# **Creating a Daemon Process in C Language with an Example Program**

*by* HIMANSHU ARORA *on* FEBRUARY 24, 2012

A daemon process is *a process which runs in background and has no controlling terminal.*

Since a daemon process usually has no controlling terminal so almost no user interaction is required. Daemon processes are used to provide services that can well be done in background without any user interaction.

For example a process that runs in background and observes network activity and logs any suspicious communication can be developed as a daemon process.

## **1) Daemon Process Design**

A daemon process can be developed just like any other process but there is one thing that differentiates it with any other normal process i.e. having no controlling terminal. This is a major design aspect in creating a daemon process. This can be achieved by:

* Create a normal process (Parent process)
* Create a child process from within the above parent process
* The process hierarchy at this stage looks like :  TERMINAL -> PARENT PROCESS -> CHILD PROCESS
* Terminate the parent process.
* The child process now becomes orphan and is taken over by the init process.
* Call setsid() function to run the process in new session and have a new group.
* After the above step we can say that now this process becomes a daemon process without having a controlling terminal.
* Change the working directory of the daemon process to root and close stdin, stdout and stderr file descriptors.
* Let the main logic of daemon process run.

So we see that above steps mark basic design steps for creating a daemon.

## **2) C fork() Function**

Before creating an actual running daemon following the above stated design steps, lets first learn a bit about the fork() system call.

fork() system creates a child process that is exact replica of the parent process. This new process is referred as ‘child’ process.

This system call gets called once (in parent process) but returns twice (once in parent and second time in child). Note that after the fork() system call, whether the parent will run first or the child is non-deterministic. It purely depends on the context switch mechanism. This call returns zero in child while returns PID of child process in the parent process.

Following are some important aspects of this call:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group.
* The child’s parent process ID is the same as the parent’s process ID.
* The child does not inherit its parent’s memory locks.
* Process resource utilization and CPU time counters are reset to zero in the child.
* The child’s set of pending signals is initially empty.
* The child does not inherit semaphore adjustments from its parent.
* The child does not inherit record locks from its parent.
* The child does not inherit timers from its parent.
* The child does not inherit outstanding asynchronous I/O operations from its parent, nor does it inherit any asynchronous I/O contexts from its parent.

For more insight information, please read the man page of this system call.

## **3) The Implementation**

Based on the design as mentioned in the first section. Here is the complete implementation:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <string.h>

int main(int argc, char\* argv[])

{

FILE \*fp= NULL;

pid\_t process\_id = 0;

pid\_t sid = 0;

// Create child process

process\_id = fork();

// Indication of fork() failure

if (process\_id < 0)

{

printf("fork failed!\n");

// Return failure in exit status

exit(1);

}

// PARENT PROCESS. Need to kill it.

if (process\_id > 0)

{

printf("process\_id of child process %d \n", process\_id);

// return success in exit status

exit(0);

}

//unmask the file mode

umask(0);

//set new session

sid = setsid();

if(sid < 0)

{

// Return failure

exit(1);

}

// Change the current working directory to root.

chdir("/");

// Close stdin. stdout and stderr

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

// Open a log file in write mode.

fp = fopen ("Log.txt", "w+");

while (1)

{

//Dont block context switches, let the process sleep

sleep(1);

fprintf(fp, "Logging info...\n");

fflush(fp);

// Implement and call some function that does core work

}

fclose(fp);

return (0);

}

Following is the way through which the code was compiled and executed:

$ gcc -Wall deamon.c -o deamon

$ sudo ./deamon

process\_id of child process 2936

Just observe that the control immediately came back to the terminal ie the daemon is now not associated to any terminal.

When you check the log.txt file located in the root directory, you could see that this daemon process is running.

$ tail -f /Log.txt

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

Logging info...

* This is an answer of a question on stackoverflow: [**Creating a daemon in Linux**](https://stackoverflow.com/questions/17954432/creating-a-daemon-in-linux/17955149#17955149)
* Fork the skeleton code: [**Basic skeleton of a linux daemon written in C**](https://github.com/pasce/daemon-skeleton-linux-c)
* Read the article here: [How to create a c-style daemon](https://nullraum.net/how-to-create-a-daemon-in-c/)

# Basic skeleton of a linux daemon written in C

Daemons work in the background and (usually...) don't belong to a TTY that's why you can't use stdout/stderr in the way you probably want. Usually a syslog daemon (syslogd) is used for logging messages to files (debug, error, ...) .

Besides that, there are a few required steps to daemonize a process.

## **1) Required Steps**

* **fork** off the parent process & let it terminate if forking was successful. -> Because the parent process has terminated, the child process now runs in the background.
* **setsid** - Create a new session. The calling process becomes the leader of the new session and the process group leader of the new process group. The process is now detached from its controlling terminal (CTTY).
* **Catch signals** - Ignore and/or handle signals.
* **fork again** & let the parent process terminate to ensure that you get rid of the session leading process. (Only session leaders may get a TTY again.)
* **chdir** - Change the working directory of the daemon.
* **umask** - Change the file mode mask according to the needs of the daemon.
* **close** - Close all open file descriptors that may be inherited from the parent process.

Look at this skeleton code that shows the basic steps:

/\*

\* daemonize.c

\* This example daemonizes a process, writes a few log messages,

\* sleeps 20 seconds and terminates afterwards.

\* This is an answer to the stackoverflow question:

\* https://stackoverflow.com/questions/17954432/creating-a-daemon-in-linux/17955149#17955149

\* Fork this code: https://github.com/pasce/daemon-skeleton-linux-c

\* Read about it: https://nullraum.net/how-to-create-a-daemon-in-c/

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <syslog.h>

static void skeleton\_daemon()

{

pid\_t pid;

/\* Fork off the parent process \*/

pid = fork();

/\* An error occurred \*/

if (pid < 0)

exit(EXIT\_FAILURE);

/\* Success: Let the parent terminate \*/

if (pid > 0)

exit(EXIT\_SUCCESS);

/\* On success: The child process becomes session leader \*/

if (setsid() < 0)

exit(EXIT\_FAILURE);

/\* Catch, ignore and handle signals \*/

/\*TODO: Implement a working signal handler \*/

signal(SIGCHLD, SIG\_IGN);

signal(SIGHUP, SIG\_IGN);

/\* Fork off for the second time\*/

pid = fork();

/\* An error occurred \*/

if (pid < 0)

exit(EXIT\_FAILURE);

/\* Success: Let the parent terminate \*/

if (pid > 0)

exit(EXIT\_SUCCESS);

/\* Set new file permissions \*/

umask(0);

/\* Change the working directory to the root directory \*/

/\* or another appropriated directory \*/

chdir("/");

/\* Close all open file descriptors \*/

int x;

for (x = sysconf(\_SC\_OPEN\_MAX); x>=0; x--)

{

close (x);

}

/\* Open the log file \*/

openlog ("firstdaemon", LOG\_PID, LOG\_DAEMON);

}

int main()

{

skeleton\_daemon();

while (1)

{

//TODO: Insert daemon code here.

syslog (LOG\_NOTICE, "First daemon started.");

sleep (20);

break;

}

syslog (LOG\_NOTICE, "First daemon terminated.");

closelog();

return EXIT\_SUCCESS;

}

## **2) Compile and run**

* Compile the code: gcc -o firstdaemon daemonize.c
* Start the daemon: ./firstdaemon
* Check if everything is working properly: ps -xj | grep firstdaemon

## **3) Test the output**

* The output should be similar to this one:

+------+------+------+------+-----+-------+------+------+------+-----+

| PPID | PID | PGID | SID | TTY | TPGID | STAT | UID | TIME | CMD |

+------+------+------+------+-----+-------+------+------+------+-----+

| 1 | 3387 | 3386 | 3386 | ? | -1 | S | 1000 | 0:00 | ./ |

+------+------+------+------+-----+-------+------+------+------+-----+

**What you should see here is:**

* The daemon has no controlling terminal (**TTY = ?**)
* The parent process ID (**PPID**) is **1** (The init process)
* The **PID != SID** which means that our process is NOT the session leader  
  (because of the second fork())
* Because PID != SID our process **can't take control of a TTY again**

**Reading the syslog:**

* Locate your syslog file. Mine is here: /var/log/syslog
* Do a: grep firstdaemon /var/log/syslog
* The output should be similar to this one:

firstdaemon[3387]: First daemon started.

firstdaemon[3387]: First daemon terminated.

**A note:** In reality you would also want to implement a signal handler and set up the logging properly (Files, log levels...).

**Further reading:**

* [Linux-UNIX-Programmierung - German](http://openbook.galileocomputing.de/linux_unix_programmierung/Kap07-000.htm#Xxx999234)
* [Unix Daemon Server Programming](http://www.enderunix.org/docs/eng/daemon.php)