OUTPUT:



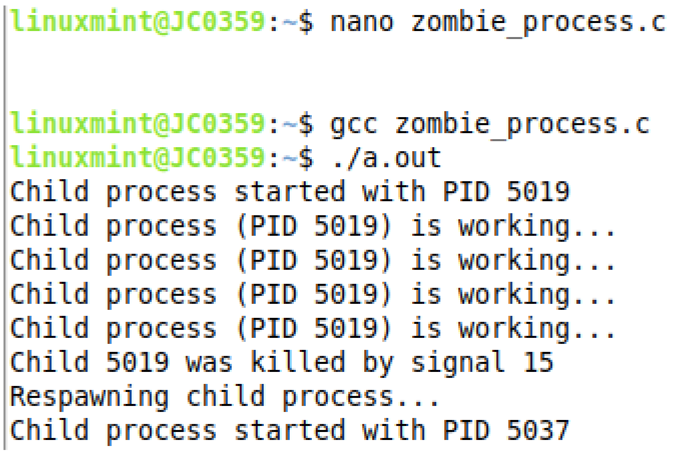
```
linuxmint@JC0359:~$ nano zombie_process.c
linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
1st child: my PID = 4713
2nd child: my PID = 4714
parent: child 4714 has stopped or exited
parent: child 4714 exited normally
parent: exit status of child 4714 = 22
parent: child 4713 has exited
```

6. Write a program to create a child process, make the parent process intelligent enough to re spawn a new child if somehow the existing child being terminated/killed.

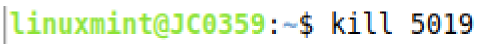
CODE:

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

int main() {
    pid_t pid;

    while (1) {
        pid = fork();

        if (pid < 0) {
            perror("fork failed");
            exit(EXIT_FAILURE);
        }
        if (pid == 0) {
            // Child process
            printf("Child process started with PID %d\n", getpid());
            // Simulate child work: infinite loop (or replace with actual work)
            while (1) {
                sleep(5);
                printf("Child process (PID %d) is working...\n", getpid());
            }
            exit(EXIT_SUCCESS); // Should never reach here
        } else {
            // Parent process
            int status;
            pid_t terminated_pid;

            // Wait for child to terminate
            terminated_pid = waitpid(pid, &status, 0);

            if (terminated_pid == -1) {
                perror("waitpid failed");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(status)) {
                printf("Child %d exited with status %d\n", terminated_pid,
WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("Child %d was killed by signal %d\n", terminated_pid,
WTERMSIG(status));
            } else {
                printf("Child %d terminated abnormally\n", terminated_pid);
            }

            printf("Respawning child process...\n");
            // Loop continues to fork a new child
        }
    }

    return 0;
}
```

OUTPUT:



```
linuxmint@JC0359:~$ nano zombie_process.c


linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
Child process started with PID 5019
Child process (PID 5019) is working...
Child process (PID 5019) is working...
Child process (PID 5019) is working...
Child process (PID 5019) is working...
Child 5019 was killed by signal 15
Respawning child process...
Child process started with PID 5037
```

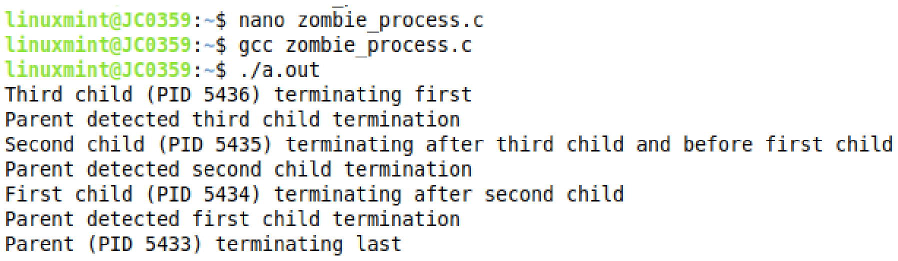


```
linuxmint@JC0359:~$ kill 5019
```

7. Write a Program to create four processes (1 parent and 3 children) where they terminates in a sequence as follows :
1. Parent process terminates at last First child terminates before parent and after second child.
2. Second child terminates after last and before first child.
3. Third child terminates first.

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t first_child, second_child, third_child;
    pid_t pid;

    // Create first child
    first_child = fork();
    if (first_child < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (first_child == 0) {
        // First child
        // Wait for second child to finish before terminating
        // We'll just wait for a signal from parent or sleep since no direct
IPC here
        // Instead, parent waits for 2nd child before allowing 1st to exit
(parent coordination)

        // Sleep to ensure second child terminates before first child
        sleep(5);
        printf("First child (PID %d) terminating after second child\n",
getpid());
        exit(0);
    }

    // Back to parent

    // Create second child
    second_child = fork();
    if (second_child < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (second_child == 0) {
        // Second child
        // Wait for third child to finish before terminating
        sleep(3); // second child sleeps longer than third child to ensure
third finishes first
        printf("Second child (PID %d) terminating after third child and before
first child\n", getpid());
        exit(0);
    }

    // Back to parent

    // Create third child
    third_child = fork();
    if (third_child < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (third_child == 0) {
        // Third child
        // Terminates first, so sleep shortest time
        sleep(1);
        printf("Third child (PID %d) terminating first\n", getpid());
        exit(0);
    }

    // Parent process

    // Wait for children in the correct order
    int status;

    // Wait for third child first
    waitpid(third_child, &status, 0);
    printf("Parent detected third child termination\n");

    // Wait for second child next
    waitpid(second_child, &status, 0);
    printf("Parent detected second child termination\n");

    // Wait for first child last before parent terminates
    waitpid(first_child, &status, 0);
    printf("Parent detected first child termination\n");

    printf("Parent (PID %d) terminating last\n", getpid());

    return 0;
}
```

OUTPUT:



```
linuxmint@JC0359:~$ nano zombie_process.c
linuxmint@JC0359:~$ gcc zombie_process.c
linuxmint@JC0359:~$ ./a.out
Third child (PID 5436) terminating first
Parent detected third child termination
Second child (PID 5435) terminating after third child and before first child
Parent detected second child termination
First child (PID 5434) terminating after second child
Parent detected first child termination
Parent (PID 5433) terminating last
```