Fork System Call

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *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. On success, the PID (process id) of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, and no child process is created.

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);   //pid_t is an int type defined in types.h
```

```
#include <iostream>
using namespace std;
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
   pid_t childPID = fork();

   if (childPID > 0) // fork was successful
   {
      cout << "I am the parent process" << endl;
      pid_t pid= wait(NULL);
      if (pid!=-1)
      {
          Cout<<"The child process with id: "<<pid<<" has terminated"<<endl;
      }

   }
   else if (childPID == 0) //it will be true in the child process only.
   {
      cout << "I am the child process" << endl;
   }
   else // fork failed
   {
      cout << "Fork Failed." << endl;;
      return 1;
   }
   return 0;
}
```

Wait System Call

One of the main purposes of **wait()** is to wait for completion of child processes. This system call blocks the calling process until one of its *child* processes exits. **wait()** takes the address of an integer variable and returns the process ID of the child process. The system call also returns some flags that indicate the completion status of the child process via the integer pointer passed as parameter. The parameter is not necessary for us at the moment so we will pass NULL to the wait system call. The wait system will return if any of the child processes terminates. If we want to wait for a specific process, then we need to use the waitpid system call. If a process has not created a child process and calls the wait system call, then the wait system call will return immediately and -1 will be returned.

```
#include <sys/wait.h>

pid_t wait(int *wstatus);  //we will pass NULL to wstatus
pointer
```

Copy-on-write

It is a technique in which the parent and child initially share the memory resources, such as memory for instructions and memory for global data. However, when one of the processes changes the contents of a shared memory portion, a separate copy of that portion is made, i.e., that portion is no longer shared. This technique helps to save memory.

Zombie Process

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released. (**From Operating Systems Concept, Tenth Edition**)

Orphan Process

Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphan. Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to orphan processes. (Recall from Section 3.3.1 that init serves as the root of the process hierarchy in UNIX systems.) The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry. (**From Operating Systems Concept, Tenth Edition**)