Assignment 10

(1) **(1 pt)** Call the daemonize function correctly.

Answer: We call the `daemonize` function using a command name:

```
daemonize("my-daemon-logs");
```

(2) **(1 pt)** Create a text file called "`assignment11.txt`", use `getlogin()` function to get the login name, and write "*Login name: [login name]*" in "`assignment11.txt`".

Answer: After `daemonize` is called, we get the login id and open the requisite file in `/` in order to write out the login name.

```
 char* login = getlogin();
 FILE* fl = fopen("assignment11.txt","w");
 if(fl) {
   fprintf(fl,"Login name: %s",login);
 } else {
     syslog(LOG_ERR, "Error on opening assignment11.txt, Loging
name: %s", login);
  puts("Error opening assignment11.txt");
}
```

(3) **(1 pt)** Explain the purpose of every step executed in the daemonize function.

Answer: The daemon follows certain coding rules to perform the full process of daemonization.

First the file mode creation mask is set to 0, in order to provide the daemon with all the permissions it requires to create files.

```
umask(0);
```

The resource limits are queried in order to make sure that the daemon can avoid consuming any resources or inherit them.

```
if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
  err_quit("%s: can't get file limit", cmd);
```

The daemon then becomes session leader in order to lose controlling tty. This prevents the daemon from interfering with tty output and keep it remaining in the background.

This call to `fork()` inside a conditional makes sure that the parent closes before the child, so the daemon can freely create its own session.

```
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);
setsid();
```

For the same reasons, the daemon ignores ass possible signals, so that new connections to controlling terminals are not created.

```
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);

sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: can't ignore SIGHUP", cmd);
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);
```

The working directory is set to `/`, the root directory. This step isn't completely necessary, often a daemon will switch to the directory where its work is required, like a folder of logfiles. This is also a precaution in case the daemon is working on an alternate filesystem.

```
if (chdir("/") < 0)
  err_quit("%s: can't change directory to /", cmd);
```

The daemon then closes all open file descriptors in order to make sure that it does not hold up those resources. Similarly, it also sets descriptors 0,1,2 to `/dev/null` so that any I/O to those streams cannot happen by accident.

```
if (rl.rlim_max == RLIM_INFINITY)
  rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);

fd0 = open("/dev/null", O_RDWR);
fd1 = dup(0);
fd2 = dup(0);
```

Finally, it opens a log file under the name of the daemon. A daemon may open more than one log file, usually with a number at the end.

```
openlog(cmd, LOG_CONS, LOG_DAEMON);
if (fd0 != 0 || fd1 != 1 || fd2 != 2)
{
    syslog(LOG_ERR, "unexpected file descriptors %d %d %d", fd0,
fd1, fd2);
    exit(1);
}
```

(4) **(1 pt)** Discuss what would happen to the process after becoming a **daemon process**.

Answer: A daemon process is intended to be a long running process with low resource usage that runs till the system is shut down. It will continue to stay in the background being a non-interfering process unless its purpose is different from that.

Usually daemons hold a log file or a folder for their work and write all the requisite data within that area. They ae usually run with superuser privileges so they can perform their work outside the userspace, often repetitive cleanup tasks and low level system work that should generally remain invisible to the user.

Daemons (for example, `cron`) can be seen running in the background with the `ps` command, generally `ps -efj` can be used for viewing.