

Tutorial 3

Operating System Lab (**CS341**)

Zombie process

- A zombie process or defunct process is a **process** that has completed execution (via the **exit system call**) but still has an entry in the **process table**:
- The child finishes its execution using `exit()` system call while the parent sleeps
- This occurs for **child processes**, where the entry is still needed to allow the **parent process** to read its child's **exit status**: once the exit status is read via the **wait system call**, the zombie's entry is removed from the process table and it is said to be "reaped".
- Unlike normal processes, the **kill** command has no effect on a zombie process.

Zombie process(cont.)

- When a process ends via `exit`, all of the memory and resources associated with it are deallocated so they can be used by other processes.
- However, the process's entry in the process table remains.
- The parent can read the child's exit status by executing the [wait system call](#), whereupon the zombie is removed.
- The `wait` call may be executed in sequential code, but it is commonly executed in a [handler](#) for the [SIGCHLD signal](#), which the parent receives whenever a child has died.

Orphan process

- Zombie processes should not be confused with orphan processes: an orphan process is a process that is still executing, but whose parent has died.
- These do not remain as zombie processes; instead, (like all orphaned processes) they are adopted by `init` (process ID 1), which waits on its children.
- This operation is called re-parenting and occurs automatically. Even though technically the process has the `init` process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

Orphan process(cont.)

- A process can be *orphaned unintentionally*, such as when the parent process terminates or crashes.
- In this case, user's shell will try to terminate all the child processes with the **SIGHUP** process signal, rather than letting them continue to run as orphans.

Orphan process(cont.)

- A process may also be *intentionally orphaned* so that it becomes detached from the user's session and left running in the background;
- Usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service.
- Under Unix, the latter kinds of processes are typically called daemon processes. The Unix `nohup` command is one means to accomplish this.

Daemon process

- Daemon process is **a process orphaned intentionally**.
- In Unix and other multitasking computer operating systems, a **daemon is a computer program that runs as a background process**,
- Typically daemon names end with the letter d: for example, `syslogd` is the daemon that implements the system logging facility and `sshd` is a daemon that services incoming SSH connections.
- In a Unix environment, the parent process of a daemon is often, but not always, the `init` process.

****Tips**

- For better understanding of any system call , refer the manual page.
 - **Command** :- man <system call>

Important system calls are :- fork(), wait(), exit(), init(), pipe() etc.

Pipe()

- In computer programming, especially in [UNIX](#) operating systems, a pipe is a technique for passing information from one program [process](#) to another.
- Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only.
- Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input.
- The system temporarily holds the piped information until it is read by the receiving process.

Pipe()

- For two-way communication between processes, two pipes can be set up, one for each direction.
- A limitation of pipes for interprocess communication is that the processes using pipes must have a common parent process (that is, share a common open or initiation process and exist as the result of a *fork* system call from a parent process).

Pipe()

- Generally, each pipe has two ends. Each end of the pipe has its own file descriptor.
- One end is for reading and one end is for writing. When you are done with a pipe, it is closed like any other file.

Creating pipe()

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

```
int pipe(int fd[2]);
```

Returns 2 file descriptors in the fd array.i.e. fd[0] is for read and fd[1] write

The system call returns 0 on success and if there is an error.

For more information regarding Pipe() system call, refer manual page.